Leiden University

Computer Science

A Parallel Relation-Based Algorithm for Symbolic Bisimulation Minimization

Name:	Richard Huybers
Date:	26/10/2018
1st supervisor:	Dr. A.W. Laarman
2nd supervisor:	Prof.dr. H.A.G. Wijshoff

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

A Parallel Relation-Based Algorithm for Symbolic Bisimulation Minimization

Richard Huybers

Abstract

Symbolic computation using BDDs and bisimulation minimization are alternative ways to cope with the state space explosion in model checking. The combination of both techniques opens up many parameters that can be tweaked for further optimization. Most importantly, the bisimulation can either be represented as equivalence classes or as a relation. While recent work argues that storing partitions is more efficient, we show that the relation-based approach is preferable. We do so by presenting a relation-based minimization algorithm based on new coarse-grained BDD operations. These coarse-grained BDD operations are derived directly from their definition using Shannon decomposition, guaranteeing correctness. The implementation demonstrates that the relational approach uses two orders of magnitude fewer memory, performs better and has better parallel scalability.

Contents

Abstract **i**

1 Introduction **1**

2 Background **4**

2.1 Model checking 4

2.2 Labelled transition systems 6

2.3 Bisimulation 7

2.4 Binary decision diagrams 8

2.4.1 Definition 9

2.4.2 Operations 12

2.4.3 Variable Ordering 16

3 Symbolic Bisimulation **18**

3.1 A relation-based symbolic bisimulation algorithm 18

3.2 Computing pairs of non-bisimilar states 20

3.3 Computing the converse of a relation 25

3.4 Parallel breadth first search extension 27

4 Related Work **30**

5 Experimental Evaluation **34**

5.1 Implementation 34

5.2 Experiments 35

5.3 Results 35

5.4 Validation 37

6 Conclusions **38**

A Program Verification using Dafny **39**

A.1 Dafny . . . 39
A.2 The verification of the union and intersection operations . . . 40
A.3 The verification of the image and preimage operations . . . 42

Bibliography **48**

Chapter 1

Introduction

Verification is an important part of both hardware and software development. As computing systems take over more and more vital roles in our society, including safety-critical applications such as ABS, the importance of verification will only increase. Verification is often accompanied by validation. The principal contrast between these two processes is that conventional validation methods, such as testing, are straightforward but incomplete, while traditional verification methods, such as theorem proving, are thorough but require a great deal of input and skill from the user.

Model checking [4] is an verification method that is automatic (push-button technology), exhaustive, and more accessible than theorem proving. In model checking, a transition system which models the behavior of a system is analysed to decide whether the system meets its formal specification. The method can be applied to both hardware and software systems which contain a finite number of states or have a finite state abstraction. The downside of model checking is that it is very resource intensive. For some problems, it is therefore incomplete in practice. Fortunately, reduction techniques exist which alleviate the required resources.

The main challenge faced in model checking is the state explosion problem. The state space of a system grows exponentially in the number of variables and processes which are present in the system. Over the years, various techniques have been suggested to combat this problem. McMillan et al. [9] introduced a very successful method which represents the state space of a system symbolically rather than explicitly. In a symbolic representation, states are no longer stored individually, but instead as sets which are represented by their characteristic function. These characteristic functions can be efficiently stored and manipulated in a normal form, e.g. binary decision diagrams (BDDs) [8]. Symbolic algorithms can directly manipulate sets represented by BDDs using basic set operations, such as union, intersection, and subtraction. This allows symbolic algorithms to manage state spaces that are many orders of magnitude larger than the state spaces handled by explicit algorithms [9].

Orthogonally, reduction techniques can be used to further reduce the size of state spaces by removing redundant states. Reduction is applied either as a pre-processing step to, or happens "on-the-fly" with, the model checking process [5]. One prominent reduction technique is bisimulation minimization [23]. State spaces of concurrent systems often contain many *bisimilar* states. These are different states with the same behavior in the represented system. Bisimulation minimization algorithms reduce the size of state spaces by merging these bisimilar states. To achieve this it computes the maximal bisimulation over the state space, which is the largest relation that relates bisimilar states. Like model checking, bisimulation minimization is an automatic process. Symbolic representations can be combined with bisimulation to handle even larger state spaces. This thesis focusses on the combination of both techniques.

The maximal bisimulation is an equivalence relation and can therefore be represented symbolically in two ways: either directly as a *relation* or as a *partition* which divides the state space into equivalence classes. It is stated by Fisler and Vardi [16] that symbolic *relation-based* algorithms are not efficient, because in practice the BDD that represents the equivalence relation grows too large. However, satisfying results for symbolic relation-based algorithms have been found in the past by Bouali and de Simone [6] and by Mumme and Ciardo [24]. Little research has been done to compare the two approaches. Thus, it remains unclear whether the relation- or partition-based approach is preferable.

For a long time, symbolic algorithms were thought to be difficult to parallelize. However, new results by Van Dijk and Van de Pol [35] show that the parallelization of symbolic algorithms can certainly be profitable. Interest in parallel algorithms has been revitalized in recent years due to the increased power consumption of sequential processors. Yet, Van Dijk's algorithm is currently the only known parallel algorithm for symbolic bisimulation minimization. Van Dijk's algorithm is partition-based, and therefore a parallel symbolic relation-based algorithm does not yet exist. In this thesis, we derive and implement a parallel symbolic relation-based bisimulation algorithm and compare its performance to Van Dijk's algorithm. Van Dijk states that BDDs representing the bisimulation as a partition, like the one used in his algorithm, should generally remain smaller than relational BDDs. However, our results show that the opposite is true. Relation-based approaches use fewer memory and perform better than the partition-based approaches. We argue that the variable interleavings allowed in the relational approach can explain these unexpected results.

Our algorithm is similar to the algorithm proposed by Mumme and Ciardo, but was found independently. We use a derivation method based on Shannon decomposition [28] to derive principal parts of the algorithm directly from the definition of the symbolic operations which they perform. This derivation method gives more insight into the inner workings of the operations. Moreover, the derived operations are coarse, avoiding BDD unique and cache table pollution by skipping intermediary results, similar to what has been done before for image computation [34].

This thesis is structured as follows. In Chapter 2 we will explain the notions of model checking, labelled transition systems, bisimulation minimization, and BDDs in more detail. Chapter 3 presents the parallel relation-based symbolic bisimulation algorithm, along with its derivation. Related work, and in particular van Dijk's algorithm, are discussed in Chapter 4. Our experiments and their results are presented in Chapter 5. Chapter 6 concludes this thesis with a discussion. The verification of symbolic algorithms using the theorem prover Dafny [21] is described in the appendix.

Chapter 2

Background

2.1 Model checking

Model checking is a Turing award winning formal verification technique which was developed independently by Clarke and Emerson [10] and by Queille and Sifakis [26] in the early 1980s. Model checking exhaustively and automatically verifies a model of a hardware or software system against a given specification [4]. The problem faced in model checking is formally expressed as the satisfaction relation $M \models \varphi$. Here M is a model of the system under study and φ is its property. The question, in the model-theoretic sense, is whether the formula φ evaluates to true or false in the model M. Semantically, the model is interpreted as an implicit transition system (see Section 2.2) whose reachable states, called the state space, represents the complete behavior of the system. Model checkers exhaustively search this state space to verify whether the given system conforms to its properties. This procedure can be performed systematically by an algorithm, and is thus fully automatic. If the verification of a model fails, a model checker produces a counterexample so that the model can be corrected and the process reiterated.

Systems are generally modelled in some formal language, which can even be a programming language. The states in a transition system which interprets a model of a system are derived from different variables valuations in the system. For instance, the state of a computer program is described by the values of its variables and program counters. Certain actions or events within a system can change the values of its variables. The transitions in the transition system are derived from these actions and events. For instance, in a computer program the execution of a program statement is an event which could change the values of some variables and program counters.

The specification of a system is written in some form of logic, such as linear temporal logic (LTL) or computation tree logic (CTL) [20]. Temporal logic can define properties over the paths and future states within a transition system. For example, for a transition system representing a traffic light, temporal logic can define properties such as “when a car arrives eventually the light should become green”. Most properties can be reduced to a reachability problem, in which we check whether a set of critical states is reachable from the initial states in the model. The sliding tile puzzle in Example 1 shows an example of a small model and its possible properties.

Example 1. *A sliding tile puzzle is a combination puzzle in which the player has to construct an image by rearranging square tiles that depict parts of the image. The tiles are lain on a rectangular grid in which one space is left empty. The player can rearrange the tiles by sliding adjacent tiles into the empty space.*
The sliding tile puzzle considered in this example consists of a 2 by 2 grid and three identical tiles. Figure 2.1 depicts a transition system which models this sliding tile puzzle. The tiles can be lain in four different arrangements. Each arrangement may be seen as a different state of the puzzle. In each state one tile can be slid horizontally (h) and one tile can be slid vertically (v). By sliding a tile, the state of the puzzle changes.

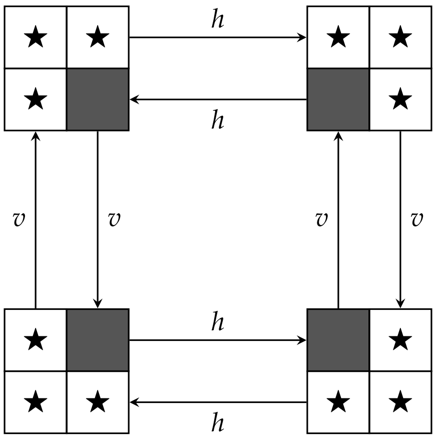


Figure 2.1: A model displaying the four arrangements of a simple sliding tile puzzle, along with the possible moves for each arrangement.

A simple property of this model could be that all states should contain exactly three tiles. A more difficult property could be that the system can always return to its starting state.

Model checking has many advantages. The exhaustive nature of model checking makes it more reliable than valdidation techniques, such as testing or simulation, which are incomplete. The fact that model checking is automatic makes it more practical than formal verification techniques, such as theorem proving, which requires a lot of input and skill from the user. The main disadvantage of model checking is the state explosion problem. The state space of a transition system grows exponentially large as the number of variables and processes in a system increases. This makes model checking highly resource intensive, and as a result incomplete for some problems. Techniques such as partial order reduction [17], symmetry reduction [11], bisimulation minimization [23], and the use of symbolic representations [9] can be applied to alleviate the state explosion problem.

2.2 Labelled transition systems

A labelled transition system, or LTS for short, is a directed graph with labelled edges. LTSs are commonly used in model checking to model different kinds of systems. The nodes of an LTS represent the system's states. The labelled edges represent actions or events which cause the system to transition to another state.

Definition 1. A *labelled transition system (LTS)* is a tuple $(S, A, \rightarrow)$ consisting of a set of states S, a set of action labels A, and a transition relation $\rightarrow \subseteq S \times A \times S$. Instead of $(s, a, t) \in \rightarrow$, we write $s \xrightarrow{a} t$.

Example 2. *Figure 2.2 demonstrates an LTS which represents the sliding tile puzzle introduced in Example 1. Each state of the puzzle is represented by a node and each move is represented by a labelled transition. Formally, the LTS consists of the following sets:*

$$\begin{aligned} S &= \{s_1, s_2, s_3, s_4\} \\ A &= \{h, v\} \\ \rightarrow &= \{(s_1,h,s_2),(s_1,v,s_3),(s_2,h,s_1),(s_2,v,s_4),(s_3,h,s_4),(s_3,v,s_1),(s_4,h,s_3),(s_4,v,s_2)\} \end{aligned}$$

$$\begin{tikzcd} s_1 \arrow[r, bend left, "h"] \arrow[d, bend right, "v"'] & s_2 \arrow[l, bend left, "h"] \arrow[d, bend right, "v"'] \\ s_3 \arrow[u, bend right, "v"'] \arrow[r, bend left, "h"] & s_4 \arrow[u, bend right, "v"'] \arrow[l, bend left, "h"] \end{tikzcd}$$

Figure 2.2: An LTS which models the sliding tile puzzle depicted in Figure 2.1.

An LTS that models a system can be automatically generated from a model description. For hardware systems, this description is formulated in a hardware description language such as Verilog or VHDL. For software systems, a (dialect of a) programming language such as C or Java can be used.
States in an LTS can be represented in many forms and do not have to be numbered. In fact, it is often better not to number states, but to instead use an encoding which preserves some structural properties. This is important for BDD representations of the LTS, as will be discussed in Section 2.4.3.

2.3 Bisimulation

An LTS can contain many redundant states. This is undesirable, as this limits the size of the system that a model checker can handle. For instance, redundant states are frequently present in LTSs which model concurrent systems. The order in which multiple concurrent events happen usually does not matter for the resulting system state. However, due to different variable valuations these similar behaving states might be physically different. This causes the LTS to contain multiple states which essentially represent the same system state from a behavioral perspective. Fortunately, we can find and remove these redundant states by computing a bisimulation relation. Definition 2 formally defines a bisimulation.

Definition 2. A binary relation $R \subseteq S \times S$ over the set of states of an LTS $(S, A, \rightarrow)$ is a *bisimulation* relation if whenever $(s, t) \in R$ and $a \in A$,

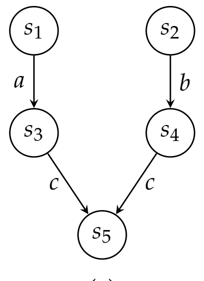
1. for all s' with $s \xrightarrow{a} s'$, there exists a t' such that $t \xrightarrow{a} t'$ and $(s', t') \in R$
2. for all t' with $t \xrightarrow{a} t'$, there exists a s' such that $s \xrightarrow{a} s'$ and $(s', t') \in R$

We say that s is bisimilar to t and write $s \sim t$ if there exists a bisimulation R with $(s, t) \in R$.

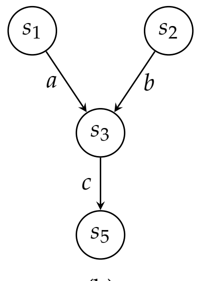
A bisimulation relation defines the set of pairs of states (s, t) that have a "similar behavior" within an LTS. For every outgoing transition of s we can find an outgoing transition for t that represents the same change in the system modelled by the LTS, and vice versa. As a consequence, two bisimilar states can be merged without changing the system which the LTS represents. This is illustrated in Example 3.

Example 3. *Consider the LTS shown in Figure 2.3a. First off, note that every state is always bisimilar to itself. Consequently, states s_3 and s_4 are also bisimilar as their outgoing transitions, (s_3, c, s_5) and (s_4, c, s_5), have the same label and $s_5 \sim s_5$. Thus, it is possible to merge these two states. This is done by first adding all incoming transitions of s_4 to s_3 and then removing s_4 from the LTS. The result is depicted in Figure 2.3b. Observe that the incoming transitions of a state do not affect its behavior.*

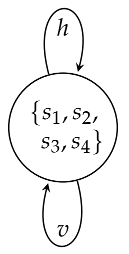
Example 4. *Figure 2.3c shows the bisimulation minimization of the LTS depicted in Figure 2.2.*



(a)



(b)



(c)

Figure 2.3: An LTS (a) and its minimized equivalent (b). And the minimization of Figure 2.1.

When the bisimilarity relation $\sim$ is the largest possible bisimulation relation over an LTS, it is more commonly known as the *maximal bisimulation*. Because the maximal bisimulation is also an equivalence relation (see Theorem 1), the size of an LTS can be minimized by computing the maximal bisimulation and merging all states that are in the same equivalence class. This practice is also called *bisimulation minimization* and is one of the techniques used in model checking to combat the state explosion problem.

Theorem 1. *The maximal bisimulation is an equivalence relation.*

Proof. 1) Reflectivity. It follows directly from Definition 2 that the identity relation $I = \{(s, s) \mid s \in S\}$ is a bisimulation relation.
2) Symmetry. Let R be a bisimulation relation. It follows directly from Definition 2 that the converse relation R^{-1} is also a bisimulation relation.
3) Transitivity. Let R_1 and R_2 be two bisimulation relations. We will show that the composition $R_1 \circ R_2$ is also a bisimulation relation. Let $(s, t) \in R_1 \circ R_2$ and $s \xrightarrow{a} s'$. For $R_1 \circ R_2$ to be a bisimulation relation, there must exist a t' such that $t \xrightarrow{a} t'$ and $(s', t') \in R_1 \circ R_2$. Since $(s, t) \in R_1 \circ R_2$, there is a r such that $(s, r) \in R_1$ and $(r, t) \in R_2$. Because R_1 is a bisimulation relation, there exists a r' such that $r \xrightarrow{a} r'$ and $(s', r') \in R_1$. Also, as R_2 is a bisimulation relation, there exists a t' such that $t \xrightarrow{a} t'$ and $(r', t') \in R_2$. Since $(s', r') \in R_1$ and $(r', t') \in R_2$, we find that $(s', t') \in R_1 \circ R_2$. Thus, for t' we have that $t \xrightarrow{a} t'$ and $(s', t') \in R_1 \circ R_2$. □

The maximal bisimulation is generally computed using a fixed point algorithm. Given an LTS $(S, A, \rightarrow)$, relation-based approaches construct a relation R which contains pairs of states in S that are *not* bisimilar. Pairs of non-bisimilar states are iteratively added to R, until such pairs can no longer be found. When this fixed point is reached, the relation $S \times S \setminus R$ contains the maximal bisimulation. The relation-based approach is discussed in more detail in Chapter 3. Partition-based approaches manage a partition P which initially contains the state space as its only subset. The partition is iteratively refined by assigning signatures to all states. Bisimilar states are given the same signature. States with the same signature are grouped together, refining the previous partition. This process continues until a fixed point is reached. The partition P now represents the maximal bisimulation as a set of equivalence classes. The partition-based approach is further explained in more detail in Chapter 4.

2.4 Binary decision diagrams

Explicit representations of an LTS, such as those utilizing lists or tables, grow in proportion to the size of the LTS. This is why the state explosion problem is such a big issue in model checking. A more efficient way to store an LTS is by a symbolic representation. In a symbolic representation, states and transitions are not stored individually, but rather as sets in a memory efficient data structure called a binary decision diagram, or BDD for short. BDDs were introduced by Bryant [8] as canonical representations of Boolean functions that could be easily manipulated. A set of states or a set of transitions is represented in a BDD by their characteristic function. The biggest asset of BDDs is that set operations can be directly applied to the data structure without the need of decompression. This feature allows us to create set-based algorithms, also known as symbolic algorithms, which can efficiently handle much larger systems.

McMillan et al. [9] were the first to introduce a symbolic algorithm for model checking. Their algorithm could be applied to systems with a state space of more than 10^{20} reachable states, while explicit state enumeration methods at the time could only manage state spaces of up to 10^8 reachable states. Symbolic algorithms can also be employed in bisimulation minimization, as will be demonstrated in Chapter 3. First, this section will discuss the structure and functionalities of BDDs in greater detail.

2.4.1 Definition

A Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ takes $n \geq 0$ arguments $x_1, x_2, \ldots, x_n \in \{0, 1\}$ and returns either a 0 or a 1. We can restrict f with respect to one of its arguments x_i by assigning the value 0 or 1 to it, notation $f_{\overline{x}_i}$ and f_{x_i} respectively, such that

$$f_{\overline{x}_i}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \quad \text{and} \quad f_{x_i}(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

Here $f_{\overline{x}_i}$ and f_{x_i} are also called the *cofactors* of f with respect to x_i.

We can abstract a variable x_i from f using Shannon decomposition [28] according to the following equivalence

$$f(x_1, \ldots, x_n) \Leftrightarrow \overline{x}_i \cdot f_{\overline{x}_i} + x_i \cdot f_{x_i}$$

By abstracting all variables from a function we find a structure which represents a binary tree. This can be seen in Figure 2.4 for the function $f(x_1, x_2, x_3) = \overline{x}_1 + x_2 x_3$.

Definition 3 defines the BDD data structure. A BDD represents a Boolean function as its Shannon decomposition. More explicitly, a BDD node with variable label x expresses a function f recursively as the disjunction $\overline{x} \cdot f_{\overline{x}} + x \cdot f_x$. Here $f_{\overline{x}}$ and f_x are the low and high child of the node respectively. Any node in a BDD is essentially the root of a sub-BDD representing its own Boolean function. The order in which the variables are abstracted determines the variable ordering used in the BDD. The variables over which a BDD is defined are usually expressed using a vector notation: $\vec{x} = x_1, \ldots, x_n$, where we assume the order $x_1 < \ldots < x_n$

Definition 3. A *binary decision diagram (BDD)* is a rooted directed acyclic graph which is made up of internal decision nodes and leaves with value 0 or 1. Internal nodes have as attributes a variable label and two child nodes called its "low" and "high" child.

$$f(x_1, x_2, x_3) = \overline{x}_1 + x_2x_3 \Leftrightarrow \overline{x}_1 \cdot f_{\overline{x}_1}(x_2, x_3) + x_1 \cdot f_{x_1}(x_2, x_3)$$

$$f_{\overline{x}_1}(x_2, x_3) = 1$$

$$f_{x_1}(x_2, x_3) = x_2x_3 \Leftrightarrow \overline{x}_2 \cdot f_{x_1\overline{x}_2}(x_3) + x_2 \cdot f_{x_1x_2}(x_3)$$

$$f_{x_1\overline{x}_2}(x_3) = 0$$

$$f_{x_1x_2}(x_3) = x_3 \Leftrightarrow \overline{x}_3 \cdot f_{x_1x_2\overline{x}_3}() + x_3 \cdot f_{x_1x_2x_3}()$$

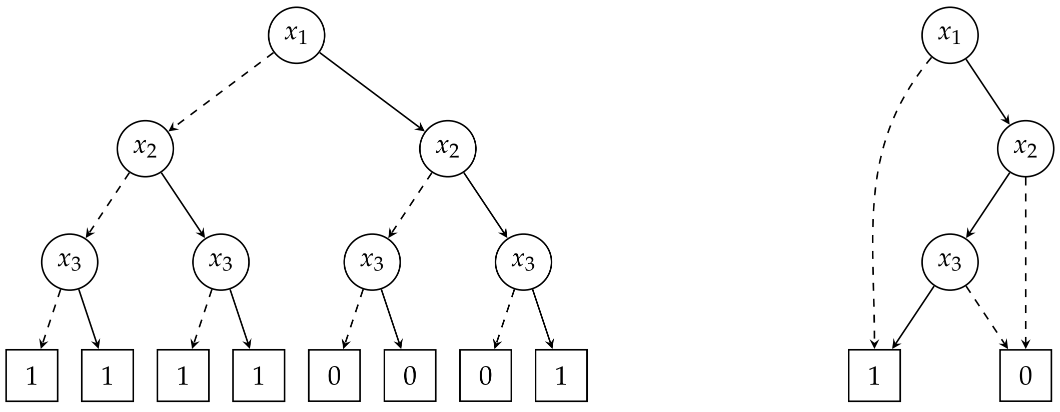
$$f_{x_1x_2\overline{x}_3}() = 0$$

$$f_{x_1x_2x_3}() = 1$$

Figure 2.4: The Shannon decomposition of $f(x_1, x_2, x_3) = \overline{x}_1 + x_2x_3$

A BDD is called *ordered* if the variable labels are encountered in the same order for every directed path down the root. A BDD is called *reduced* if it contains no redundant nodes (nodes with the same low and high child) and no duplicate nodes (two nodes with the same label, low child, and high child).

Figure 2.5 shows two BDDs representing $f(x_1, x_2, x_3) = \overline{x}_1 + x_2x_3$. The BDD on the right hand side is reduced, the BDD on the left hand side is not. Observed how the structure of the reduced BDD is related to the Shannon decomposition in Figure 2.4.
Given a fixed variable ordering, a reduced ordered BDD is a canonical representation of a Boolean function. From this point forward we will use the term BDD to refer to reduced ordered BDDs, as is common in the literature.

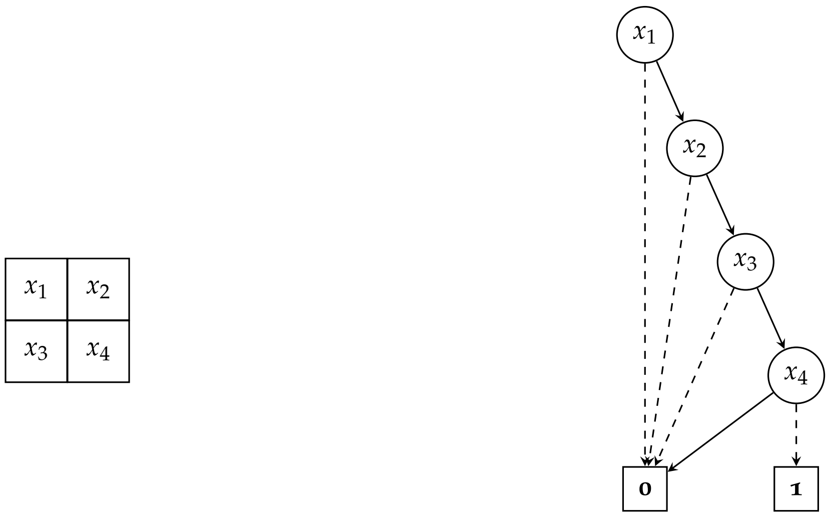


(a) (b)

Figure 2.5: A non-reduced (a) and reduced (b) BDD representing the function $f(x_1, x_2, x_3) = \overline{x}_1 + x_2x_3$

The low and high child of an internal node can be found by following the dashed and solid arrows originating in the node respectively. Paths which go from the root of a BDD down to a leaf specify different variable assignments. Whenever we traverse from a node to its low (or high) child, we assign 0 (or 1) to the variable contained in the node. If a path jumps over a certain variable, then it does not matter which value is assigned to the variable. For instance, in the BDD shown in Figure 2.5b the path $(x_1, 1)$ skips variables x_2 and x_3. The value in the leaf at the end of a path determines the return value of the function represented by the BDD for the corresponding variable valuation. In essence, non-reduced BDDs are binary trees which enumerate all possible valuations of arguments for a Boolean function. Reduced BDDs remove from the binary tree all isomorphic subgraphs and all nodes for which the low and high edges point to the same node.

Instead of saying that a BDD represents a Boolean function, we could also say that it symbolises a set of binary strings. Namely, the set of variable assignments for which the corresponding path ends in a 1-leaf. This is the satisfying set of the function represented by the BDD. For example, the BDDs shown in Figure 2.5 represents the set of strings $\{000, 001, 010, 011, 111\}$. The notion that a BDD represents a Boolean function or a set of binary strings can be used interchangeably. In model checking and bisimulation minimization, BDDs are used as representations of sets of states or sets of transitions. The states and transitions of an LTS are physically represented as variable valuations from the system under verification. By "bit blasting" any such state can be represented as a valuation of Boolean variables and stored in a BDD. This approach is quite memory efficient, as encodings with identical prefixes are represented by overlapping paths in the BDD. Example 5 illustrates how the sliding tile puzzle from Example 1 can be represented by a BDD.



(a) The four variables which encode a state of the puzzle.

(b) The BDD for $\{s_1\}$.

Figure 2.6

Example 5. *The states of the LTS which represents the sliding tile puzzle from Example 1 can be encoded by strings of four binary variables, $x_1x_2x_3x_4$. Each variable corresponds to a position on the puzzle grid, as shown in Figure 2.6a. A variable is set to 1 if a tile is present in the corresponding grid position. Otherwise, the variable is set to 0. For instance, the state s_1 is encoded by the string 1110. A state like s_1 can be stored individually in a BDD by letting the BDD represent the singleton set $\{s_1\}$, as shown in Figure 2.6b. For the string 1110 the path from the root to the 1-leaf consists of three solid arrows followed by one dashed arrow. Observe that this is the only path to the 1-leaf. Thus, the BDD represents no other states than s_1.*

Using BDDs to represent individual states is costly and defeats the purpose of BDDs, which is to efficiently represent sets. It would be better to represent the entire state space $S = \{s_1, s_2, s_3, s_4\}$ by one BDD, as is done in Figure 2.7. This BDD contains four paths from the root to the 1-leaf, representing the set of all state encodings $\{1110, 1101, 1011, 0111\}$. Instead of four BDDs containing four internal nodes each, we now have one BDD containing just seven internal nodes. The transitions of the LTS can be encoded by strings of nine variables: four variables for the source state, four variables for the target state, and one variable for the action label. This BDD is quite large and is therefore not shown.

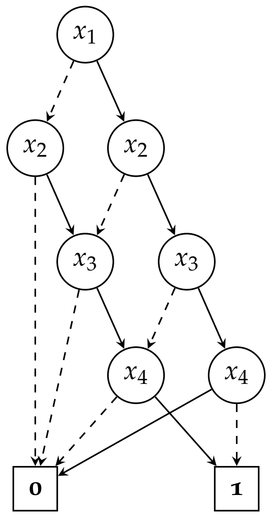


Figure 2.7: The BDD which represents the state space S of the sliding tile puzzle.

2.4.2 Operations

Another reason why BDDs make for such great set representations is that binary set operations, such as union ($\cup$) and intersection ($\cap$), can all be directly applied to the data structure. Similarly, we have disjunction ($\vee$) and conjunction ($\wedge$), for BDDs representing Boolean functions. The required time of these operations is linear in the number of nodes in the result BDD, whereas the BDD might represent an exponential number of assignments in the number of variables. Given two BDDs F and G, and a binary set operation <op>, we can compute a BDD representing the result of F <op> G using Algorithm 1. This recursive algorithm is based on the Shannon decomposition of F <op> G.

$$F \texttt{ <op> } G \Leftrightarrow \overline{x} \cdot (F_{\overline{x}} \texttt{ <op> } G_{\overline{x}}) + x \cdot (F_x \texttt{ <op> } G_x)$$

It recursively traverses F and G at the same time, creating nodes in the result BDD as it backtracks. The algorithm assumes that F and G utilize the same variable ordering. The first line in the algorithm checks for the terminal case where F and G are both leafs. Here a leaf containing the value of F <op> G is returned. If F or G is not a leaf, then Line 3 determines the variable label x of the highest node in F and G. Lines 4–6 create a node with label x whose children are the results of the recursively calls to $F_{\overline{x}}$ <op> $G_{\overline{x}}$ and F_x <op> G_x. This node represents the Shannon decomposition of F <op> G with respect to x. Note that if the variable label of the highest node in F (or G) is ordered after x, then $F_{\overline{x}} = F_x = F$ (or $G_{\overline{x}} = G_x = G$) because the BDD contains no nodes with label x. In summary, this algorithm builds up a BDD which follows the structure of the Shannon decomposition of F <op> G and has the same variable ordering as F and G.

To ensure that the result BDD is reduced, a table, also known as the *unique table*, is used to store every unique node that is created. The call to BDD_node in Line 6 checks whether the node that needs to be created already exists in the unique table before constructing a new one. It also makes sure that no redundant nodes are created. If the low and high child of a new node are the same, BDD_node returns the low child instead of creating a new node. In order to make the operation linear in the number of BDD nodes, another table, called the *operation cache*, is used to store the result of every function call. In Line 2 we check whether the result of the current call has already been computed and in Line 7 we store a new result in the cache. Without this dynamic programming the time complexity of the algorithm would be exponential in the number of variables used in the operands.

Algorithm 1 apply($F, G,$ <op>)

```
1: if $(F = 0 \vee F = 1) \wedge (G = 0 \vee G = 1)$ then return $F$ <op> $G$
2: if $(F, G,$ <op>$)$ in cache then return cache$[(F, G,$ <op>$)]$
3: $x \leftarrow$ min(var($F$), var($G$))
4: $L \leftarrow$ apply($F_{\overline{x}}, G_{\overline{x}},$ <op>)
5: $H \leftarrow$ apply($F_x, G_x,$ <op>)
6: $result \leftarrow$ BDD_node($x, L, H$)
7: cache$[(F, G,$ <op>$)] \leftarrow result$
8: return $result$
```

Another BDD operation that is frequently applied in symbolic algorithms is existential quantification: $\exists\vec{x} : F$. The existential quantifier abstracts (= removes) all nodes with a variable label in $\vec{x}$ from the BDD F. $\exists\vec{x} : F$ is the smallest function independent of $\vec{x}$ which still contains F [31]. After applying existential quantification over $\vec{x}$, all paths in F skip over the variables contained in $\vec{x}$. Thus, it does no longer matters which values are assigned to these variables. The BDD operation $\exists x : F$ which abstracts a single variable x from F is derived from the following equivalence

$$\exists x : F \Leftrightarrow F_{\overline{x}} \cup F_x$$

The variable x is not necessarily contained in the root of F. Therefore, the algorithm which computes the BDD representing $\exists x : F$ traverses F and finds all sub-BDDs F' which do contain x in their root. Each sub-BDD F' is replaced by a BDD representing the union of its children, $F'_{\overline{x}} \cup F'_x$. This abstracts all nodes containing x from F.

The BDD representing $\exists\vec{x} : F$ is computed by applying this node abstraction procedure to all nodes having a variable label in $\vec{x}$. Algorithm 2 illustrates this process. Line 1 check for the terminal case where F is a leaf. Line 3 selects the variable label x of the root of F. Lines 4 and 5 recursively search F for nodes with variable labels in $\vec{x}$. If $x \notin \vec{x}$, then Line 7 creates a new node for x, as the variable should not be abstracted. Else, x is abstracted from F in Line 9 by computing the union of the two recursive calls. The result is returned in Line 11.

Algorithm 2 $\exists \vec{x} : F$

```
1:  if $F = 0 \vee F = 1$ then return $F$
2:  if $(F, \vec{x}, \exists)$ in cache then return cache$[(F, \vec{x}, \exists)]$
3:  $x \leftarrow$ var$(F)$
4:  $L \leftarrow \exists \vec{x} : F_{\overline{x}}$
5:  $H \leftarrow \exists \vec{x} : F_x$
6:  if $x \notin \vec{x}$ then
7:      $result \leftarrow$ BDD_node$(x, L, H)$
8:  else
9:      $result \leftarrow$ apply$(L, H, \cup))$
10: cache$[(F, \vec{x}, \exists)] \leftarrow result$
11: return $result$
```

The existential quantifier is prominently used in two other important BDD operations: the computation of the image and preimage of a set S under a transition relation T. Suppose that S is a BDD representing a set of states encoded over variables $\vec{x} = x_1, \ldots, x_n$, and that T is a BDD representing a transition relation between states of which the source states are encoded over $\vec{x}$ and the target states are encoded over $\vec{x}' = x'_1, \ldots, x'_n$. The image of S under T is computed by the BDD operation

$$\exists \vec{x} : (S \cap T)[\vec{x}' := \vec{x}]$$

This operation consists of three steps as illustrated in Figure 2.8. First, the intersection of S and T is determined. This results in a BDD containing all transitions in T with a source in S. Next, the source states are removed from this BDD by existential quantification over $\vec{x}$. The result of this abstraction is a BDD containing the image of S under T. As a last step, the $\vec{x}'$ variables are renamed to $\vec{x}$ variables. In practice, the three steps just described can be combined into one coarse operation with the same time complexity as the existential quantification operation. The preimage of S under T is computed in a similar fashion using the operation $\exists \vec{x}' : (S[\vec{x} := \vec{x}'] \cap T)$. Example 6 illustrates the computation of the image operation for a small sliding tile puzzle.

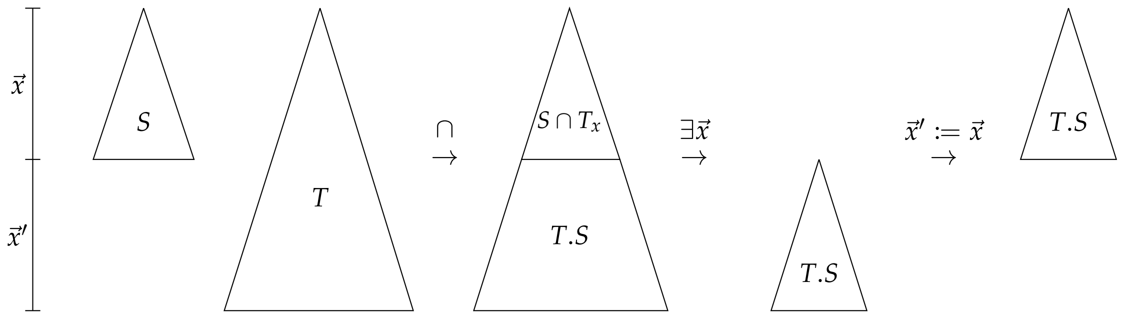
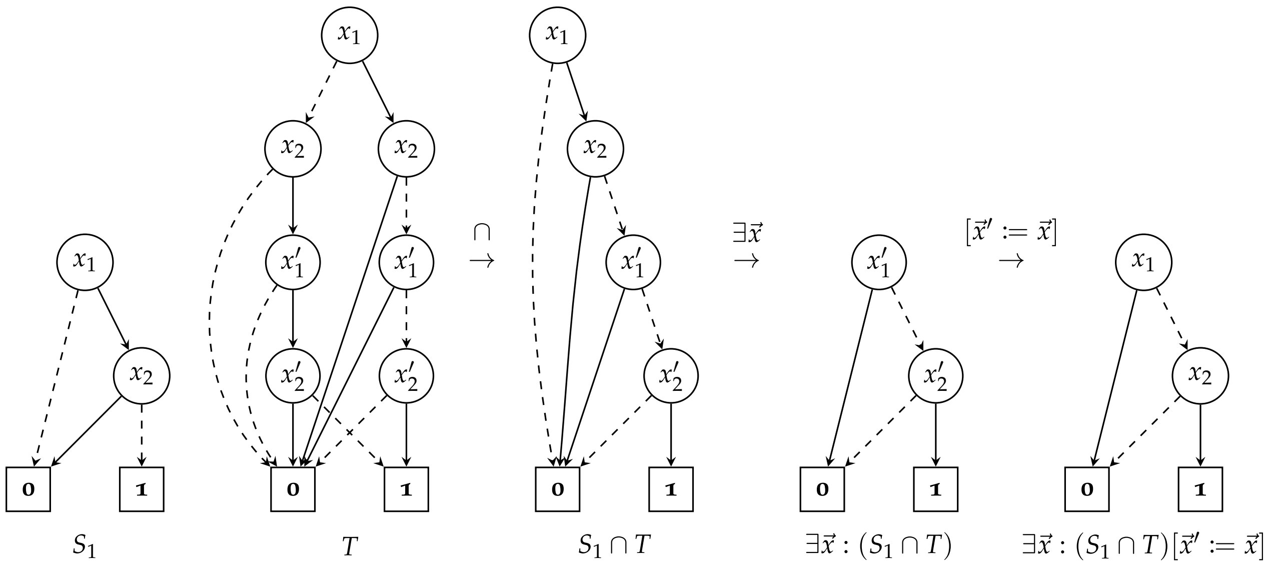


Figure 2.8: Illustration of the three steps performed in the computation of the image of S under T.

Example 6. *Consider a sliding tile puzzle which consists of just one tile and a 2 by 1 grid. This puzzle can be represented by an LTS with state space $S = \{s_1, s_2\}$ and (unlabelled) transition relation $\rightarrow = \{(s_1, s_2), (s_2, s_1)\}$. Thus, the puzzle can only go back and forth between two states. The states are encoded by a string of two variables, x_1x_2, which represent the two grid positions. Again, a variable is set to 1 if a tile is present in the corresponding grid position. We encode s_1 by 10 and s_2 by 01. The transitions are encoded by concatenating the source state with the target state.*
Let S_1 be a BDD which encodes $\{s_1\}$ over $\vec{x} = x_1, x_2$, and let T be a BDD which encodes $\rightarrow$ over $\vec{x}$ and $\vec{x}'$. Figure 2.9 shows the results of all three steps that must be performed in order to compute the image of $\{s_1\}$ under $\rightarrow$. The final BDD represents $\{s_2\}$.



x_1
x_2
x'_1
x'_2
0
1
S_1
T
$\cap \rightarrow$
$S_1 \cap T$
$\exists\vec{x} \rightarrow$
$\exists\vec{x} : (S_1 \cap T)$
$[\vec{x}' := \vec{x}] \rightarrow$
$\exists\vec{x} : (S_1 \cap T)[\vec{x}' := \vec{x}]$

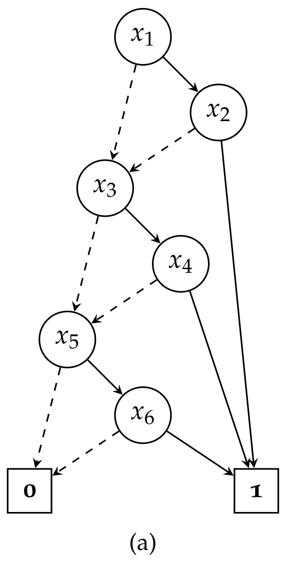
Figure 2.9: The five BDDs that are involved in the computation of the image of S_1 under T.

As stated before, the complexity of basic set operations is limited by the size of the result BDD. The complexity of the existential quantification operation is more difficult to define. It applies the union operation repeatedly at different depths in the BDD, depending on which set of variables is abstracted from the BDD. Operations which apply quantification, like the (pre)image operation, are also called symbolic steps. Symbolic steps are computationally much heavier than binary set operations, which is why the complexity of symbolic algorithms is expressed in the number of symbolic steps that they require.
The last operation which we will discuss in this section is the computation of a complement of a set represented by a BDD. A BDD containing n variables can represent a total of 2^n possible binary strings. We call this set the universe U of the BDD. The complement of a BDD F is defined as $\overline{F} := U \setminus F$. The BDDs F and $\overline{F}$ have the exact same structure, except for their 0- and 1-leafs which are interchanged. We can exploit this similarity by employing complement edges [7]. A complement edge indicates that the BDD to which it points should be negated. Thus F and $\overline{F}$ can be represented by the same BDD, only F is reached through a ordinary edge whereas $\overline{F}$ is reached through a complement edge. As a result, complement edges allow us to both save memory space and negate BDDs in constant time. In order to maintain an canonical form, complement edges are only used on low edges. For high edges, the complement edge can be pushed down the BDD using

Shannon decomposition: $\overline{F} \Leftrightarrow \overline{x} \cdot \overline{F}_{\overline{x}} + x \cdot \overline{F}_x$. Eventually, an high edge pointing to a leaf will be negated, which is done by simply changing its destination to the opposite leaf.

2.4.3 Variable Ordering

The run times of symbolic algorithms rely heavily upon the sizes of the BDDs which they manage during their execution. Thus, it is important that these BDDs are kept as small as possible. The size of a BDD is strongly dependent on the chosen variable ordering. Figure 2.10 illustrates an extreme case for $f(x_1, \ldots, x_6) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$. Figure 2.10a wields variable ordering $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$, while Figure 2.10b employs variable ordering $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$. In the general case of $f(x_1, \ldots, x_n) = x_1 \cdot x_2 + \ldots + x_{n-1} \cdot x_n$, the first variable ordering will result in a BDD containing $n + 2$ nodes, opposed to $2^{\frac{n}{2}+1}$ nodes for the second variable ordering.



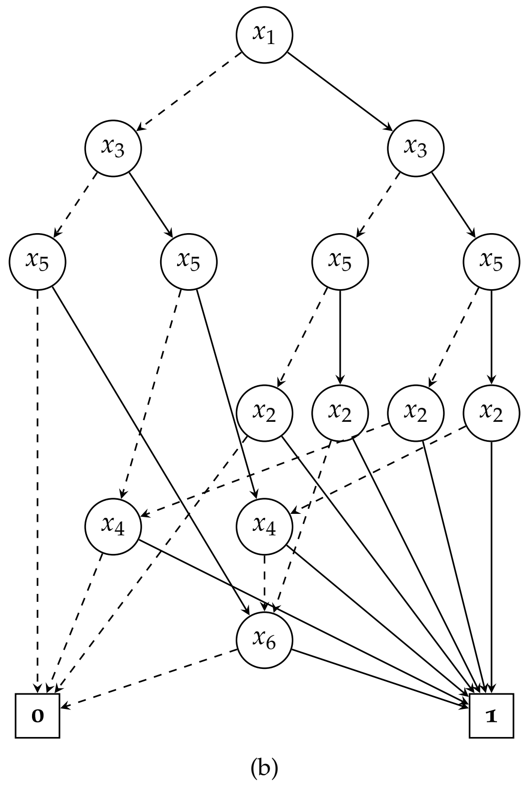


Figure 2.10: The BDD representing $f(x_1, x_2, x_3, x_4, x_5, x_6) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ for two different variable orderings.

A bad variable ordering can thus result in a BDD which grows exponentially in the number of variables. Fortunately, it is often possible to choose a good variable ordering based on prior knowledge about the problem domain. There are, however, some functions which are inherently complex. BDDs representing these kinds of functions will grow exponentially for every possible variable ordering. Generally, variables that are related in some way or are dependent on each other should be located in close proximity to each other in the variable ordering.

For transition and bisimulation relations over the states of a system model, this means an interleaved variable ordering is best [14]. Transitions in a system are often local and do not change many variables within the system. Thus, many variables in a transition keep the same value. Likewise, two states which are bisimilar, and thus have a similar behavior, often differ in just a few variables. So two variables x_i and x'_i which encode part of the source and target state in a transition or bisimulation relation likely have the same value. Because of this correlation, interleaved variable orderings, like $x_1 < x'_1 < x_2 < x'_2 < \ldots < x_n < x'_n$, will generally result in a small BDD.

Chapter 3

Symbolic Bisimulation

This chapter introduces a relation-based fixed point algorithm that computes the maximal bisimulation. It then derives the coarse BDD operations which are used to efficiently implement this algorithm. Finally, the algorithm is parallelized.

3.1 A relation-based symbolic bisimulation algorithm

Given an LTS $(S, A, \rightarrow)$, our algorithm aims to find the maximal bisimulation over $S \times S$. Towards building a least fixed point algorithm, as discussed at the end of Section 2.3, we give a recursive definition of the non-bisimilarity relation $\not\sim$ in Equation 3.1. In words, $\not\sim$ is the set of all pairs of states that are not bisimilar. The recursive definition of $\not\sim$ can be directly derived from Definition 2.

$$\begin{aligned} \not\sim := {} & \{(s,t) \mid \exists s' \exists a : (s \xrightarrow{a} s' \land \neg\exists t' : (t \xrightarrow{a} t' \land (s',t') \in \sim))\} \,\cup \\ & \{(s,t) \mid \exists t' \exists a : (t \xrightarrow{a} t' \land \neg\exists s' : (s \xrightarrow{a} s' \land (s',t') \in \sim))\} \end{aligned} \tag{3.1}$$

In order to construct $\not\sim$, our algorithm maintains a relation R which contains all pairs of non-bisimilar states that our algorithm has found thus far. R is initialized as the empty set. Pairs of non-bisimilar states are iteratively added to R until a fixed point is reached, at which point R equals $\not\sim$. For this purpose, the following functions are used

$$\begin{aligned} f_a(R) &:= \{(s,t) \mid \exists s' : (s \xrightarrow{a} s' \land \neg\exists t' : (t \xrightarrow{a} t' \land (s',t') \notin R))\} \,\cup \\ &\qquad \{(s,t) \mid \exists t' : (t \xrightarrow{a} t' \land \neg\exists s' : (s \xrightarrow{a} s' \land (s',t') \notin R))\} \\ f(R) &:= \bigcup_{a \in A} f_a(R) \end{aligned}$$

R is called a *fixed point* of f if $f(R) = R$. The function f is a monotonic (non-decreasing) function with respect to $\subseteq$, as $R_1 \subseteq R_2$ implies that $f(R_1) \subseteq f(R_2)$. Also, $(\mathcal{P}(S \times S), \subseteq)$ forms a complete lattice. Thus, according to the Knaster-Tarski lemma [20], there exists a (unique) least fixed point of f.

The notation $f^i(R)$ is used to denote that f is applied i many times, e.g. $f^3(R) = f(f(f(R)))$. If S contains n states then the least fixed point of f must be equal to $f^n(\emptyset)$, because f is strictly increasing until the fixed point is reached. The least fixed point contains exactly all pairs of non-bisimilar states. Thus, $\not\sim \; = \; f^n(\emptyset)$ and we find that the maximal bisimulation is equal to $S \times S \setminus f^n(\emptyset)$.

In order to compute f_a the following sets need to be constructed

$$Y_a := \{(s,t) \mid \exists s' : (s \xrightarrow{a} s' \wedge \neg\exists t' : (t \xrightarrow{a} t' \wedge (s',t') \notin R))\}$$

and

$$Z_a := \{(s,t) \mid \exists t' : (t \xrightarrow{a} t' \wedge \neg\exists s' : (s \xrightarrow{a} s' \wedge (s',t') \notin R))\}$$

such that $f_a(R) := Y_a \cup Z_a$. R is initially symmetric, as the empty set is a symmetric relation. If R is symmetric then $Z_a = Y_a^{-1}$, and thus the result of $f_a(R)$ is also symmetric. Therefore, R is symmetric during every step of the construction of $\not\sim$. Furthermore, this means that Z_a can be determined by taking the converse of Y_a. This is an advantage because computing the converse of a relation requires no symbolic steps (see Section 3.3), while computing Z_a takes two.

We found it hard to find a symbolic operation that computes Y_a directly due to the two quantifiers in its definition, which both quantify over the target states of the transition relation. Therefore, one of the quantifications is performed in an intermediate step. This intermediate step computes the following set

$$X_a := \{(s',t) \mid \neg\exists t' : (t \xrightarrow{a} t' \wedge (s',t') \notin R)\}$$

such that $Y_a := \{(s,t) \mid \exists s' : (s \xrightarrow{a} s' \wedge (s',t) \in X_a\}$.

Algorithm 3 presents our symbolic bisimulation algorithm *bisim*. As input, *bisim* takes a BDD S and a list of BDDs T. S represents the set of states contained in the given LTS. The list T forms a partition of the transition relation $\rightarrow$, where the transitions are split on their action label. That is, each BDD T_a in T contains all transitions having label $a \in A$ but no other transitions. This way, the actions labels need not be stored in the BDDs. This is beneficial, because the encoding of an action label a in a transition (s, a, t) has no correlation with, or is in no way depending on, the encoding of s or t. Thus, including action labels will cause the transition relation BDD to blow up in size, as is explained in Section 2.4. The relation R is represented in the algorithm by a BDD with the same name, which uses an interleaved variable ordering to store pairs of non-bisimilar states.

Line 1 initializes R to the empty set. The algorithm computes the least fixed point in the loop starting at Line

Algorithm 3 bisim(S, T)

```
1: $R \leftarrow \emptyset$
2: repeat
3:     $R' \leftarrow R$
4:     for all $T_a \in T$ do
5:         $X_a \leftarrow \{(s', t) \mid \neg\exists t' : ((t, t') \in T_a \wedge (s', t') \notin R)\}$
6:         $Y_a \leftarrow \{(s, t) \mid \exists s' : ((s, s') \in T_a \wedge (s', t) \in X_a)\}$
7:         $R \leftarrow Y_a \cup Y_a^{-1}$
8: until $R = R'$
9: return $S \times S \setminus R$
```

2. Line 3 stores a copy of R, Lines 4–7 extend R by computing f_a for every action a, and Line 8 checks if a fixed point has been reached. After a fixed point is found, the maximal bisimulation is returned in Line 9. While iterating over actions $a \in A$, there are two approaches for extending R: breadth first search (BFS) or chaining. Both of these approaches are illustrated in Figure 3.1. In BFS strategies, the result of $f_a(R)$ is computed for every $a \in A$ before the found pairs are added to R. Contrarily, in chaining the result of $f_a(R)$ is immediately added to R. Thus, R is extended after every call to f_a. This is the approach employed by Algorithm 3. After R is extended, subsequent calls to f_a might find pairs that would not have been found before the extension. Therefore, chaining will generally find more pairs per iteration than BFS strategies. This makes chaining the more efficient approach [30].

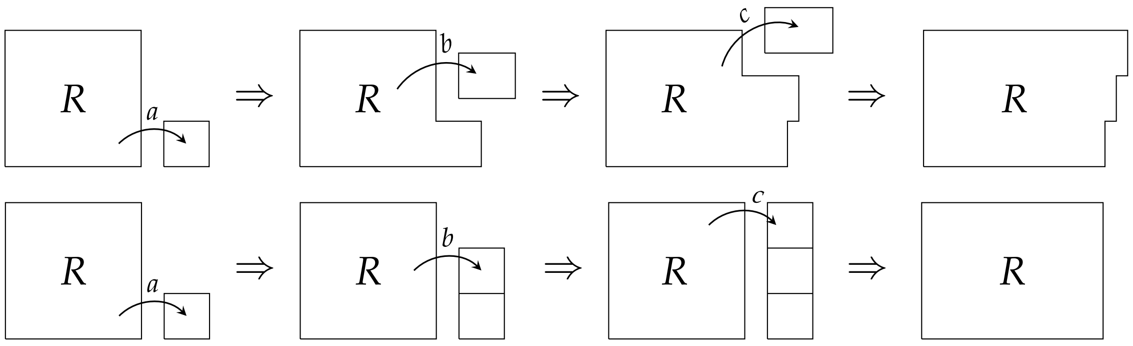


Figure 3.1: The extension of R using chaining (above), as used by Algorithm 3, vs BFS (below).

The next section will derive the symbolic operations that compute X_a and Y_a. Section 3.3 describes how the converse relation Y_a^{-1} is computed. Finally, the last section of this chapter introduces a variant of *bisim* that employs parallel BFS extension instead of sequential chaining.

3.2 Computing pairs of non-bisimilar states

Pairs of non-bisimilar states are determined in two steps using two symbolic operations, ∀preimage and relcomp, where $X_a = \forall\text{preimage}(R, T_a)$ and $Y_a = \text{relcomp}(T_a, \forall\text{preimage}(R, T_a))$. Thus, given a BDD R representing pairs of non-bisimilar states and a BDD T representing an unlabelled transition relation, the

$\forall$preimage needs to compute the following set

$$\{(s',t) \mid \neg\exists t' : ((t,t') \in T \wedge (s',t') \notin R)\}$$

The BDD operation which computes this set follows directly from the predicate which defines it. It takes as arguments the two BDDs R and T, which are both defined over the interleaved variables $\vec{x}$ and $\vec{x}'$. The result should be a BDD which is defined over the same variables. Note that the source states of T, which are defined over $\vec{x}$, form the target states of the result, which are defined over $\vec{x}'$. The source states T cannot be redefined over $\vec{x}'$, as these variables are already used to encode the target states of R and T. The source states of T must therefore be relabelled to a new set of variables $\vec{x}''$. As a final step, the operation relabels these states once more to $\vec{x}'$ in order to define the result over the desired variables. We find the following operation:

$$\forall\text{preimage}(R,T) := (\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}']$$

An algorithm which computes this operation can be found by rewriting the definition of $\forall$preimage to a recursive form. This is done by abstracting a single variable x from the formula using Shannon decomposition and subsequently rewriting the formula to a recursive form. Here there are three different cases, as x can be a part of either $\vec{x}$, $\vec{x}'$, or $\vec{x}''$.

1) $x \in \vec{x}$

$$\begin{aligned}
\forall\text{preimage}(R,T) &:= (\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}'] \\
&= \overline{x}((\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}'])_{\overline{x}} \vee x((\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}'])_{x} \\
&= \overline{x}((\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R_{\overline{x}}}))[\vec{x}'' := \vec{x}']) \vee x((\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R_{x}}))[\vec{x}'' := \vec{x}']) \\
&= \overline{x} \cdot \forall\text{preimage}(R_{\overline{x}}, T) \vee x \cdot \forall\text{preimage}(R_{x}, T)
\end{aligned}$$

2) $x' \in \vec{x}'$

$$\begin{aligned}
\forall\text{preimage}(R,T) &:= (\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}'] \\
&= (\neg\exists\vec{x}' : ((T[\vec{x} := \vec{x}''] \wedge \overline{R})_{\overline{x}'} \vee (T[\vec{x} := \vec{x}''] \wedge \overline{R})_{x'}))[\vec{x}'' := \vec{x}'] \\
&= (\neg\exists\vec{x}' : ((T_{\overline{x}'}[\vec{x} := \vec{x}''] \wedge \overline{R_{\overline{x}'}}) \vee (T_{x'}[\vec{x} := \vec{x}''] \wedge \overline{R_{x'}})))[\vec{x}'' := \vec{x}'] \\
&= (\neg(\exists\vec{x}' : (T_{\overline{x}'}[\vec{x} := \vec{x}''] \wedge \overline{R_{\overline{x}'}}) \vee \exists\vec{x}' : (T_{x'}[\vec{x} := \vec{x}''] \wedge \overline{R_{x'}})))[\vec{x}'' := \vec{x}'] \\
&= (\neg\exists\vec{x}' : (T_{\overline{x}'}[\vec{x} := \vec{x}''] \wedge \overline{R_{\overline{x}'}}) \wedge \neg\exists\vec{x}' : (T_{x'}[\vec{x} := \vec{x}''] \wedge \overline{R_{x'}}))[\vec{x}'' := \vec{x}'] \\
&= (\neg\exists\vec{x}' : (T_{\overline{x}'}[\vec{x} := \vec{x}''] \wedge \overline{R_{\overline{x}'}}))[\vec{x}'' := \vec{x}'] \wedge (\neg\exists\vec{x}' : (T_{x'}[\vec{x} := \vec{x}''] \wedge \overline{R_{x'}}))[\vec{x}'' := \vec{x}'] \\
&= \forall\text{preimage}(R_{\overline{x}'}, T_{\overline{x}'}) \wedge \forall\text{preimage}(R_{x'}, T_{x'})
\end{aligned}$$

3) $x'' \in \vec{x}''$

$$\begin{aligned}
\forall\text{preimage}(R,T) &:= (\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}'] \\
&= (\overline{x}''(\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))_{\overline{x}''} \vee x''(\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}''] \wedge \overline{R}))_{x''})[\vec{x}'' := \vec{x}'] \\
&= (\overline{x}''(\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}'']_{\overline{x}''} \wedge \overline{R})) \vee x''(\neg\exists\vec{x}' : (T[\vec{x} := \vec{x}'']_{x''} \wedge \overline{R})))[\vec{x}'' := \vec{x}'] \\
&= (\overline{x}''(\neg\exists\vec{x}' : (T_{\overline{x}}[\vec{x} := \vec{x}''] \wedge \overline{R})) \vee x''(\neg\exists\vec{x}' : (T_x[\vec{x} := \vec{x}''] \wedge \overline{R})))[\vec{x}'' := \vec{x}'] \\
&= (\overline{x}''(\neg\exists\vec{x}' : (T_{\overline{x}}[\vec{x} := \vec{x}''] \wedge \overline{R})))[\vec{x}'' := \vec{x}'] \vee (x''(\neg\exists\vec{x}' : (T_x[\vec{x} := \vec{x}''] \wedge \overline{R})))[\vec{x}'' := \vec{x}'] \\
&= \overline{x}'(\neg\exists\vec{x}' : (T_{\overline{x}}[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}'] \vee x'(\neg\exists\vec{x}' : (T_x[\vec{x} := \vec{x}''] \wedge \overline{R}))[\vec{x}'' := \vec{x}'] \\
&= \overline{x}' \cdot \forall\text{preimage}(R, T_{\overline{x}}) \vee x' \cdot \forall\text{preimage}(R, T_x)
\end{aligned}$$

The terminal cases of ∀preimage are found by assigning values 0 or 1 to R and T: $\forall\text{preimage}(0,1) = 0$ and $\forall\text{preimage}(1,T) = \forall\text{preimage}(R,0) = 1$. The algorithm which results from these derivations is shown in Algorithm 4.

Algorithm 4 $\forall\text{preimage}(R,T)$

```
1: if $R = 0 \wedge T = 1$ then return 0
2: if $R = 1 \vee T = 0$ then return 1
3: if $(R, T, \forall\text{preimage})$ in cache then return cache$[(R, T, \forall\text{preimage})]$
4: $x \leftarrow \min(\texttt{var}(R), \texttt{var}(T))$
5: if $x \in \vec{x} \wedge x = \texttt{var}(R)$ then
6:     do in parallel:
7:         $L \leftarrow \forall\text{preimage}(R_{\overline{x}}, T)$
8:         $H \leftarrow \forall\text{preimage}(R_x, T)$
9:     $result \leftarrow \texttt{BDD\_node}(x, L, H)$
10: else if $x \in \vec{x} \wedge x = \texttt{var}(T)$ then
11:     do in parallel:
12:         $L \leftarrow \forall\text{preimage}(R, T_{\overline{x}})$
13:         $H \leftarrow \forall\text{preimage}(R, T_x)$
14:     $result \leftarrow \texttt{BDD\_node}(\vec{x}'(x), L, H)$
15: else
16:     do in parallel:
17:         $L \leftarrow \forall\text{preimage}(R_{\overline{x}}, T_{\overline{x}})$
18:         $H \leftarrow \forall\text{preimage}(R_x, T_x)$
19:     $result \leftarrow \text{apply}(L, H, \cap)$
20: cache$[(R, T, \forall\text{preimage})] \leftarrow result$
21: return $result$
```

Lines 1 and 2 check for terminal cases. Line 4 selects the variable label x of the highest node in R and T. As stated before, x can be a part of either $\vec{x}$, $\vec{x}'$, or $\vec{x}''$. The use of $\vec{x}''$ variables can be avoided by distinguishing the source states of R and T with a simple check. This is done by changing the case $x \in \vec{x}$ to $x \in \vec{x} \wedge x = \texttt{var}(R)$ and changing the case $x \in \vec{x}''$ to $x \in \vec{x} \wedge x = \texttt{var}(T)$. Lines 5–19 create nodes in the result BDD according to the three recursive definitions of ∀preimage which followed from our derivations. The two recursive calls in each case are done in parallel. The expression $\vec{x}'(x)$ in Line 14 maps the variable x to its primed equivalent in $\vec{x}'$, e.g. $\vec{x}'(x_i) = x_i'$. Lines 3 and 20 add dynamic programming to the algorithm, reducing its complexity, as is typical for BDD operations (see Section 2.4). The result is returned in Line 21.

Observe that the algorithm is coarse-grained. The existential quantification, conjunction and renaming operations, which need to be performed in order to compute the ∀preimage, are all combined in one operation. This avoids pollution of the unique table and operation cache by skipping intermediary results, making the operation more efficient.

The relcomp (relational composition) operation is given as inputs a BDD T representing an unlabelled transition relation and a BDD X representing the result of the ∀preimage operation. It computes the composition of the two relations.

$$\{(s,t) \mid \exists s' : ((s,s') \in T \wedge (s',t) \in X)\}$$

The BDD operation which constructs this composition follows directly from the predicate which defines it. It takes as arguments the BDDs T and X, which are both defined over the interleaved variables $\vec{x}$ and $\vec{x}'$. The result BDD should also be defined over $\vec{x}$ and $\vec{x}'$. The operation matches the target states of T with the source states of X. Thus, these should be defined over the same variables. Both the target states of T and the source states of X are therefore relabelled to $\vec{x}''$. We find the following operation:

$$\text{relcomp}(T,X) := \exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}''])$$

We again derive an algorithm for the operation by abstracting a single variable from this formula. There are three cases.

1) $x \in \vec{x}$

$$\begin{aligned}\text{relcomp}(T,X) &:= \exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}''])\\ &= \overline{x}(\exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}'']))_{\overline{x}} \vee x(\exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}'']))_{x}\\ &= \overline{x}(\exists \vec{x}'' : (T_{\overline{x}}[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}''])) \vee x(\exists \vec{x}'' : (T_{x}[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}'']))\\ &= \overline{x} \cdot \text{relcomp}(T_{\overline{x}}, X) \vee x \cdot \text{relcomp}(T_x, X))\end{aligned}$$

2) $x' \in \vec{x}'$

$$\begin{aligned}\text{relcomp}(T,X) &:= \exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}''])\\ &= \overline{x}'(\exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}'']))_{\overline{x}'} \vee x'(\exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}'']))_{x'}\\ &= \overline{x}'(\exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X_{\overline{x}'}[\vec{x} := \vec{x}''])) \vee x'(\exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X_{x'}[\vec{x} := \vec{x}'']))\\ &= \overline{x}' \cdot \text{relcomp}(T, X_{\overline{x}'}) \vee x' \cdot \text{relcomp}(T, X_{x'}))\end{aligned}$$

3) $x'' \in \vec{x}''$

$$\begin{aligned}
\text{relcomp}(T, X) &:= \exists \vec{x}'' : (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}'']) \\
&= \exists \vec{x}'' : ((T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}''])_{\overline{x}''} \vee (T[\vec{x}' := \vec{x}''] \wedge X[\vec{x} := \vec{x}''])_{x''}) \\
&= \exists \vec{x}'' : ((T[\vec{x}' := \vec{x}'']_{\overline{x}''} \wedge X[\vec{x} := \vec{x}'']_{\overline{x}''}) \vee (T[\vec{x}' := \vec{x}'']_{x''} \wedge X[\vec{x} := \vec{x}'']_{x''})) \\
&= \exists \vec{x}'' : ((T_{\overline{x}'}[\vec{x}' := \vec{x}''] \wedge X_{\overline{x}}[\vec{x} := \vec{x}'']) \vee (T_{x'}[\vec{x}' := \vec{x}''] \wedge X_{x}[\vec{x} := \vec{x}''])) \\
&= \exists \vec{x}'' : (T_{\overline{x}'}[\vec{x}' := \vec{x}''] \wedge X_{\overline{x}}[\vec{x} := \vec{x}'']) \vee \exists \vec{x}'' : (T_{x'}[\vec{x}' := \vec{x}''] \wedge X_{x}[\vec{x} := \vec{x}'']) \\
&= \text{relcomp}(T_{\overline{x}'}, X_{\overline{x}}) \vee \text{relcomp}(T_{x'}, X_{x})
\end{aligned}$$

The terminal cases of relcomp are: $\text{relcomp}(T, 0) = \text{relcomp}(0, X) = 0$ and $\text{relcomp}(1, 1) = 1$. The algorithm which performs the relcomp operation using these derivations is shown in Algorithm 5. The use of the variables $\vec{x}''$ can be avoided by changing the case $x \in \vec{x}$ to $x \in \vec{x} \wedge x = \texttt{var}(T)$ and changing the case $x \in \vec{x}'$ to $x \in \vec{x}' \wedge x = \texttt{var}(X)$.

Both the ∀preimage and relcomp operation require one symbolic step, as they contain one existential quantifier. Apart from the memoization operations, both operations are entirely derived from their mathematical definition. Therefore, if these derivations are correct, then so are the algorithms.

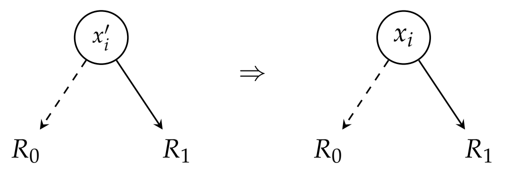
Algorithm 5 relcomp(T, X)

```
1:  if $T = 0 \vee X = 0$ then return $0$
2:  if $T = 1 \wedge X = 1$ then return $1$
3:  if $(T, X, \text{relcomp})$ in cache then return cache$[(T, X, \text{relcomp})]$
4:  $x \leftarrow \min(\texttt{var}(T), \texttt{var}(X))$
5:  if $x \in \vec{x} \wedge x = \texttt{var}(T)$ then
6:      do in parallel:
7:          $L \leftarrow \text{relcomp}(T_{\overline{x}}, X)$
8:          $H \leftarrow \text{relcomp}(T_{x}, X)$
9:      $result \leftarrow \texttt{BDD\_node}(x, L, H)$
10: else if $x \in \vec{x}' \wedge x = \texttt{var}(X)$ then
11:     do in parallel:
12:         $L \leftarrow \text{relcomp}(T, X_{\overline{x}})$
13:         $H \leftarrow \text{relcomp}(T, X_{x})$
14:     $result \leftarrow \texttt{BDD\_node}(x, L, H)$
15: else
16:     $x \leftarrow \vec{x}(x)$, $x' \leftarrow \vec{x}'(x)$
17:     do in parallel:
18:         $L \leftarrow \text{relcomp}(T_{\overline{x}'}, X_{\overline{x}})$
19:         $H \leftarrow \text{relcomp}(T_{x'}, X_{x})$
20:     $result \leftarrow \text{apply}(L, H, \cup)$
21: cache$[(T, X, \text{relcomp})] \leftarrow result$
22: return $result$
```

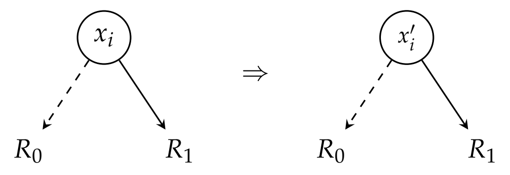
3.3 Computing the converse of a relation

Theoretically, computing the converse of a relation represented by a BDD R is a simple task. Let the source states of R be defined over $\vec{x}$ and the target states over $\vec{x}'$. If R uses an interleaved variable ordering, then R can be converted to the converse relation R^{-1} by relabelling its source states to $\vec{x}'$, relabelling its target states to $\vec{x}$, and adopting the new variable ordering $x'_1 < x_1 < x'_2 < x_2 < \ldots < x'_n < x_n$. The structure of the BDD does not change, thus R^{-1} contains exactly the same paths as R. However, a path encoding a pair (s, t) in R encodes the pair (t, s) in R^{-1}. The time required for this operation is linear in the size of R, as every node in R needs to be visited only once to relabel its variable. The downside of this approach is that it changes the order of the variables. Most BDD operations require that their operands employ the same variable ordering. It is therefore desirable that the same variable ordering is used in all BDDs managed by an algorithm. Otherwise, the algorithm would have to spend a lot of time relabelling. Fortunately, the converse of a relation can also be computed without changing the variable ordering. The algorithm which achieves this repeatedly swaps two levels in the BDD. This changes the structure of the BDD, but preserves its variable ordering. This swapping technique originates from dynamic variable reordering algorithms [27].

The conversion algorithm takes as input a single BDD R, which is again defined over $\vec{x}$ and $\vec{x}'$ and uses an interleaved variable ordering. The algorithm recursively traverses R and in each call considers the first two variables x_i and x'_i in the variable ordering of the current (sub-)BDD. As an initial step, the algorithm recursively computes the converse of the sub-BDDs directly below the two levels containing x_i and x'_i. Then, it swaps the levels of x_i and x'_i without changing the relation represented by the BDD. Finally, it relabels the nodes containing x_i to x'_i and the nodes containing x'_i to x_i. This both restores the variable ordering of the BDD and converts R to R^{-1}.



(a) $\texttt{var}(R) = x'_i$

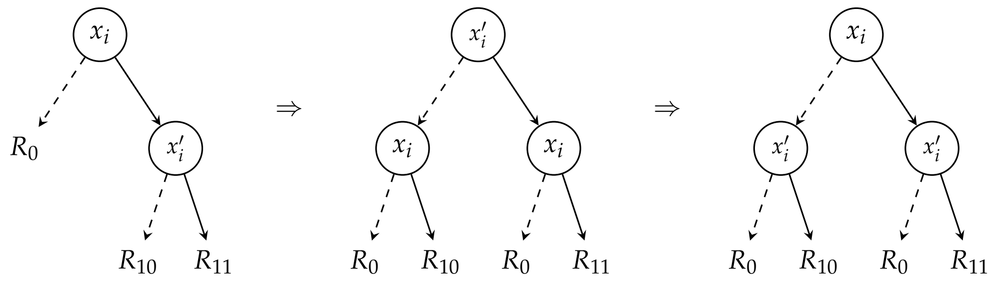


(b) $\texttt{var}(R) = x_i \wedge \texttt{var}(R_0) \neq x'_i \wedge \texttt{var}(R_1) \neq x'_i$

Figure 3.2: Conversion on structures a and b.

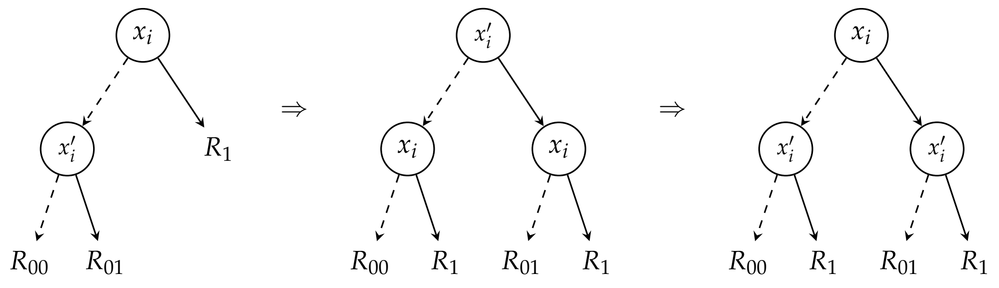
The two levels containing x_i and x'_i can take on five different structures. The first two of these structures are shown in Figure 3.2. Here R has either no nodes containing x_i (structure a) or no nodes containing x'_i (structure b). In both of these cases the two levels do not have to be swapped, as one of the levels is not present in the BDD. Therefore, the algorithm only relabels the root node of the BDD to its (un)primed counterpart.

The next three structures are depicted by Figure 3.3, Figure 3.4, and Figure 3.5. In these structures x_i is the variable label of the root of R and x'_i is the variable label of the low child R_0, or of the high child R_1, or of both.



(c) $\texttt{var}(R) = x_i \wedge \texttt{var}(R_0) \neq x'_i \wedge \texttt{var}(R_1) = x'_i$

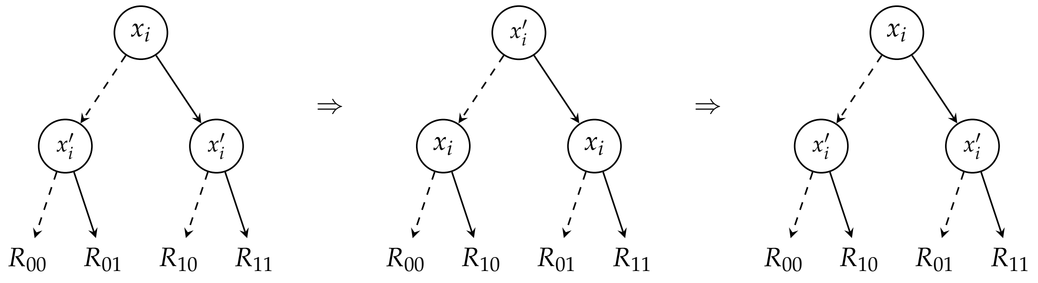
Figure 3.3: Conversion on structure c.



(d) $\texttt{var}(R) = x_i \wedge \texttt{var}(R_0) = x'_i \wedge \texttt{var}(R_1) \neq x'_i$

Figure 3.4: Conversion on structure d.

Here the two levels do need to be swapped. The swapping of the levels is illustrated for each structure in its corresponding figure. After swapping, the (grand)children of R are still reached through the same outgoing edges of x_i and x'_i. For example, in Figure 3.4 R_{01} is reached by taking the dashed edge in the x_i node and the solid edge in the x'_i node. This holds both before and after the levels are swapped. Only the order in which the edges are followed has changed. The final step, which relabels the nodes, can in practice be performed while the levels are being swapped.



(e) $\texttt{var}(R) = x_i \wedge \texttt{var}(R_0) = x'_i \wedge \texttt{var}(R_1) = x'_i$

Figure 3.5: Conversion on structure e.

Algorithm 6 depicts the conversion algorithm. Line 1 checks for the terminal case where R is a leaf. Lines 2 and 35 are memoization operations which again reduce the complexity of the algorithm. Lines 3–34 perform the recursive calls and level swaps for all five structures. The recursive calls are performed in parallel. For each case a BDD node is created which represents the rightmost BDD in the corresponding figure above. The result is returned in Line 36. The algorithm visits every node in R exactly once. The time complexity of the algorithm is therefore linear in the size of R. This means that the algorithm requires no symbolic steps.

Algorithm 6 converse(R)

```
1: if $R = 0 \vee R = 1$ then return $R$
2: if $(R, \texttt{converse})$ in cache then return $\texttt{cache}[(R, \texttt{converse})]$
3: $x \leftarrow \texttt{var}(R)$
4: $R_0 \leftarrow \texttt{low}(R)$, $R_1 \leftarrow \texttt{high}(R)$
5: if $x \in \vec{x}'$ then                                                   ▷ Structure a
6:     do in parallel:
7:         $low \leftarrow$ converse$(R_0)$
8:         $high \leftarrow$ converse$(R_1)$
9:     $result \leftarrow$ BDD_node$(\vec{x}(x), low, high)$
10: else if var$(R_0) \neq \vec{x}'(x) \wedge$ var$(R_1) \neq \vec{x}'(x)$ then     ▷ Structure b
11:     do in parallel:
12:         $low \leftarrow$ converse$(R_0)$
13:         $high \leftarrow$ converse$(R_1)$
14:     $result \leftarrow$ BDD_node$(\vec{x}'(x), low, high)$
15: else
16:     if var$(R_0) \neq \vec{x}'(x)$ then                                      ▷ Structure c
17:         do in parallel:
18:             $R_{00} \leftarrow R_{01} \leftarrow$ converse$(R_0)$
19:             $R_{10} \leftarrow$ converse(low$(R_1)$)
20:             $R_{11} \leftarrow$ converse(high$(R_1)$)
21:     else if var$(R_1) \neq \vec{x}'(x)$ then                                 ▷ Structure d
22:         do in parallel:
23:             $R_{00} \leftarrow$ converse(low$(R_0)$)
24:             $R_{01} \leftarrow$ converse(high$(R_0)$)
25:             $R_{10} \leftarrow R_{11} \leftarrow$ converse$(R_1)$
26:     else                                                                ▷ Structure e
27:         do in parallel:
28:             $R_{00} \leftarrow$ converse(low$(R_0)$)
29:             $R_{01} \leftarrow$ converse(high$(R_0)$)
30:             $R_{10} \leftarrow$ converse(low$(R_1)$)
31:             $R_{11} \leftarrow$ converse(high$(R_1)$
32:     $low \leftarrow$ BDD_node$(\vec{x}'(x), R_{00}, R_{10})$
33:     $high \leftarrow$ BDD_node$(\vec{x}'(x), R_{01}, R_{11})$
34:     $result \leftarrow$ BDD_node$(x, low, high)$
35: cache$[(R,$ converse$)] \leftarrow result$
36: return $result$
```

3.4 Parallel breadth first search extension

All three symbolic operations which were described in the previous two sections compute their recursive calls in parallel. Thus, the *bisim* algorithm already contains a lot of concurrency. The algorithm can be further parallelized by concurrently performing the computation of f_a for each action a, instead of looping over all actions. This approach employs a BFS extension of R, rather than an extension by chaining (which is inherently sequential). Algorithm 7 displays the version of *bisim* which implements this approach.

Algorithm 7 bisim2(S, T)

```
1: $R \leftarrow \emptyset$
2: repeat
3:     $R' \leftarrow R$
4:     $R \leftarrow \text{parLoop}(R, T)$
5: until $R \neq R'$
6: return $S \times S \setminus R$
```

The parallelization of the loop which computes f_a is handled by the function parLoop, shown in Algorithm 8. This function takes as inputs a BDD R which represents non-bisimilar state pairs and a list T of BDDs representing transition relations which are separated on action labels. Instead of considering each transition relation in T sequentially, the function decomposes T into two halves. It then recursively calls itself on both halves, and computes the results in parallel. This takes place in Lines 6–8. The union of the two results is returned in Line 9. When T only contains one transition relation, the recursive procedure is stopped. In Lines 2–5 the result of $f_a(R)$ is returned for the action corresponding to the transition relation contained in T.

Algorithm 8 parLoop(R, T)

```
1: $n \leftarrow |T|$
2: if $n = 1$ then
3:     $X \leftarrow \{(s', t) \mid \neg\exists t' : ((t, t') \in T[1] \wedge (s', t') \notin R)\}$
4:     $Y \leftarrow \{(s, t) \mid \exists s' : ((s, s') \in T[1] \wedge (s', t) \in X)\}$
5:     return $Y \cup Y^{-1}$
6: do in parallel:
7:     $L \leftarrow \text{parLoop}(R, T[1 : \lfloor \frac{n}{2} \rfloor])$
8:     $H \leftarrow \text{parLoop}(R, T[\lfloor \frac{n}{2} \rfloor + 1 : n])$
9: return $L \cup H$
```

The recursive decomposition applied in parLoop corresponds to a task dependency graph. In a task dependency graph each node represents a task and each edge expresses a dependency between two different tasks. A dependency indicates that the result of a task (source of edge) is required for the execution of another task (target of edge). Thus, a directed path in the graph depicts a sequence of tasks which must be performed consecutively. By summing up the execution times of all tasks which lie on the longest path in the graph, the minimal time which is needed to perform all tasks in the graph in parallel can be found. In our algorithm the tasks consist of computing the union of the two recursive calls and computing $f_a(R)$. An example of a task dependency graph for parLoop is shown in Figure 3.6. The only symbolic steps required in parLoop are found at the bottom level of the graph for the computation of $f_a(R)$. The computation of parLoop should in theory thus only need two symbolic steps, given that enough computing resources are available. A chaining algorithm would require $2 \cdot n$ symbolic steps, with n the number of actions contained in the given LTS.

It is not immediately clear if the *bisim*2 algorithm is more efficient than the standard *bisim* algorithm. The parallelization of the extension of R should cause *bisim*2 to have a better scalability than *bisim*, but chaining

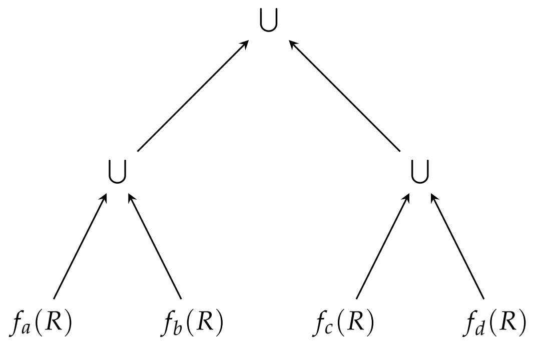


Figure 3.6: A task dependency graph of the parLoop algorithm for an input LTS containing actions $A = \{a, b, c, d\}$.

is the more efficient extension approach.

Chapter 4

Related Work

There exist two main approaches to bisimulation minimization. The first approach refines an equivalence relation by iteratively removing pairs of non-bisimilar states from the relation. This relation-based approach is used in the *bisim* algorithm. Boauli and de Simone [6] were the first to implement a symbolic relation-based bisimulation algorithm. Their results showed small improvements over the existing explicit methods. Another symbolic relation-based algorithm was introduced by Mumme and Ciardo [24]. Their algorithm functions very much like our *bisim* algorithm, but the two algorithms were found independently. Mumme and Ciardo's algorithm differs from *bisim* in four ways. Firstly, the algorithm is sequential instead of parallel. Secondly, the algorithm does not apply the converse operation utilized in *bisim*. As a consequence, the computation of non-bisimilar pairs of states requires two extra symbolic steps. Thirdly, the algorithm does not use coarse-grained operations, such as ∀preimage and relcomp. Fourthly, the algorithm operates on quasi-reduced MDDs instead of BDDs and employs a heuristic called saturation [29]. MDDs expand on BDDs by allowing internal nodes to have more than two outgoing edges. Saturation is a heuristic which makes MDD operations more efficient. Slow run times of symbolic algorithms are often caused by decision diagrams which grow too large. This explosion in size generally takes place in the lower levels of the decision diagram. The saturation heuristic tries to combat this problem by saturating the lower levels of a decision diagram as much as possible.

Saturation accomplishes this by performing operations locally on sub-BDDs, instead of operating on the entire BDD. For instance, let R be a BDD representing a relation which is encoded over variables $\vec{x}$ and $\vec{x}'$. Thus, a pair of states (s, t) is encoded as the string $x_1x_1'x_2x_2' \dots x_nx_n'$. Let T be a BDD representing a transition relation which is also encoded over $\vec{x}$ and $\vec{x}'$ and has a root node containing variable label x_i. Any statement of the form

$$R := R \texttt{<op>} T$$

does not affect the nodes in R containing variables $x_1, x'_1, \ldots, x_{i-1}, x'_{i-1}$. Or in other words, the operation <op> functions on these nodes like an identity function. Therefore, we can derive the result of $R := R$ <op> T by traversing R and applying the operation to every sub-BDD which is directly below the level of nodes containing x'_{i-1}. The saturation heuristic utilizes this approach to apply operations to the lowest nodes in a decision diagram first. Mumme and Ciardo's algorithm takes as inputs the MDD B, representing the maximal bisimulation, and a list of MDDs T, containing MDDs T_a representing all transitions with label a. This list is sorted on the height of the root nodes. The algorithm traverses to the bottom of B and finds new non-bisimilar pairs of states using the transition relation with the lowest root first. Next, it searches for new non-bisimilar pairs of states using the transition relation with the second lowest root, etc. Every time a new pair is found, the algorithm traverses back to the bottom of B and repeats the process, starting again with the transition relation with the lowest root. This maximally saturates the lower levels of the MDD, which keeps it small.
The saturation heuristic has been shown to be very successful in MDD algorithms. To our knowledge, saturation has not yet been applied to BDD algorithms. We are very interested in seeing what benefits saturation could bring to BDD algorithms like *bisim*.

The second approach to bisimulation minimization uses signatures to iteratively refine a partition which devides the state space into equivalence classes that contain states which are pairwise bisimilar. This partition-based approach is used in explicit algorithms by Paige and Tarjan [25] and by Groote et al. [19], and in symbolic algorithms by Wimmer et al. [37] and Van Dijk [35] among others. The algorithm presented by Van Dijk is, as far as we are aware, the only parallel symbolic bisimulation algorithm. The algorithm maintains a partition π which initially contains just one equivalence class that comprises the entire state space. The partition is iteratively refined based on signatures, which characterise each state, until a fixed point is reached. Given an LTS $(S, A, \rightarrow)$ and a partition π, the signature σ of a state s is defined as follows,

$$\sigma(\pi)(s) := \{(a, C) \mid \exists s' \in C : s \xrightarrow{a} s'\}$$

where C is an equivalence class contained in π. Partitions are refined using a function named sigref, which groups states with the same signature into the same equivalence class. Let π^i denote the partition found in iteration i. The initial partition is refined until a fixed point $\pi^{n+1} = \pi^n$ is reached, using the following definitions

$$\begin{aligned}
\text{sigref}(\pi, \sigma) &:= \{\{t \in S \mid \sigma(\pi)(s) = \sigma(\pi)(t)\} \mid s \in S\} \\
\pi^0 &:= \{S\} \\
\pi^{n+1} &:= \text{sigref}(\pi^n, \sigma)
\end{aligned}$$

After every partition refinement step, a new signature is assigned to each state. Thus, each iteration of the algorithm consists of two steps, a partition refinement step and a signature computation step.

The algorithm uses BDDs to symbolically represent all data. The partition π is represented by a BDD P, which encodes a state s followed by a block number (= equivalence class label) b. The signatures σ are represented by a BDD σ, which encodes a state s followed by a action a and a block number b. Lastly, the transition relation $\rightarrow$ is represented by a BDD T, which encodes a source state s and a target state s' in an interleaved fashion followed by an action a.

The symbolic partition refinement operation efficiently reuses block numbers and requires no symbolic steps. However, the operation is too intricate to further discuss here. New signatures are computed by the following operation

$$\exists s' : (T \wedge P[s := s'])$$

which does comprise a single symbolic step. The signature computation operation is computationally the heaviest step in the algorithm and occupies 70 to 99 percent of the sequential execution time.

It is stated by Van Dijk and by Fisler and Vardi [16] that partition-based BDDs form a smaller representation of the maximal bisimulation than relation-based BDDs. This statement is based on the number of variables which are used in partition- and relation-based BDDs. If an LTS contains n states, then a relation-based R will require at least $2 \cdot \lceil \lg n \rceil$ variables to encode all pairs of states. The number of variables used in partition-based BDD P depends on the number of equivalence classes contained in the partition. If just one equivalence class is present, then P does not need any variables to label the classes. Thus, a minimum of $\lceil \lg n \rceil$ variables are needed to encode the partition. However, in the worst case the partition contains n equivalence classes. P now needs at least $\lceil \lg n \rceil$ variables to label all equivalence classes. Thus, P also requires a total of $2 \cdot \lceil \lg n \rceil$ variables to represent the bisimulation. The number of nodes in a BDD can grow exponentially in the number of used variables. Therefore, the conclusion is drawn that partition-based BDDs are generally smaller.

The maximal bisimulation of an LTS can also be computed by fixed point computation on dependency graphs, as is demonstrated in [12]. A dependency graph $G = (V, E)$ consists of a set of nodes V and a set of hyperedges $E \subseteq V \times \mathcal{P}(V)$. An assignment on a dependency graph is a function $f_A : V \rightarrow \{0, 1\}$. A fixed point assignment is an assignment f_A for which every hyperedge (v, V') satisfies the following property: if $f_A(v') = 1$ for all $v' \in V'$, then $f_A(v) = 1$. The least fixed point assignment is a fixed point assignment $f_{A_{min}}$ where $\sum_{v \in V} f_{A_{min}}(v)$ is minimal. Figure 4.1 shows an example of a least fixed point assignment on a dependency graph. A number of problems, such as the problem of Horn formula satisfiability, can be reduced to fixed point computation on dependency graphs [22].

Given an LTS $(S, A, \rightarrow)$, the dependency graph $G = (V, E)$ associated with the LTS is defined as follows

$$V := S \times S$$

$$E := \{((s,t), \{(s',t') \mid t \xrightarrow{a} t'\}) \mid s \xrightarrow{a} s'\} \cup \{((s,t), \{(s',t') \mid s \xrightarrow{a} s'\}) \mid t \xrightarrow{a} t'\}$$

Two states $s, t \in S$ are bisimilar if and only if $f_{A_{min}}((s,t)) = 0$ in the dependency graph associated with the given LTS. Therefore, bisimulation minimization can be reduced to fixed point computation on dependency graphs (and vice versa). As a consequence, the *bisim* algorithm is also able to find the least fixed point of a dependency graph. However, *bisim* can only be applied to dependency graphs that are derived from an LTS. An efficient symbolic algorithm for general dependency graphs does not yet exist. We expect that such an algorithm would make a significant contribution.

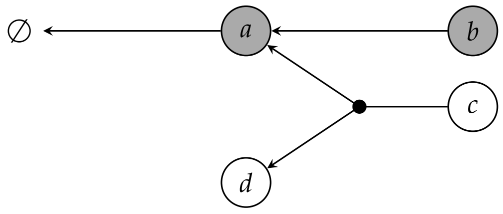


Figure 4.1: A least fixed point assignment on a simple dependency graph. The colored nodes are mapped to 1 by the assignment.

The benefits of bisimulation minimization in the context of state space reduction prior to or during the verification of invariance properties are examined by Fisler and Vardi in [16]. In their work, three algorithms are presented which verify the invariance properties of a model while computing its maximal bisimulation. These partially symbolic algorithms were produced by incorporating on-the-fly model checking in three existing bisimulation algorithms. The algorithms mark all states that are visited during the bisimulation minimization process. When a critical state is reached, the algorithm terminates. Thus, if the model can not be verified, the bisimulation minimization process is stopped prematurely.

Fisler and Vardi found in previous work [15] that the cost of bisimulation minimization often exceeds the cost of model checking by a considerable margin. They hoped that their new approach would be more efficient than model checking without the use of bisimulation minimization. Unfortunately, the results show that using bisimulation minimization as either a part of, or as a preprocessor to, model checking invariant properties is not profitable. However, this does not mean that bisimulation minimization can not be beneficial in other verification contexts.

Chapter 5

Experimental Evaluation

5.1 Implementation

The *bisim* and *bisim*2 algorithms (Algorithm 3 and 7), and the ∀preimage and relcomp BDD operations were implemented in C using Sylvan [33], a parallel BDD package. Sylvan offers (internally) parallel BDD operations, by splitting the recursive calls into fine-grained tasks, which are scheduled by Lace [36]. We used the same approach to implement the recursive calls listed in our new coarse BDD operations, ∀preimage and relcomp, as parallel calls. Thus, all BDD operations performed in our algorithms are fully parallel.

The *bisim*2 algorithm also features high-level parallelism by creating a task dependency graph of BDD operations themselves. We implemented these parallel calls as separate tasks in Lace, which ensures that the fine-grained internal tasks and the high-level (external) calls are all executed in parallel, while maintaining the partial order stipulated by the dependency graphs.

We did not succeed with implementing the bisim and bisim2 without the coarse-grained BDD operations from Algorithms 4 and 5. Therefore, we could not compare the individual gains achieved by the development of these coarse operations. Since [34] showed an up to 40% percent performance and space savings improvement for an integrated image BDD operation (RelProdS), we can reasonably expect at least a similar improvement from our coarse BDD operations. In fact, the benefit likely exceeds that amount, because the bisimulation computation in small steps requires the introduction of doubly primed variables (and RelProdS does not).

All algorithms and BDD operations were implemented within the SigrefMC package [35]. This extended version of SigrefMC can be found online at [2].

5.2 Experiments

We repeated the experiment ran by Van Dijk in [35] to study the difference in performance between *bisim*, *bisim*2, and Van Dijk's state-of-the-art algorithm. In the experiment, the maximal bisimulation is computed for six LTS models which portray Kanban production systems [37]. The smallest model contains 256 states and 904 transitions, while the largest model contains 264,515,056 states and 1,689,124,864 transitions.
The experiment was performed on a machine that contains two Intel Xeon E5-2630v3 CPUs, which each have eight physical cores. To evaluate the parallel speedup of each algorithm, the experiment was run once using only one core and once using all sixteen cores. We do not use any hyperthreading. At the start of the experiment, 2.625 GB of memory space is allocated for the two hash tables which contain the unique table and the operation cache. The maximum size of each hash table was limited to 84 GB.

5.3 Results

The results of the experiment are presented in Table 5.1. The run times shown in the table are the average of at least 16 runs. Entries containing a dashed line (-) went over the time limit, which was set to 2400 seconds. The number behind the algorithm name in the column header indicates the number of cores that were used in the experiment. The speedup of each algorithm is determined per model by dividing the sequential run time by the parallel run time.

Table 5.1: Results of the experiments. The shown run times are the average of at least 16 runs. The timeout is set to 2400 seconds.

	Time (s)						Speedups		
Model	Van Dijk 1	Van Dijk 16	*bisim* 1	*bisim* 16	*bisim*2 1	*bisim*2 16	Van Dijk	*bisim*	*bisim*2
kanban01	0.07	0.09	0.02	0.08	0.03	0.11	0.78	0.25	0.27
kanban02	0.17	0.61	0.72	1.93	1.42	1.71	0.28	0.37	0.83
kanban03	3.66	3.18	7.14	5.83	18.67	7.18	1.15	1.22	2.60
kanban04	53.84	24.60	42.81	18.38	194.17	39.45	2.19	2.33	4.92
kanban05	549.47	201.82	216.82	55.44	1223.02	157.65	2.72	3.91	7.76
kanban06	-	1758.71	724.84	147.12	-	576.85	-	4.93	-

For kanban04, -05, and -06 (the three largest models) *bisim* provides the best results. The parallel execution of *bisim* on the kanban06 model is almost 4 times faster than the parallel execution of *bisim*2 and almost 12 times faster than the parallel execution of Van Dijk's algorithm. Also, *bisim* is the only algorithm that can compute the maximal bisimulation of the kanban06 model within the time limit using only one core. For the smaller kanban02 and -03 models, Van Dijk's algorithm does outperform *bisim* and *bisim*2. The speedups of both *bisim* and *bisim*2 are higher than the speedup of Van Dijk's algorithm, indicating a better scalability. As expected, *bisim*2 does have a better speedup than *bisim*.

The efficiency of a symbolic algorithm directly depends on the size of the BDDs handled by the algorithm. The graphs in Figure 5.1 depicts the number of nodes in the non-bisimilarity relation BDD R during the execution of *bisim* and *bisim*2 on the kanban05 model. At its peak, the BDD used in *bisim*2 is about 4 times as large as the BDD used in *bisim*. It appears that the chaining heuristic used in *bisim* keeps R much smaller than the breadth first search heuristic employed in *bisim*2. We also measured the peak size of the BDDs X_a and Y_a, which represent the results of the ∀preimage and relcomp operations respectively. For *bisim*, X_a contains 1,965,391 nodes and Y_a contains 1,165,512 at its peak. The peak sizes of *bisim*2 are much higher, with 11,405,465 nodes for X_a and 16,761,754 nodes for Y_a. Together, the transition relation BDDs contain a total of 2108 nodes, which is negligible compared to the other BDDs.

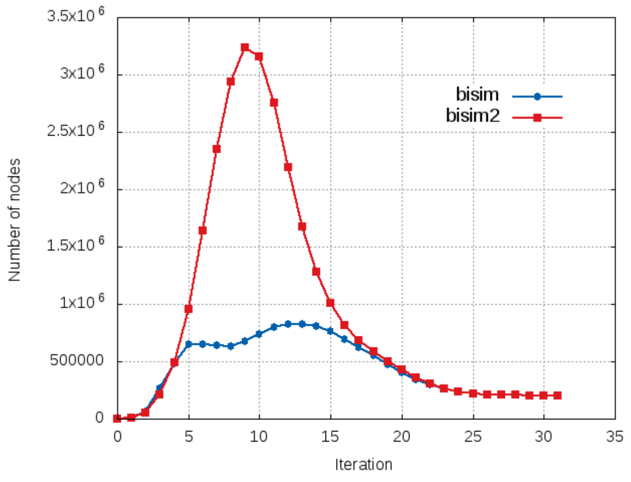
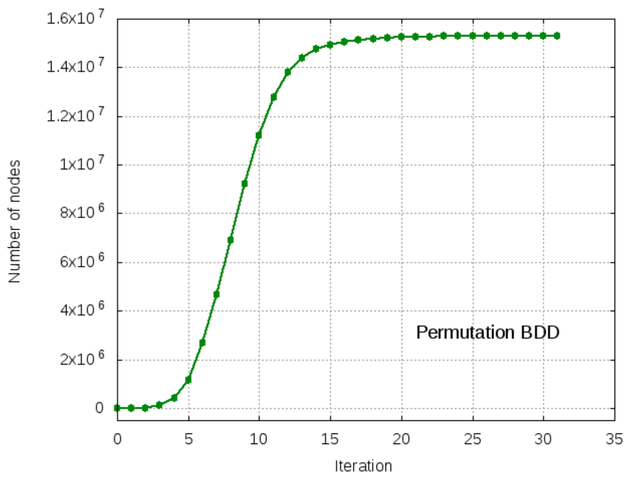


Figure 5.1: Number of nodes contained in R per iteration for the execution of *bisim* and *bisim*2 on the kanban05 model.

The kanban05 model contains 16,772,032 states, which, after bisimulation, are divided over 5,033,631 equivalence classes. We therefore expected that the permutation BDD P (see Chapter 4) used in Van Dijk's algorithm would be smaller than the relation BDD R used in *bisim*, as it requires less variables to represent the maximal bisimulation. However, the leftmost graph shown in Figure 5.2 reveals that P becomes much larger than R. The signatures BDD σ grows even larger still, as can be seen in the rightmost graph in Figure 5.2. At its peak, σ is at least two orders of magnitude larger than R. We presume that the equivalence relation BDDs stay smaller because of their interleaved variable ordering. In the permutation BDDs, all block variables are located below the state variables. In the signatures BDDs, there are action variables located in between the state and block variables as well. Encoding different objects underneath each other in a BDD typically causes the BDD to blow up in size. So, even though permutation BDDs contain less variables, they still grow much larger than equivalence relation BDDs. Interleaved variable orderings will not be effective for the permutation and signatures BDDs, as there is no correlation between states, actions, and block number encodings.

Observe that P and σ also do not decrease in size at any time during the execution of the algorithm. This is likely caused by the fact that Van Dijk's algorithm adds new block variables at the bottom of the BDDs during its execution.



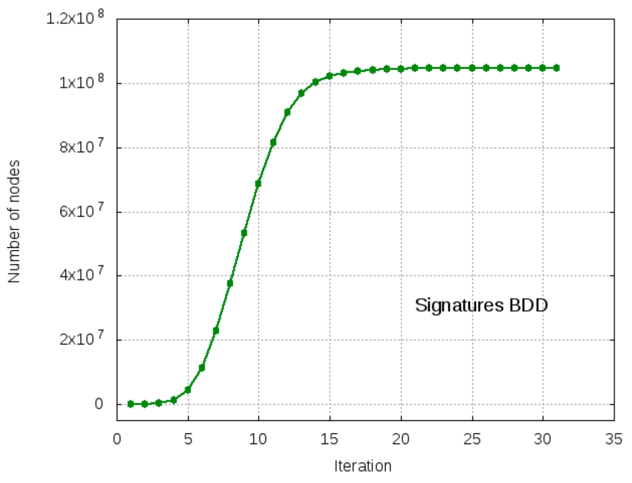


Figure 5.2: Number of nodes contained in permutation BDD P and signatures BDD σ per iteration for the execution of Van Dijk's algorithm on the kanban05 model.

The size of the signatures BDD is crucial for the time efficiency of Van Dijk's algorithm, as it is the result of the time consuming signature computation step. One iteration of Van Dijk's algorithm takes one symbolic step, while a single iteration in *bisim* could require at most $2 \cdot |A|$ symbolic steps. Coincidentally, both algorithms terminate after 31 iterations for the kanban05 model. Thus, the difference in size between the BDDs used in the two algorithms seems to outweigh the difference in symbolic steps in terms of time efficiency for the kanban05 model.

As BDDs can grow exponentially, the differences between the number of nodes in the relation and partition BDDs are not as substantial for smaller models. For instance, for the kanban02 model R peaks at 13,883 nodes, while P peaks at 19,057 nodes and σ at 86,689 nodes. This causes Van Dijk's algorithm to be more efficient than *bisim* for smaller models, as is shown by the results for the kanban02 and -03 models in Table 5.1.

5.4 Validation

We have used the results of Van Dijk's algorithm to validate the results of *bisim* and *bisim*2. In order to do so, the partition returned by Van Dijk's algorithm had to be converted to a relation. Unfortunately, this conversion was too time consuming to validate the results of the two largest models. All other results were successfully validated.

Chapter 6

Conclusions

We have presented a parallel relation-based algorithm for symbolic bisimulation minimization. For an efficient implementation in the BDD package, we derived novel coarse-grained BDD operations (∀preimage and relcomp). The correctness of the derived BDD is assured by the derivation. Even though the derivation method is straightforward and fundamentally connected with BDDs, we have not yet seen it applied in other works.

We have compared the performance of our algorithm to the state-of-the-art parallel symbolic bisimulation algorithm, which is partition-based. The surprising result is that our relation-based algorithm outperforms the partition-based algorithm by a significant margin. The consensus in the literature is that partition-based algorithms handle smaller BDDs and should therefore be more efficient than relation-based algorithms. However, our results suggest that the opposite is true. We found that, even though partition-based BDDs generally require less variables to represent the maximal bisimulation, their inefficient variable ordering make them considerably larger than corresponding relation-based BDDs. In particular, partition-based approaches cannot use the interleaved variable ordering that is so efficient for storing relations in BDDs.

As future work we advocate more research on the differences between relation-based and partition-based symbolic bisimulation algorithms, as many contradictions can be found in the current literature. Furthermore, we are interested in the application of the saturation heuristic on BDDs. This heuristic has significantly improved MDD algorithms in the past, but has not yet been applied to BDD algorithms. We suspect that the use of saturation would also be beneficial to our symbolic bisimulation algorithm. Another possible improvement to our algorithm is the inclusion of actions in the transition relation BDD. This increases the number of nodes which are needed to store all possible transitions, but reduces the number of symbolic operations which need to be performed.

Appendix A

Program Verification using Dafny

Dafny [21] is a theorem prover for program and algorithm verification. In the past, the tool was used by van de Pol [32] to successfully verify the Nested Depth-First Search algorithm. Stimulated by this effort, we used Dafny for the verification of symbolic operations, with the goal to ultimately verify the $\forall$preimage and relcomp operations (Algorithms 4 and 5). We were able to successfully verify the union and intersection operations, and partially verify the (pre)image operation, showing completeness. In what follows, we will describe Dafny in more detail, and present our verification attempts and the difficulty in verifying soundness for the (pre)image operation. All presented code is accessible online at [1].

A.1 Dafny

Dafny is both an automatic program verifier (or theorem prover) and an imperative programming language. Programs written in the Dafny programming language can be augmented with specification annotations, such as invariants and pre-/postconditions, which describe the intended meaning of the program, as presented by Floyd [18]. The program verifier automatically verifies the functional correctness of Dafny programs based on the provided annotations. It also type checks the program, confirms the absence of run time errors, and assures termination. Programs are verified in a modular fashion, such that the correctness of a program is implied by the correctness of all its subroutines. When verification fails, the user is provided with a sensible error message about the origin of the failure. Dafny's automatic program verifier is build upon the SMT solver Z3 [13].

The Dafny programming language supports sequential programs, generic classes, and dynamic allocation. It also provides abstract data types, such as sets, sequences, and user-defined algebraic data types. The keyword *method* is used to declare subroutines. Specification annotations are added between the declaration and the body of a method. Annotations include preconditions (*requires*), postconditions (*ensures*), and invariants (*invariant*). Annotations are expressed using basic forms of logic and can include *functions* and *predicates*. Functions are parametrized expressions and are mainly used for queries, such as computing the absolute value of an integer. Predicates are used to define Boolean functions. Both functions and predicates can only be used in annotations, unless their declaration is appended by the method keyword. Like methods, the intended meaning of functions and predicates can be specified by annotations. The keyword *decreases* is used in annotations to help Dafny decide the termination of a program. These annotations state that a data object should decrease in positive size after the execution of a loop iteration or a recursive subroutine call.

A.2 The verification of the union and intersection operations

Figure A.1 expresses the union operation in Dafny syntax. Line 1 defines BDDs as an algebraic data type. This avoids the need for a unique table, as instances are treated as terms. A Bdd can be a 0-leaf (O), a 1-leaf (I), or an internal node (Node) containing a low child (l), a high child (h), and an integer (v) which represents the variable label. The data type xint and function method x (Lines 3 and 4) are used to compare and distinguish leafs and internal nodes. Leaf nodes are given the variable label Infty by x, such that the variable labels of leaf nodes and internal nodes can be compared. The function methods low and high (Lines 6 and 7) are used to access the low and high child of its argument Bdd X respectively. They also take a variable label v as an argument, such that the child node is only returned if v is contained in the root of X. This ensures that the two Bdd operands of the union operation are traversed at the same pace, as the algorithm only goes deeper into the BDD which contains the lowest variable label. The function method min (Line 9) simply returns the smallest value of its two arguments and is used to find the node with the lowest variable label in two BDDs.

```
1  datatype Bdd = Node(l: Bdd, h: Bdd, v: int) | O | I
2
3  datatype xint = Int(n: int) | Infty
4  function method x(X: Bdd): xint { if X = O ∨ X = I then Infty else Int(X.v) }
5
6  function method low(X: Bdd, v: int): Bdd { if x(X) = Int(v) then X.l else X }
7  function method high(X: Bdd, v: int): Bdd { if x(X) = Int(v) then X.h else X }
8
9  function method min(a: int, b: int): int { if a < b then a else b }
10
11 method union(A: Bdd, B: Bdd) returns (R: Bdd)
12 {
13     if A = I ∨ B = I { return I; }
14     if A = O ∨ A = B { return B; }
15     if B = O         { return A; }
16     var v := min(A.v,B.v);
17     var L := union(low(A,v),low(B,v));
18     var H := union(high(A,v),high(B,v));
19     if H = L { return H; }
20     R := Node(L,H,v);
21 }
```

Figure A.1: Expressing the union operation in Dafny.

The algorithm for the union operation is displayed in Lines 11–21. It takes as arguments two Bdds A and

B, and returns a Bdd R which should represent the union of A and B. The algorithm can be obtained from apply (Algorithm 1) by implementing some minor changes. Lines 13–15 showcase three base cases which are specific for the union operation. Line 19 avoids the creation of redundant nodes. Duplicates nodes do not exist, as Bdds are algebraic data types. The operation has been verified with and without the use of an operation cache. For simplicity, we only show the version of union that does not use cache operations. Apart from these changes, the algorithm functions exactly like apply.

Dafny successfully verifies the union operation displayed in Figure A.1. However, this code does not yet contain any specification annotations. Thus, Dafny only verifies the absence of run time errors, such as null dereferences (e.g. A.v, where A is a leaf node). In order to also verify the functional correction and termination of union, we need to add annotations to the code. For the functional correctness of the operation we need to ensure two things. First, the result should represent the set with exactly all elements which are present in A or B, but no more. In other words, the result should be mathematically sound and complete. Second, the result Bdd R should have the correct form. That is, it should be both reduced and ordered, its root can not be ordered before the roots of A and B, and R should be defined over the same variables as A and B.

Figure A.2 displays the added annotations which declare the correctness conditions just described. The ror predicate (Line 12) ensures that its argument Bdd is reduced and ordered. Reduction is checked by the predicate reduced (Line 8). It recursively traverses its argument Bdd and makes sure that no internal node has an identical low and high child. Again, Bdds can not contain duplicate nodes. The predicate ordered (Line 6) ensures that its argument Bdd is ordered. It does so by calling another predicate increasing (Line 1), which recursively traverses its argument Bdd to ensure that every internal node contains a variable label which is greater than the variable label of its parent. As a precondition union requires that both of A and B are reduced and ordered (Line 40). In return, Dafny proves that R is also reduced and ordered, as is required by the postcondition in Line 41. The second postcondition (Line 42) checks whether the root of R is not ordered before the roots of A and B. The third postcondition (Line 43) makes sure that R is defined over the same variables as A and B using the defOver predicate (Line 23). This predicate takes as arguments a Bdd X and a set of integers (= variable labels) s. It recursively traverses X and assures that the variable label of every internal node is contained in s. The last postcondition (Line 44) ensures the soundness and completeness of union. It states that any binary string contained in A or B must also be contained in R, and vice versa. The contains predicate (Line 27) takes as arguments a Bdd X and a Boolean sequence i and checks whether i is contained in X. We do this by traversing X and following the path defined by i. If the path ends in a 1-leaf we return true, if it ends in a 0-leaf we return false. Note that we also return true if the path defined by i ends before reaching a leaf (Line 35). Because R should contain all paths from both A and B, the union operation might remove some redundant nodes from the paths in R which are present in A and/or B. For instance, let A and B be defined over variables x_1 and x_2. The Bdd A might only contain the valuation $x_1 \leftarrow 1$, $x_2 \leftarrow 0$, while B only contains $x_1 \leftarrow 1$, $x_2 \leftarrow 1$. Now R will not contain any nodes with variable label x_2, as it is redundant.

Thus if i is the singleton sequence [1], a 1-leaf will be reached in R but not in A and B. By returning true for sequences which don't reach a leaf, we can still proof the soundness of R.

```
 1  predicate increasing(X: Bdd, prev_X: Bdd)
 2      requires prev_X ≠ O ∧ prev_X ≠ I
 3      decreases X
 4  { X = O ∨ X = I ∨ (X.v > prev_X.v ∧ increasing(X.l,X) ∧ increasing(X.h,X)) }
 5
 6  predicate ordered(X: Bdd) { X = O ∨ X = I ∨ (X.v ≥ 0 ∧ increasing(X.l,X) ∧ increasing(X.h,X)) }
 7
 8  predicate reduced(X: Bdd)
 9      decreases X
10  { X = I ∨ X = O ∨ (X.h ≠ X.l ∧ reduced(X.h) ∧ reduced(X.l)) }
11
12  predicate ror(X: Bdd) { reduced(X) ∧ ordered(X) }
13
14  predicate method xleq(a: xint, b: xint)
15  {
16      if b = Infty then true
17      else if a = Infty then false
18      else a.n ≤ b.n
19  }
20
21  function method xmin(a: xint, b: xint): xint { if xleq(a,b) then a else b }
22
23  predicate defOver(X: Bdd, s: set<int>)
24      decreases X
25  { X = I ∨ X = O ∨ (X.v in s ∧ defOver(X.h,s) ∧ defOver(X.l,s)) }
26
27  predicate contains(X: Bdd, i: seq<bool>)
28      requires ordered(X)
29      decreases X
30  {
31      match X
32      case O ⇒ false
33      case I ⇒ true
34      case Node(L,H,v) ⇒
35          if v ≥ |i| then true
36          else if i[v] then contains(H,i) else contains(L,i)
37  }
38
39  method union(A: Bdd, B: Bdd) returns (R: Bdd)
40      requires ror(A) ∧ ror(B)
41      ensures ror(R)
42      ensures xleq(xmin(x(A),x(B)),x(R))
43      ensures ∀ s: set<int> • defOver(A,s) ∧ defOver(B,s) ⟹ defOver(R,s)
44      ensures ∀ i : seq<bool> • contains(R,i) ⟺ contains(A,i) ∨ contains(B,i)
45      decreases A, B
46  { ... }
```

Figure A.2: The added annotations and predicates for the verification of the union operation.

Finally, the decreases annotations in all recursive subroutines help Dafny in deciding the termination of the algorithms. Dafny is able verify the annotated union algorithm without any problems. A similar proof can be generated for the symbolic intersection operation. One small alteration is needed for this proof. Line 36 in the contains predicate should return false for the intersection operation, as R should not contain any shorter paths than found in A and B.

A.3 The verification of the image and preimage operations

Figure A.3 presents the image operation $\exists \vec{x} : S \cap R$ in Dafny syntax. The operation is performed by the relnext algorithm, shown in Lines 27–51. This algorithm was obtained by the derivation method described in

```
1  predicate disjoint(s: set<int>, t: set<int>) { ∀ i: int • i in s ⟹ i ∉ t }
2
3  predicate method xless(a: xint, b: xint)
4  {
5      if b = Infty ∧ a ≠ Infty then true
6      else if a = Infty then false
7      else a.n < b.n
8  }
9
10 predicate containsR(R: Bdd, a: seq<bool>, b: seq<bool>, s: set<int>, s': set<int>)
11     requires ordered(R)
12     requires defOver(R,s+s')
13     decreases R
14 {
15     match R
16     case O ⇒ false
17     case I ⇒ true
18     case Node(L,H,v) ⇒
19     if v in s then if v ≥ |a| then false
20                    else if a[v] then containsR(H,a,b,s,s')
21                    else containsR(L,a,b,s,s')
22     else if v ≥ |b| then false
23          else if b[v] then containsR(H,a,b,s,s')
24          else containsR(L,a,b,s,s')
25 }
26
27 method relnext(S: Bdd, R: Bdd, s: set<int>, s': set<int>) returns (T: Bdd)
28     requires ror(S) ∧ ror(R)
29     requires defOver(S,s) ∧ defOver(R,s+s')
30     requires |s| = |s'| ∧ disjoint(s,s')
31     ensures ror(T)
32     ensures xleq(xmin(x(S),x(R)),x(T))
33     ensures defOver(T,s')
34     ensures ∀ b: seq<bool> • (∃ a: seq<bool> • contains(S,a) ∧ containsR(R,a,b,s,s')) ⟹ contains(T,b)
35  // ensures ∀ b: seq<bool> • contains(T,b) ⟹ (∃ a: seq<bool> • contains(S,a) ∧ containsR(R,a,b,s,s'))
36     decreases R, S
37 {
38     if S = O ∨ R = O { return O; }
39     if R = I { return I; }
40     if xless(x(R),x(S)) ∧ R.v in s' {
41         var L := relnext(S,R.l,s,s');
42         var H := relnext(S,R.h,s,s');
43         return if L = H then L else Node(L,H,R.v);
44     }
45     else {
46         var m := if xless(x(R),x(S)) then R.v else S.v;
47         var L := relnext(low(S,m),low(R,m),s,s');
48         var H := relnext(high(S,m),high(R,m),s,s');
49         T := union(L,H);
50     }
51 }
```

Figure A.3: Our attempt to verify the image operation.

Chapter 3. The algorithm takes as arguments a Bdd S representing a set, a Bdd R representing a relation, and two sets of variable labels s and s'. It returns a Bdd T which should represent the image of S under R.

In order to prove the functional correctness of relnext we again need to show two things. First, T should have the correct form. This is assured by the pre- and postconditions in Lines 28–33. Second, T needs to be sound and complete. The completeness of T is ensured by the postcondition in Line 34. In words, it states that if $a \in S$ and $(a, b) \in R$ then T should contain b. The completeness of R can be successfully verified by Dafny. A similar proof can be generated for the preimage operation.

The soundness postcondition (Line 35), cannot be verified. In words, the soundness condition states that if $b \in T$ then there should exist an a such that $a \in S$ and $(a, b) \in R$. If we uncomment the soundness postcondition, Dafny will give an error message stating the verification process has timed out. After further analysis, we found that checking the soundness of R is much more complex than checking its completeness.

The soundness condition can be rewritten to the following form

$$\forall b : (b \notin T \lor \exists a : (a \in S \land (a, b) \in R))$$

This is different from completeness, which can be rewritten to the following form

$$\forall b : (\forall a : \neg(a \in S \land (a, b) \in R) \lor b \in T)$$

Leino, the creator of Dafny, has stated that proving that an existential quantifier holds can be problematic for Dafny [3]. Therefore, universal quantification over an expression containing an existential quantifier is inherently difficult to validate. The completeness condition does not suffer from this difficulty. Due to the aforementioned issue and time constraints we gave up on the verification of the $\forall$`preimage` and `relcomp` operations.

Bibliography

[1] Dafny-verification-symbolic-operations. https://github.com/RichardHuybers/Dafny-Verification-Symbolic-Operations/blob/master/symbolic_operations.dfy. Accessed: 25-10-2018.

[2] Sigrefmc_bisim. https://github.com/RichardHuybers/sigrefmc_bisim. Accessed: 25-10-2018.

[3] Trivial asserts about sequences fail. https://github.com/Microsoft/dafny/issues/184#issuecomment-376323353. Accessed: 25-10-2018.

[4] C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 55 Hayward Street, Cambridge, MA 02142, 2008.

[5] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, pages 197–203, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[6] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In G. von Bochmann and D. K. Probst, editors, *Computer Aided Verification*, pages 96–108, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[7] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[9] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[11] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *Computer Aided Verification*, pages 147–158, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[12] A. E. Dalsgaard, S. Enevoldsen, K. G. Larsen, and J. Srba. Distributed computation of fixed points on dependency graphs. In M. Fränzle, D. Kapur, and N. Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications*, pages 197–212, Cham, 2016. Springer International Publishing.

[13] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] R. Enders, T. Filkorn, and D. Taubner. Generating bdds for symbolic model checking in ccs. *Distributed Computing*, 6(3):155–164, 1993.

[15] K. Fisler and M. Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, pages 115–132, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[16] K. Fisler and M. Y. Vardi. Bisimulation and model checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 338–342, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[17] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[18] R. W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[19] J. F. Groote, D. N. Jansen, J. J. A. Keiren, and A. J. Wijs. An o(mlogn) algorithm for computing stuttering equivalence and branching bisimulation. *ACM Trans. Comput. Logic*, 18(2):13:1–13:34, June 2017.

[20] M. Huth and M. Ryan. *Logic in Computer Science*, chapter Verification by Model Checking, pages 241–241. Cambridge University Press, The Edinburgh Building, Cambridge CB2 8RU, UK, 2004.

[21] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[22] X. Liu and S. A. Smolka. Simple linear-time algorithms for minimal fixed points. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming*, pages 53–66, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[23] R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science*, volume 92, Berlin, 1980. Springer Verlag.

[24] M. Mumme and G. Ciardo. An efficient fully symbolic bisimulation algorithm for non-deterministic systems. 24, 2013.

[25] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, Dec. 1987.

[26] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[27] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[28] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Electrical Engineering*, 57(12):713–723, 1938.

[29] T. L. Siaw. Saturation for ltsmin. Master's thesis, University of Twente, 2012.

[30] M. Solé and E. Pastor. Traversal techniques for concurrent systems. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design*, pages 220–237, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[31] F. Somenzi. Binary decision diagrams. In *Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[32] J. C. van de Pol. Automated verification of nested dfs. In M. Núñez and M. Güdemann, editors, *Formal Methods for Industrial Critical Systems*, pages 181–197, Cham, 2015. Springer International Publishing.

[33] T. van Dijk. *Sylvan: multi-core decision diagrams*. PhD thesis, University of Twente, 2016.

[34] T. van Dijk, A. Laarman, and J. van de Pol. Multi-core bdd operations for symbolic reachability. *Electronic Notes in Theoretical Computer Science*, 296:127 – 143, 2013. Proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC).

[35] T. van Dijk and J. van de Pol. Multi-core symbolic bisimulation minimisation. *International Journal on Software Tools for Technology Transfer*, 20(2):157–177, 2018.

[36] T. van Dijk and J. C. van de Pol. Lace: Non-blocking split deque for work-stealing. In L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 206–217, Cham, 2014. Springer International Publishing.

[37] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. Sigref – a symbolic bisimulation tool box. In S. Graf and W. Zhang, editors, *Automated Technology for Verification and Analysis*, pages 477–492, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.