EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Integrating a BDD prover and a DPLL SAT solver for abstract data types

Schuijers, M.P.

Award date:
2006

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 17. May. 2025

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Integrating a BDD prover and a DPLL SAT solver for Abstract Data Types

by
M.P. Schuijers

Supervisor:
Dr. J.C. van de Pol

Eindhoven, January 2006

Abstract

This thesis presents various techniques that can be used in the process of theorem proving and solving satisfiability problems. Particular interest in this goes to satisfiability problems in the (quantifier-free) logic of equality over an infinite ground term algebra (sometimes referred to as abstract data types). As a basis for the research work an existing, incomplete BDD prover was taken. The techniques presented in this thesis focus on strengthening this prover. A stronger prover results in a better state space reduction which in its turn allows the analysis of more realistic industrial systems. In total three different techniques will be presented, of which one involves integrating the prover with a DPLL SAT solver. Furthermore, a tool has been developed which allows the use of these techniques in combination with the prover. The functionality of this tool will also be discussed in this thesis.

Contents

1 Introduction **1**

2 State of the art **3**
2.1 Binary Decision Diagrams . . . 3
2.2 Equational Binary Decision Diagrams . . . 4
2.3 The μCRL toolset . . . 6
2.4 A prover for the μCRL toolset . . . 7

3 Finding true -and false-guards **9**
3.1 Definitions and properties . . . 9
3.2 First approach . . . 10
3.2.1 Observations . . . 10
3.2.2 Algorithm . . . 11
3.3 Graph-based approach . . . 11
3.3.1 Finding possible true -and false-guards . . . 11
3.3.2 Filtering out the true -and false-guards . . . 12
3.3.3 Algorithm . . . 13
3.4 Application example . . . 14

4 Simplification operators **15**
4.1 Desired properties . . . 15
4.2 The Restrict operator . . . 16
4.2.1 Rules . . . 16
4.2.2 Verification . . . 16
4.2.3 Algorithm . . . 18
4.3 The Constrain Operator . . . 18
4.3.1 Rules . . . 19
4.3.2 Verification . . . 19
4.3.3 Algorithm . . . 20
4.4 Application example . . . 21

5 Simplification using a SMT solver **23**
5.1 General framework . . . 23
5.1.1 Choosing a path . . . 24
5.1.2 Translating a path . . . 24
5.1.3 Checking a path . . . 24
5.1.4 Interpreting the result . . . 24
5.2 Simplifying BDDs with equalities using Tera . . . 25
5.2.1 Naive version . . . 25
5.2.2 Improving the naive version . . . 27
5.2.3 Experimental results . . . 31
5.2.4 Future work . . . 32

6 Prover Shell **35**
6.1 The user interface . . . 35
6.1.1 Commands . . . 35
6.1.2 An example scenario . . . 38
6.2 The libraries . . . 40
6.2.1 Libraries created for the shell . . . 40
6.2.2 Other libraries used . . . 43

7 Conclusions and directions for further work **45**

Bibliography **47**

A Graph Theory **49**
A.1 Basic definitions . . . 49
A.2 Representations of graphs . . . 50
A.3 Depth-first search . . . 51
A.3.1 Classification of edges . . . 52
A.4 Articulation points . . . 53

B Proofs of properties of the simplification operators **55**
B.1 The Restrict operator . . . 55
B.1.1 Preserving logical equivalence . . . 55
B.1.2 Path-consistency . . . 57
B.2 The Constrain operator . . . 59
B.2.1 Preserving logical equivalence . . . 59
B.2.2 Path-consistency . . . 59

C Contents of `standard.mcrl` **61**

Chapter 1

Introduction

In this thesis a number of techniques will be presented which can be used in the process of theorem proving and solving satisfiability problems. Particular interest in this goes to satisfiability problems in the (quantifier-free) logic of equality over an infinite ground term algebra (sometimes referred to as abstract data types, or inductive data types). An instance of a formula in this logic would be:

$$(x = S(y) \vee y = S(head(tail(z)))) \wedge z = cons(x, w) \wedge (x = 0 \vee z = nil).$$

As a basis for the research work, an existing EQ-BDD prover was taken. The main functionality of this prover is to transform a formula into an equivalent Equational Binary Decision Diagram (EQ-BDD) with respect to some algebraic data specification. The resulting EQ-BDD can be either `True`, `False` or something else. In the first two cases the original formula was a tautology or a contradiction, respectively. In the third case nothing is known due to incompleteness of the prover. This thesis focuses on strengthening the prover by presenting techniques that can be used to simplify this last group of EQ-BDDs (in the ideal case resulting in `True` or `False`). A stronger prover results in a better state space reduction which in its turn allows the analysis of more realistic industrial systems. In total, three different techniques will be presented.

First two special types of guards in an EQ-BDD will be introduced. These guards have the property that their truth-value is invariant under assumption that the underlying function of the EQ-BDD evaluates to `true`. Finding such guards appears to be useful for eliminating certain variables. Two algorithms for this, of which one has been implemented, will be presented.

Subsequently, two existing operators for simplifying an EQ-BDD given another EQ-BDD will be discussed and extended. If it is known to which function value an EQ-BDD evaluates, it is possible to use this knowledge to simplify another EQ-BDD. In this way, the operators can be used to remove branches that are unreachable given an invariant.

Finally, a general approach for simplifying BDDs using a *Satisifiability Modulo Theories* (SMT) solver will be presented. This approach consists of four basic steps, that can be performed for simplifying BDDs using different SMT solvers. These steps correspond to choosing a path in the BDD, translating the path to the correct input format for the SMT solver, checking the satisfiability of the translated path and interpreting the result of the satisfiability check. A concrete algorithm for integrating a DPLL SAT solver for abstract data types with the prover will be provided.

In addition to these three techniques, a tool has been designed which integrates the techniques with the existing prover. This tool also adds constructor-induction for recursive data types to the prover. This tool and its functionality will also be discussed in this thesis.

Outline. Chapter 2 gives a brief overview of the current state of the art with respect to the μCRL toolset. The theory of BDDs, their extension with equalities, and the current toolset and prover will be discussed in this chapter. In chapter 3 a means of finding the special types of guards mentioned above will be discussed. The operators for simplifying an EQ-BDD, based on

knowledge about the function value of another EQ-BDD, will be defined in chapter 4. In chapter 5 the general approach for simplifying BDDs using a suitable SMT solver will be defined. Besides this general approach, also a practical example of this will be discussed in this chapter. The tool that has been created in between the research work, and which can be used to apply the techniques of chapters 3, 4 and 5, will be discussed in chapter 6. Finally, Chapter 7 contains some conclusions and directions for further work.

Chapter 2

State of the art

This chapter gives an overview of the μCRL toolset, the current prover for the μCRL toolset and the analysis tools based on this prover. First the basics of *Binary Decision Diagrams* (BDDs) and *Equational Binary Decision Diagrams* (EQ-BDDs) will be discussed. EQ-BDDs (which are an extension of BDDs) form the basis on which the prover operates. After this, an overview of the μCRL toolset will be given. The chapter concludes with a short overview of the current prover and the tools based on this prover.

2.1 Binary Decision Diagrams

The basics of *Binary Decision Diagrams* can be found in [10, 26] but will be repeated here for reference.

Definition 2.1.1 (Binary Decision Diagram)**.** A *Binary Decision Diagram* (BDD) represents a boolean function as a rooted, directed acyclic graph. The leafs of this graph are labelled with `true` and `false`. Each internal node v is labelled with a boolean variable $var(v)$ and has two outgoing edges towards its children: $low(v)$ corresponding to the case where the variable evaluates to `false`, and $high(v)$ corresponding to the case where the variable evaluates to `true`. For a given assignment to the variables, the function value can be determined by following a path from the root to a leaf.

Example 2.1.2. Figure 2.1 illustrates BDDs representing the boolean functions $(p \vee q) \wedge r$ (a) and $p \wedge q \wedge \neg r$ (b), where the symbols $\vee$, $\wedge$, and $\neg$ are used to indicate boolean Or, And, and Not, respectively. A dashed line indicates the edge where the variable evaluates to `false`, a solid line indicates the edge where the variable evaluates to `true`.

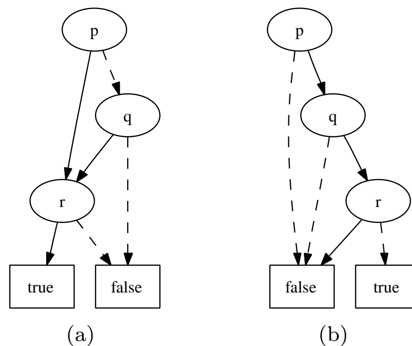

Figure 2.1: Example (O)BDDs.

Although the idea of representing boolean functions as BDDs is quite old (they were introduced by Lee in 1959 [24]), their widespread use as a data structure for boolean manipulation only started when Bryant imposed further restrictions on the ordering of the decision variables in a BDD in 1986 [10]. These restrictions enabled the development of algorithms for manipulating BDDs in a more efficient manner.

Definition 2.1.3 (Ordered BDD)**.** Let $\prec$ be a total order on the set of boolean variables. Then a BDD is called *ordered* (OBDD) with respect to the variable order $\prec$ if the variables on every path from the root to a leaf are encountered in ascending order with respect to $\prec$.

Example 2.1.4. The BDDs depicted in figure 2.1 are examples of OBDDs with respect to the variable ordering $p \prec q \prec r$.

Although the ordering of the variables may be chosen arbitrarily, in practice, the choice of a good ordering is crucial for the efficiency of manipulation. This issue will not be discussed here.

Despite being a step forward toward an efficient datastructure, OBDDs are not optimal. There is the possibility that redundancies occur within an OBDD. Two types of redundancy are possible:

- For a node v the high -and low-successor can be identical. In this case, the decision in node v does not yield any new information.
- Within the decision diagram, certain sub-diagrams can occur multiple times. In this way, the same information about the function is represented multiple times.

To remove these types of redundancies from an OBDD, the following two reduction rules are defined:

- **Elimination rule:** If the high-edge and the low-edge of a node v point to the same node u, then remove v and direct all incoming edges of v to u.
- **Merging rule:** If the internal nodes u and v are labelled by the same boolean variable, their high-edges lead to the same node, and their low-edges lead to the same node, then remove one of the nodes u, v and redirect all incoming edges of this node to the remaining one.

Definition 2.1.5 (Reduced (O)BDD)**.** A (O)BDD is called a *Reduced (and Ordered) Binary Decision Diagram* (R(O)BDD) if it does not contain redundant tests (e.g. the elimination rule cannot be applied) and if it is maximally shared (e.g. the merging rule cannot be applied).

For a fixed order on the variables, a ROBDD is a *canonical* representation of a boolean function: two boolean functions are equal if, and only if, their ROBDD representations are equal.

One of the more important operations for constructing OBDDs is the Apply operation [10]. Given two OBDDs ϕ representing a function f and ψ representing a function g, and a binary boolean operator $\otimes$ (e.g. And or Or), Apply returns the OBDD representing the function $f \otimes g$. Pseudocode for the Apply algorithm is given in Figure 2.2.

2.2 Equational Binary Decision Diagrams

The basics of *Equational Binary Decision Diagrams* can be found in [19]. The approach taken there extends the notion of orderedness to capture the properties of reflexivity, symmetry, transitivity and substitutivity. The advantage of the method is that satisfiability checking for a given ordered EQ-BDD can be done immediately. However, it is restricted to the case where equalities do not contain function symbols. In [4] EQ-BDDs were extended by incorporating some *interpreted* functions: natural numbers with zero and successor were added. An alternative solution, with a different orientation of the equations was provided in [3]. Finally, in [29], a BDD respresentation for the logic of equality with *uninterpreted* functions was presented. This representation allows equalities between ground terms as labels. In this thesis, the definitions introduced in [19] will be used with the changed orientation of [3]. These definitions will be repeated here for reference.

Apply$(\phi, \psi, \otimes)$: BDD =
1 **begin**
2 **if** ($\otimes$ = "$\vee$" **and** (ϕ = True **or** ψ = True)) **then return** True;
3 **if** ($\otimes$ = "$\vee$" **and** ϕ = False) **then return** ψ;
4 **if** ($\otimes$ = "$\vee$" **and** ψ = False) **then return** ϕ;
5 **if** ($\otimes$ = "$\wedge$" **and** (ϕ = False **or** ψ = False)) **then return** False;
6 **if** ($\otimes$ = "$\wedge$" **and** ϕ = True) **then return** ψ;
7 **if** ($\otimes$ = "$\wedge$" **and** ψ = True) **then return** ϕ;
8 *result* := *"get entry for* $(\phi, \psi, \otimes)$ *from hash-table"*;
9 **if** ($result \neq null$) **then return** *result*;
10 **if** ($root(\phi) \prec root(\psi)$) **then**
11 $result := ITE(root(\phi)$, Apply$(high(\phi), \psi, \otimes)$, Apply$(low(\phi), \psi, \otimes))$;
12 **else if** ($root(\phi) = root(\psi)$) **then**
13 $result := ITE(root(\phi)$, Apply$(high(\phi), high(\psi), \otimes)$, Apply$(low(\phi), low(\psi), \otimes))$;
14 **else if** ($root(\phi) \succ root(\psi)$) **then**
15 $result := ITE(root(\psi)$, Apply$(\phi, high(\psi), \otimes)$, Apply$(\phi, low(\psi), \otimes))$;
16 *"create an entry for* $(\phi, \psi, \otimes)$ *with value 'result' in hash-table"*;
17 **return** *result*;
18 **end**

Figure 2.2: The Apply algorithm.

Definition 2.2.1 (Equational Binary Decision Diagram)**.** An *Equational Binary Decision Diagram* (EQ-BDD) is an ordinary BDD with the only difference that a guard can, in addition to boolean variables, also consist of equations between domain variables.

Example 2.2.2. Figure 2.3 illustrates EQ-BDDs representing the formulae $y = x \wedge z = y$ (a) and $p \vee (y = x \wedge z \neq x)$ (b). A dashed line indicates the edge where the guard is assigned to `false`, a solid line indicates the edge where the guard is assigned to `true`.

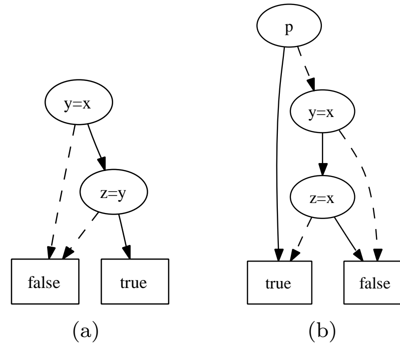

Figure 2.3: Example EQ-BDDs.

In order to compute whether an EQ-BDD is tautological or satisfiable, it first needs to be ordered. Like ordinary BDDs, in an *ordered* EQ-BDD the guards on a path may only appear in a fixed order.

Definition 2.2.3 (Order on guards)**.** Let $\prec$ be a total order on the set of all variables (boolean

variables and domain variables). This order is extended to guards as follows:

$$\begin{array}{rl} p \prec q & \text{as usual,} \\ (y = x) \prec p & \text{if, and only if, } y \prec p, \\ p \prec (y = x) & \text{if, and only if, } p \prec y, \\ (v = u) \prec (y = x) & \text{if, and only if, either } v \prec y, \text{ or } v \equiv y \text{ and } u \prec x. \end{array}$$

Given the order on guards, it is possible to define an ordered EQ-BDD. As in [19], a system of reduction rules is defined, but now using the orientation of the equations as in [3].

Definition 2.2.4 (Ordered EQ-BDD)**.** An EQ-BDD is called *ordered* (EQ-OBDD) if, and only if, it is a normal form with respect to the following term rewrite system:

1. $ITE(G, T, T) \rightarrow T$.
2. $ITE(G, ITE(G, T_1, T_2), T_3) \rightarrow ITE(G, T_1, T_3)$.
3. $ITE(G, T_1, ITE(G, T_2, T_3)) \rightarrow ITE(G, T_1, T_3)$.
4. $ITE(G_1, ITE(G_2, T_1, T_2), T_3) \rightarrow ITE(G_2, ITE(G_1, T_1, T_3), ITE(G_1, T_2, T_3))$, provided $G_1 \succ G_2$.
5. $ITE(G_1, T_1, ITE(G_2, T_2, T_3)) \rightarrow ITE(G_2, ITE(G_1, T_1, T_2), ITE(G_1, T_1, T_3))$, provided $G_1 \succ G_2$.
6. $ITE(x = x, T_1, T_2) \rightarrow T_1$.
7. $ITE(x = y, T_1, T_2) \rightarrow ITE(y = x, T_1, T_2)$, provided $x \prec y$.
8. $ITE(y = x, T_1[y], T_2) \rightarrow ITE(y = x, T_1[x], T_2)$, provided $x \prec y$ and y occurs in T_1.

Rules 1-5 are the standard rules for obtaining ordered BDDs, with the difference that G now ranges over boolean variables and equations between domain variables. Rules 6-7 allow simplifcation of equations and EQ-BDDs. Finally, rule 8 allows to substitute equals for equals. Application of these reduction rules yields a logically equivalent EQ-BDD.

Example 2.2.5. Let $p \prec x \prec y \prec z$. Then the graph depicted in figure 2.3(b) is an example of an EQ-OBDD. The graph depicted in figure 2.3(a) is not an EQ-OBDD, but can be transformed to one by applying rule 8.

2.3 The μCRL toolset

The μCRL toolset [8, 31] (see `http://www.cwi.nl/~mcrl`) is a collection of tools for manipulating process and data descriptions written in μCRL (micro Common Representation Language). An overview of the toolset is given in Table 2.1.

μCRL [18] is a process algebraic language for describing communicating processes. It is based on the process algebra ACP [6, 15] extended with equational abstract data types [25]. Despite its simplicity, μCRL has proven to be quite adequate for describing and analysing (large) distributed systems and algorithms. It has been used for the verification of a large number of distributed systems, often in combination with a proof checker or theorem prover. For an overview of case studies see [17]. μCRL has been extended with features to express time [16], but this extension is not supported by the tools (except for the possibility to check the static semantic constraints).

The toolset is built around a restricted form of μCRL, called the *Linear Process Operator* (LPO) format [27]. An LPO describes a system as a process that consists of a series of summands; each summand gives a condition, an action that can be performed if that condition applies, and the modified process that results from that action.

mcrl	Checks whether a specification in (timed) μCRL is well formed, and linearises certain μCRL specifications.
msim	Allows interactive simulation of a system described in μCRL.
instantiator	Generates a finite transition system from a linearised μCRL specification.
pp	Pretty prints a linearised μCRL specification.
rewr	Normalises the data terms in a linearised μCRL specification.
constelm	Removes from a linearised μCRL specification the data parameters that are constant throughout any run of the process.
parelm	Removes from a linearised μCRL specification the data parameters and sum variables that do not influence the behavior of the system.
structelm	Expands the composite data types of a linearised μCRL specification.
sumelm	Replaces in a linearised μCRL specification the sum variables that must be equal to a certain data term by that data term.

Table 2.1: Overview of the μCRL toolset.

Example 2.3.1. Consider a simple buffer that can receive a datum with the action `read` and deliver it with the action `send`. The following specification is not linear, since there are two actions before the recursive call to the process `Buffer`:

```
proc Buffer = sum(d:Datum, read(d).send(d).Buffer)
```

The specification can be linearised by dividing the process `Buffer` in two phases. In phase `r` a datum is read, in phase `s` the datum is sent. The linearised `Buffer` process needs two parameters: one to keep track of a read datum and one to keep track of the phase.

```
proc Buffer(d1:Datum, p:Phase) =
    sum(d2:Datum, read(d2).Buffer(d2,s)) <| eq(p,r) |> delta
  + send(d1).Buffer(d1,r)                <| eq(p,s) |> delta
```

The tool `mcrl` checks whether a given specification is well formed μCRL and attempts to transform it into a linearised (i.e. LPO) format. This linearised form is stored in a binary format or as a plain text file. The resulting LPO and its data structures are stored as ATerms. The ATerm library [9] stores terms in an efficient way by using maximal subterm sharing and automatic garbage collection. All tools other than `mcrl` use LPOs as their starting point (see Table 2.1).

For more detailed information about the μCRL toolset the reader is referred to [31].

2.4 A prover for the μCRL toolset

In addition to the μCRL toolset an automated theorem prover has been developed to support the analysis of distributed systems specified in μCRL, as well as four analysis tools based on that prover [28]. The prover has been implemented as a library in C and is meant for developers that need theorem prover assistance in their analysis tools. The four tools are meant for end-users that want to analyze a particular distributed system specified in μCRL.

The main functionality of the prover is to transform a formula into an equivalent binary decision diagram (BDD) with respect to some algebraic data specification. The resulting BDD can be either `True`, `False` or something else. In the first two cases the original formula was a tautology or a contradiction, respectively. In the third case nothing is known due to incompleteness of the prover. In that case it is possible to compute counter examples and witnesses from the BDD for diagnostic purposes.

The four tools are listed in Table 2.2. These tools analyze and modify a μCRL specification in LPO format (see Section 2.3). The tools generate certain formulae from an LPO, representing invariants or confluence properties, and check these formulae by means of the prover. Depending on the result of the check they modify the LPO accordingly.

`formcheck`	Checks whether a certain formula is true in a given data specification.
`invcheck`	Checks whether a formula is an invariant of an LPO.
`invelm`	Simplifies an LPO by using an invariant to eliminate unreachable summands.
`confcheck`	Checks which τ-summands are confluent with respect to all other summands.

Table 2.2: The four prover-based tools.

An LPO can be used to generate the full state space. If a summand of the LPO is confluent (i.e. commutes with all other summands), it is possible to generate a much smaller state space [20]. Sometimes, invariant properties are helpful in proving confluence properties.

For more detailed information about the prover and the tools based on this prover the reader is referred to [28].

Chapter 3

Finding true -and false-guards

In this chapter an algorithm will be derived for finding those guards of an EQ-BDD that either always hold (evaluate to `true`) or never hold (evaluate to `false`) whenever the underlying function of the EQ-BDD evaluates to `true`. In the remainder of this chapter these guards will be referred to as true -and false-guards, respectively.

In Section 3.1 more formal definitions of true -and false-guards will be given first, as well as a list of properties of these guards (in case the EQ-BDD under consideration is ordered and reduced). In Sections 3.2 and 3.3 two approaches for finding the true -and false-guards of a EQ-BDD will be given.

Finally, in Section 3.4 an application example of the use of finding the true -and false-guards will be given.

Note: since EQ-BDDs are an extension of BDDs, both approaches presented in this chapter can also be applied to BDDs containing boolean variables only.

3.1 Definitions and properties

Let v be a function that maps boolean guards to the truth-values `false` and `true` (such a function is often referred to as a *valuation* function) and let $[\![\phi]\!]_v$ denote the function value of the EQ-BDD ϕ after applying v to its guards. Then, using this notation, true -and false-guards can be defined more formally as follows:

Definition 3.1.1 (true-guard)**.** A guard g of an EQ-BDD ϕ is a true-guard of ϕ if, and only if, for every valuation v the following holds:

$$[\![\phi]\!]_v = \mathtt{true} \Rightarrow v(g) = \mathtt{true}.$$

Definition 3.1.2 (false-guard)**.** A guard g of an EQ-BDD ϕ is a false-guard of ϕ if, and only if, for every valuation v the following holds:

$$[\![\phi]\!]_v = \mathtt{true} \Rightarrow v(g) = \mathtt{false}.$$

Example 3.1.3. Let ϕ be the EQ-BDD representing the boolean function $(p \vee q) \wedge r$ (see Figure 2.1(a)). Then there are three possible valuation functions v_i, $0 \leq i \leq 2$, for which $[\![\phi]\!]_{v_i} = \mathtt{true}$, namely: $v_0 = \{p \rightarrow \mathtt{false}, q \rightarrow \mathtt{true}, r \rightarrow \mathtt{true}\}$, $v_1 = \{p \rightarrow \mathtt{true}, q \rightarrow \mathtt{false}, r \rightarrow \mathtt{true}\}$ and $v_2 = \{p \rightarrow \mathtt{true}, q \rightarrow \mathtt{true}, r \rightarrow \mathtt{true}\}$. It can be easily verified that the only true-guard is r and that there are no false-guards in this case.

When the EQ-BDD under consideration is ordered and reduced the previous definitions result in some nice properties of its true -and false-guards. These properties have to do with the positioning of the guard in the EQ-BDD.

Let ϕ be a reduced and ordered EQ-BDD, and let g be a true-guard occurring in ϕ. Then the following properties hold for g:

1. g occurs on every path from the root of ϕ to the leaf labelled with `true`, and
2. the low-branch of every sub-BDD rooted at an occurrence of g (in ϕ) is directly connected to the leaf labelled with `false`.

Similarly, the following properties hold for a false-guard g' of ϕ:

1. g' occurs on every path from the root of ϕ to the leaf labelled with `true`, and
2. the high-branch of every sub-BDD rooted at an occurrence of g' (in ϕ) is directly connected to the leaf labelled with `false`.

The first property is exactly the same for both types of guards and is a straightforward one: if a guard g does not occur on every path from the root of ϕ to `true`, then there exist valuation functions υ and τ such that $[\![\phi]\!]_\upsilon = \texttt{true}$ but not $\upsilon(g) = \texttt{true}$, and $[\![\phi]\!]_\tau = \texttt{true}$ but not $\tau(g) = \texttt{false}$. Therefore g cannot be a true-guard or a false-guard of ϕ if it does not occur on every path from the root of ϕ to `true`.

The second requirement is similar for both types of guards and is also straightforward. Let ψ be a sub-BDD of ϕ rooted at an occurrence of the guard g. Suppose that there is a path from the root of ϕ to `true` which includes the low-branch of ψ. Then g cannot be a true-guard of ϕ since taking the low-branch of ψ implies that there exists a valuation function υ such that $[\![\phi]\!]_\upsilon = \texttt{true}$ but not $\upsilon(g) = \texttt{true}$. Because of this the low-branch of ψ must directly be connected to the leaf labelled by `false`, if g is a true-guard of ϕ (ϕ is reduced and ordered). Similarly for false-guards, but with the high-branch of ψ instead of the low-branch.

3.2 First approach

In this section a simple, but unfortunately not too efficient, algorithm for finding the true -and false-guards will be derived. This algorithm starts at the root of the BDD and recursively visits each node in the BDD once. It uses the properties of true -and false-guards to determine whether a node is a true -or a false-variable.

3.2.1 Observations

Based on the properties given in the previous section a number of observations can be made about when a guard is a true-guard or a false-guard of a reduced and ordered EQ-BDD. Let ϕ be a EQ-BDD and g a guard occurring in ϕ. Then one of the following cases must hold if g is a true-guard of ϕ:

- $root(\phi) = g$ and $low(\phi) = \texttt{False}$, or
- g is a true-guard of $high(\phi)$ and $low(\phi) = \texttt{False}$, or
- g is a true-guard of $low(\phi)$ and $high(\phi) = \texttt{False}$, or
- g is a true-guard of $high(\phi)$ and g is a true-guard of $low(\phi)$.

Similarly, if g is a false-guard of ϕ one of the following cases must hold:

- $root(\phi) = g$ and $high(\phi) = \texttt{False}$, or
- g is a false-guard of $high(\phi)$ and $low(\phi) = \texttt{False}$, or
- g is a false-guard of $low(\phi)$ and $high(\phi) = \texttt{False}$, or
- g is a false-guard of $high(\phi)$ and g is a false-guard of $low(\phi)$.

The algorithm resulting from these observations is given in the next subsection.

3.2.2 Algorithm

The pseudocode for the algorithm by which all true -and false-variables of an EQ-BDD can be found is presented in Figure 3.1. If $g \in \mathsf{TFGuards}(\phi)$ then g is a true-guard of ϕ. Similarly, if $\neg g \in \mathsf{TFGuards}(\phi)$ then g is a false-guard of ϕ. For efficiency reasons, the usage of a hash-table to store intermediate results is included.

```
TFGuards(φ): Set =
1   begin
2       "initialise hash-table";
3       result := TFGuards-Step(φ);
4       "destroy hash-table";
5       return result;
6   end

TFGuards-Step(φ): Set =
1   begin
2       if (φ = False ∨ φ = True) then return ∅;
3       result := "get entry for φ from the hash-table";
4       if (result ≠ Null) then return result;
5       if (high(φ) = False) then result := {¬root(φ)} ∪ TFGuards-Step(low(φ));
6       else if (low(φ) = False) then result := {root(φ)} ∪ TFGuards-Step(high(φ));
7       else if (high(φ) ≠ True ∧ low(φ) ≠ True) then
8           result := TFGuards-Step(high(φ)) ∩ TFGuards-Step(low(φ));
9       else return ∅;
10      "create an entry for φ with value 'result' in hash-table";
11      return result;
12  end
```

Figure 3.1: The algorithm for finding true -and false-guards.

What is the running time of TFGuards? The function TFGuards takes $\mathcal{O}(1)$ time, not counting the time it takes to execute the calls to TFGuards-Step. Lines 5-8 of the function TFGuards-Step are executed exactly once for each node in the EQ-BDD, due to the use of the hash-table. The cost of executing these lines is $\mathcal{O}(|\phi|)$, due to the intersection operation of line 8. Therefore the total running time of TFGuards is $\mathcal{O}(|\phi|^2)$.

In the next section a second algorithm will be derived which is more efficient but has not been implemented.

3.3 Graph-based approach

In this section a second algorithm for finding the true -and false-guards of a (reduced and ordered) EQ-BDD will be derived. This algorithm makes use of a graph algorithm to find the true -and false-guards. For this it is assumed that the reader is familiar with the basics of graph theory. If not, the reader is referred to Appendix A.

3.3.1 Finding possible true -and false-guards

Because of the nature of the first property of true -and false-guards (see Section 3.1), it seems like the problem can (at least partially) be solved by a suitable graph algorithm.

Consider a connected directed acyclic graph $G = (V, A)$ with the following properties:

- there exists a vertex $s \in V$ with indegree 0 and outdegree > 0;

- there exists a vertex $t \in V$ with outdegree 0 and indegree > 0;
- all nodes $v \in V \backslash \{s, t\}$, have an indegree > 0 and an outdegree > 0.

Example 3.3.1. In Figure 3.2 a connected directed acyclic graph with the properties listed above is depicted. Here $V = \{s, a, b, c, d, t\}$ and $A = \{(s, a), (s, b), (a, b), (b, c), (b, d), (c, d), (d, t)\}$.

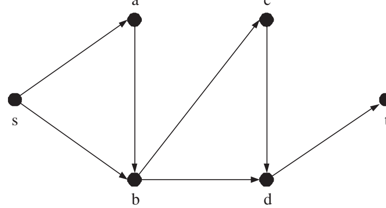

Figure 3.2: A connected directed acyclic graph.

Lemma 3.3.2. *If $G = (V, A)$ is a connected directed acyclic graph with the properties listed above, then a node $v \in V \backslash \{s, t\}$ is on every path from s to t if, and only if, the removal of v results in G being disconnected.*

Proof. ($\Rightarrow$): Suppose the node v occurs on every path from s to t. Then if v is removed the vertex t will no longer be reachable from s. Hence, G has become disconnected. ($\Leftarrow$): This will be shown by contradiction. Suppose v is a node whose removal results in G becoming disconnected, but v does not occur on every path from s to t. Let p be an arbitrary path from s to t going through v. If v is removed, it's predecessor on the path p is still connected to the part containing s: either it is s or there is a path from s to it. Similarly, the successor of v on p is still connected to the part containing t: either it is t or there is a path to t from it. Because there also exists a path $p' \neq p$ from s to t, the parts containing s and t also remain connected to each other. Contradiction: the removal of the node v does not result in G becoming disconnected. Hence, v must occur on every path from s to t if its removal results in G becoming disconnected.

□

The vertices whose removal results in the graph becoming disconnected are often referred to as *articulation points* (or cut-vertices) (see Section A.4).

Example 3.3.3. The graph in Figure 3.2 contains two articulation points, namely b and d. It can be easily verified that these vertices are also the only two vertices occurring on every path from s to t.

Using all of this, the guards that occur on every path from the root of a EQ-BDD to the leaf labelled with `true` can be found by first constructing a connected directed acyclic graph from the EQ-BDD with the properties listed above and then finding the articulation points of this graph. The graph can be easily created from the EQ-BDD by removing the leaf labelled with `false` (including all arcs to it) and by merging all nodes with the same label. All guards occuring on every path from the root to `true` (other than the root itself) can then be found with a simple *depth-first search* (DFS) algorithm, which is given in Section A.4. This algorithm uses the tree structure provided by the depth-first search to find the articulation points.

3.3.2 Filtering out the true -and false-guards

When the guards occuring on every path from the root of the BDD to `true` have been found, the guards that do not satisfy the second property of true -and false-guards (see Section 3.1) need to be filtered out. To find the guards for which the second property of true -or false-variables holds, it suffices to traverse the BDD once. During this traversal all guards for which the second property

does not hold can be marked. This can even be done during the creation of the connected directed acyclic graph.

In the next section the complete algorithm for finding the true -and false-guards of an EQ-BDD using depth-first search will be presented. This algorithm has not been implemented.

3.3.3 Algorithm

The pseudocode for the algorithm by which all true -and false-variables of an EQ-BDD can be found using a DFS algorithm is presented in Figure 3.3.

```
TFGuards(φ) =
1   begin
2     G := "create graph from φ";
3     for each v ∈ G.V do
4       visited[v] := false;
5       pred[v] := NULL;
6     time := 0;
7     guards := ∅;
8     for each v ∈ G.V do
9       if (visited[v] = false) then TFGuards-Visit(v);
10  end

TFGuardsVisit(v) =
1   begin
2     visited[v] := true;
3     time := time + 1;
4     Low[v] := d[v] := time;
5     for each u ∈ Adj(v) do
6       if (visited[v] = false) then
7         pred[u] := v;
8         TFGuards-Visit(u);
9         Low[v] := min(Low[v], Low[u]);
10        if (pred[v] = NULL)
11          if ("this is v's second child") then
12            if ("the second property of a true-guard holds for v") then
13              "add v to set of true-guards";
14            if ("the second property of a false-guard holds for v") then
15              "add v to set of false-guards";
16        else if (Low[u] ≥ d[v]) then
17          if ("the second property of a true-guard holds for v") then
18            "add v to set of true-guards";
19          if ("the second property of a false-guard holds for v") then
20            "add v to set of false-guards";
21      else if (u ≠ pred[v]) then Low[v] = min(Low[v], d[u]);
22  end
```

Figure 3.3: The algorithm for finding true -and false-guards using a DFS algorithm.

The running time of the TFGuards algorithm is $\mathcal{O}(|\phi| + |arcs(\phi)|)$. Creating the graph (and marking the guards for which the second property of true -and false-guards does not hold) can be done in $\mathcal{O}(|\phi|)$ time, while the rest of the algorithm has the same running time as every other DFS algorithm. In this case $\mathcal{O}(|\phi| + |arcs(\phi)|)$, since the graph has less arcs than the original BDD. Combining this results in a total running time of $\mathcal{O}(|\phi| + |arcs(\phi)|)$.

3.4 Application example

Finding the true -and false-guards can be useful for the tool sumelm of the μCRL toolset. This tool simplifies an LPO by replacing sum variables that must be equal to a certain data term with that data term. Finding such sum variables can be done by determining the true-guards of the BDD representing the condition of the LPO. For example, if x is a sum variable of an LPO, and the guard $x = 5$ is a true-guard of the condition of this LPO, then all occurrences of x in the sum can be replaced with 5. In contrast to true-guards, the false-guards of a condition are less useful. They only yield concrete information if the sort of the involved sum variable has only two possible values (booleans or bits, for example). If, for example, $x = 5$ is a false-guard this information can not be used to eliminate sum variable x since there remain a lot of possible values for x.

Chapter 4

Simplification operators

In this chapter two operators for simplifying an EQ-BDD given another EQ-BDD will be given. To be more precise, if it is known that the underlying function of a certain EQ-BDD evaluates to `true`, then it may be possible to use this knowledge to simplify another EQ-BDD by removing certain paths from it.

Example 4.0.1. Consider the boolean function $(x = y \vee y = z) \wedge p$. If it is known that the boolean function $x = y \wedge q$ (and therefore also its EQ-BDD) evaluates to `true`, then the function $(x = y \vee y = z) \wedge p$ can be simplified to p since it is known that $x = y$ holds.

In Section 4.1 some desired properties of a simplification operator for EQ-BDDs will be given. In Section 4.2 and in Section 4.3 two simplification operators will be defined. For each of these operators first a set of rules will be given which define the behaviour of the operator. This behaviour will then be verified against the desired properties: for each operator it will be checked which of the desired properties hold and which do not. Finally for each operator the resulting algorithm will be given. The chapter concludes by giving an application example of the operators (Section 4.4).

Note: since EQ-BDDs are an extension of BDDs, both operators defined in this chapter are also applicable to BDDs containing boolean variables only. In fact, the operators defined in this chapter are based on existing operators for this kind of BDDs.

4.1 Desired properties

Let ϕ and ψ be reduced and ordered EQ-BDDs. It is desired that the following properties hold for a simplification operator $\upharpoonright$ (where $\phi \upharpoonright \psi$ denotes that ϕ is going to be simplified using ψ):

i. the size of the EQ-BDD resulting from $\phi \upharpoonright \psi$ should be less than or equal to the size of ϕ;

ii. the EQ-BDD resulting from $\phi \upharpoonright \psi$ should be logically equivalent to ϕ, under the assumption that ψ holds;

iii. the EQ-BDD resulting from $\phi \upharpoonright \psi$ should only contain guards that are also in ϕ;

iv. the EQ-BDD resulting from $\phi \upharpoonright \psi$ should not contain a path that is inconsistent with ψ.

In the next two sections two simplification operators will be defined. These operators are based on two existing operators often referred to as the Restrict [12] and Constrain [11] operators. These operators have originally been defined on a refinement of BDDs called *Typed Decision Graphs* (TDGs) [7]. Here the rules for these operators will be extended by adding additional rules for usage with EQ-BDDs. Although the difference in the rules for the Restrict and Constrain operators is subtle, their application can result in different EQ-BDDs. Examples of this will be given in the next two sections.

4.2 The Restrict operator

The first simplification operator that will be defined is the Restrict operator. This operator, denoted by the symbol $\Downarrow$ in an infix notation, takes as input two reduced and ordered EQ-BDDs ϕ and ψ and tries to simplify ϕ by restricting it to the domain covered by ψ. This is done by restricting guards in ϕ to the constant `true` if they are included in their positive form in ψ, and `false` if they are included in their negative form.

In Section 4.2.1 the rules for the Restrict operator for EQ-BDDs are given. In Section 4.2.2 these rules are verified against the desired properties given in the previous section. Finally, in Section 4.2.3 an algorithm that is suitable for implementing the Restrict operator will be given.

4.2.1 Rules

Let $\prec$ be a total order on guards. Then the Restrict operator is recursively defined by the rules listed in Figure 4.1. These rules are based on case distinction on ϕ and ψ.

1. if $\psi = \texttt{False}$ then $\phi \Downarrow \psi = \texttt{False}$;
2. if $\phi \in \{\texttt{True}, \texttt{False}\}$ or $\psi = \texttt{True}$ then $\phi \Downarrow \psi = \phi$;
3. if $root(\phi) \prec root(\psi)$ then $\phi \Downarrow \psi = ITE(root(\phi), high(\phi) \Downarrow \psi, low(\phi) \Downarrow \psi)$;
4. if $root(\phi) \succ root(\psi)$ then $\phi \Downarrow \psi = \phi \Downarrow (high(\psi) \vee low(\psi))$;
5. if $root(\phi) = root(\psi)$ and $high(\psi) = \texttt{False}$ then $\phi \Downarrow \psi = low(\phi) \Downarrow low(\psi)$;
6. if $root(\phi) = root(\psi)$ and $low(\psi) = \texttt{False}$ then $\phi \Downarrow \psi = high(\phi) \Downarrow high(\psi)$;
7. if $root(\phi) = root(\psi)$ and $high(\psi) \neq \texttt{False}$ and $low(\psi) \neq \texttt{False}$ then $\phi \Downarrow \psi = ITE(root(\phi), high(\phi) \Downarrow high(\psi), low(\phi) \Downarrow low(\psi))$;
8. if $root(\psi)$ contains an equality of the form $y = x$ and $low(\psi) = \texttt{False}$ then $\phi \Downarrow \psi = \phi[y := x] \Downarrow high(\psi)$.

Figure 4.1: The rules for the Restrict operator.

The connective $\vee$ in rule 4 is used to denote the logical OR operation defined on BDDs by Bryant's Apply function (see 2.1). The rule that was added for EQ-BDDs, and which distinguishes the operator defined here from the operator defined by Coudert et al., is rule 8. This rule allows to substitute equals for equals. This is illustrated by the following example.

Example 4.2.1. Let $\prec$ be a total order on variables such that $x \prec y \prec z$. Let ϕ be the EQ-BDD representing the boolean formula $y = x \wedge z = x$ (see Figure 4.2(a)), and let ψ be the EQ-BDD representing the boolean formula $z = y$ (see Figure 4.2(b)). Then the result of $\phi \Downarrow \psi$ is the EQ-BDD representing the boolean formula $y = x$ (see Figure 4.2(c)). This is because the variable z in ϕ is substituted by the smaller variable y (and $(y = x \wedge y = x) \equiv y = x$).

In the next subsection the rules for the Restrict will be verified against the desired properties of Section 4.1.

4.2.2 Verification

For each of the properties it can either be proven that the rules defined in the previous subsection satisfy this property, or a counterexample can be given to show that they do not. For some of these properties it is known that they hold (or do not hold) for the Restrict operator defined on ordinary BDDs. However, at the time of writing the author was not aware of a full formal proof of any of these properties.

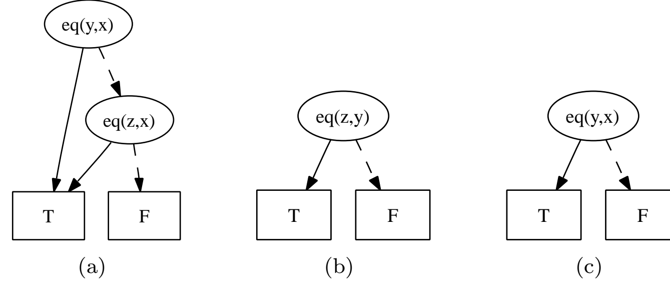

Figure 4.2: Example where variables are substituted when applying the Restrict operator.

Property i:

Since this property does not hold for the Restrict operator defined on BDDs, it automatically does not hold for the operator defined on EQ-BDDs, since EQ-BDDs are an extension of BDDs. This can be illustrated by the following counterexample:

> Let $\prec$ be a total order on variables such that $p \prec q \prec r \prec s$, let ϕ be the reduced and ordered EQ-BDD representing the boolean formula $((p \wedge q) \vee \neg p) \wedge r \wedge s$ (see Figure 4.3(a)), and let ψ be the reduced and ordered EQ-BDD representing the boolean formula $q \Rightarrow s$ (see Figure 4.3(b)). Then the EQ-BDD resulting from $\phi \Downarrow \psi$, depicted in Figure 4.3(c), has more nodes than ϕ. This is due to the unsharing that takes place. Still, the EQ-BDD of (c) might be preferred over (a) because it contains less paths from the root to a leaf, and its paths are consistent with ψ (see property iv).

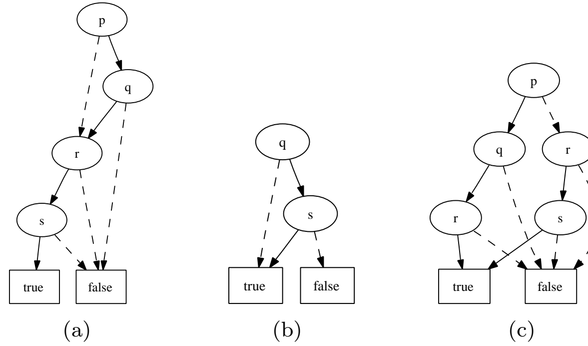

Figure 4.3: Example where applying the Restrict operator results in a larger (EQ-)BDD.

Property ii:

As is the case with the Restrict operator defined on BDDs, it can be proven via induction that this property holds for the extended operator on EQ-BDDs. This proof is given in Section B.1.1.

Property iii:

Although this property holds for the Restrict operator defined on BDDs (every time a BDD is created using the ITE operator a guard of ϕ is chosen as its root), this property does not hold for the extended operator. This can be illustrated by the following counterexample:

> Let $\prec$ be a total order on variables such that $x \prec y \prec z$, let ϕ be the EQ-BDD representing the formula $z = x$, and let ψ be the EQ-BDD representing the formula $z = y$. Then the result of $\phi \Downarrow \psi$ is the EQ-BDD representing the formula $y = x$. Hence, $guards(\phi \Downarrow \psi) = \{y = x\} \nsubseteq \{z = x\} = guards(\phi)$.

Property iv:

Like property ii, it can be proven using induction that this property holds for the extended Restrict operator (and therefore also for the Restrict operator defined on BDDs). This proof is given in Section B.1.2.

In the next subsection an algorithm suitable for implementing the Restrict operator will be given.

4.2.3 Algorithm

The pseudocode for the Restrict algorithm is given in Figure 4.4. This algorithm is directly based on the rules given in Section 4.2.1, but for efficiency reasons the use of a hash-table to store intermediate results is included here.

```
Restrict(φ, ψ): EQ-BDD =
1   begin
2       "initialise hash-table";
3       ω := Rest-Step(φ, ψ);
4       "destroy hash-table";
5       return ω;
6   end;

Rest-Step(φ, ψ): EQ-BDD =
1   begin
2       if (ψ = False) then return False;
3       if (φ = False or φ = True or ψ = True) then return φ;
4       if ("there exists an entry in the hash-table for the pair (φ, ψ)") then
5           return "value for the pair (φ, ψ) in the hash-table";
6       if (IsEQ(root(ψ)) and low(ψ) = False) then
7           ω := Rest-Step(φ[lhs(root(φ)) := rhs(root(ψ))], high(ψ));
8       else if (root(φ) ≺ root(ψ)) then
9           ω := ITE(root(φ), Rest-Step(high(φ), ψ), Rest-Step(low(φ), ψ));
10      else if (root(φ) ≻ root(ψ)) then ω := Rest-Step(φ, (high(ψ) ∨ low(ψ));
11      else if (high(ψ) = False) then ω := Rest-Step(low(φ), low(ψ));
12      else if (low(ψ) = False) then ω := Rest-Step(high(φ), high(ψ));
13      else
14          ω := ITE(root(φ), Rest-Step(high(φ), high(ψ)), Rest-Step(low(φ), low(ψ)));
15      "create a hash-table entry for the pair (φ, ψ) with value ω";
16      return ω;
17  end;
```

Figure 4.4: The algorithm for the Restrict operator.

In the next section a second simplification operator will be defined.

4.3 The Constrain Operator

The second simplification operator that will be defined is the Constrain operator. This operator, denoted by the symbol $\downarrow$ in an infix notation, takes as input two reduced and ordered EQ-BDDs ϕ and ψ and computes the generalized cofactor of ϕ with respect to ψ.

In Section 4.3.1 the rules for the Constrain operator for EQ-BDDs are given. In Section 4.3.2 these rules are verified against the desired properties given in Section 4.1. This section will also give an overview of which properties are satisfied (or not) by the different operators. Finally, in Section 4.3.3 an algorithm that is suitable for implementing the Constrain operator will be given.

4.3.1 Rules

Let $\prec$ be a total order on guards. Then the Constrain operator is recursively defined by the rules listed in Figure 4.5. These rules are based on case distinction on ϕ and ψ.

1. if $\psi = \texttt{False}$ then $\phi \downarrow \psi = \texttt{False}$;
2. if $\phi \in \{\texttt{True}, \texttt{False}\}$ or $\psi = \texttt{True}$ then $\phi \downarrow \psi = \phi$;
3. if $root(\phi) \prec root(\psi)$ then $\phi \downarrow \psi = ITE(root(\phi), high(\phi) \downarrow \psi, low(\phi) \downarrow \psi)$;
4. if $root(\phi) \succ root(\psi)$ then $\phi \downarrow \psi = ITE(root(\psi), \phi \downarrow high(\psi), \phi \downarrow low(\psi))$;
5. if $root(\phi) = root(\psi)$ and $high(\psi) = \texttt{False}$ then $\phi \downarrow \psi = low(\phi) \downarrow low(\psi)$;
6. if $root(\phi) = root(\psi)$ and $low(\psi) = \texttt{False}$ then $\phi \downarrow \psi = high(\phi) \downarrow high(\psi)$;
7. if $root(\phi) = root(\psi)$ and $high(\psi) \neq \texttt{False}$ and $low(\psi) \neq \texttt{False}$ then $\phi \downarrow \psi = ITE(root(\phi), high(\phi) \downarrow high(\psi), low(\phi) \downarrow low(\psi))$;
8. if $root(\psi)$ contains an equality of the form $y = x$ and $low(\psi) = \texttt{False}$ then $\phi \downarrow \psi = \phi[y := x] \downarrow high(\psi)$.

Figure 4.5: The rules for the Constrain operator.

Again the rule that was added for EQ-BDDs, and which distinguishes the operator defined here from the operator defined by Coudert et al., is rule 8. The Constrain operator differs in only one rule from the Restrict operator, namely rule 4. The execution of this rule is in general cheaper than executing rule 4 of the Restrict operator, but the resulting EQ-BDD can be larger. This is illustrated by the counterexample for the first property (Section 4.3.2).

In the next subsection these rules will be verified against the desired properties of Section 4.1.

4.3.2 Verification

For each of the properties it can either be proven that the rules defined in the previous subsection satisfy this property, or a counterexample can be given to show that they do not. For some of these properties it is known that they hold (or do not hold) for the Constrain operator defined on BDDs. However, like with the Restrict operator, at the time of writing the author was not aware of a full formal proof of any of these properties.

Property i:

Like with the Restrict operator, this property does not hold for the Constrain operator defined on BDDs. Therefore, again it automatically does not hold for the operator defined on EQ-BDDs. This can be illustrated by the following counterexample:

> Let $\prec$ be a total order on guards such that $p \prec q \prec r \prec s$, let ϕ be the reduced and ordered EQ-BDD representing the boolean formula $((p \wedge q) \vee \neg p) \wedge r \wedge s$ (see Figure 4.3(a)), and let ψ be the reduced and ordered EQ-BDD representing the boolean formula $q \Rightarrow s$ (see Figure 4.3(b)). Then the EQ-BDD resulting from $\phi \downarrow \psi$, depicted in Figure 4.6, has more nodes than ϕ. This is due to the unsharing that takes place.

The example given here also illustrates the difference between the Restrict and Constrain operator. The result of applying the Constrain operator is even larger than the result of the Restrict operator.

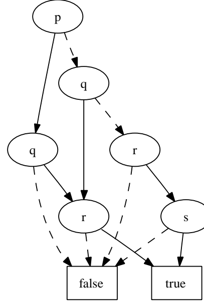

Figure 4.6: Example where applying the Constrain operator results in a larger (EQ-)BDD.

Property ii:

As is the case with the Restrict operator defined in the previous section, it can be proven via induction that this property holds for the Constrain operator. This proof is given in Section B.2.1.

Property iii:

In contrast to the Restrict operator, this property does not hold for the Constrain operator defined on BDDs as well as the extended Constrain operator. For the extended operator the same example as for the extended Restrict operator can be used. That this property does not hold for BDDs can be illustrated by the following counterexample:

> Let $\prec$ be a total order on guards such that $p \prec q \prec r$, let ϕ be the BDD representing the formula $q \wedge r$, and let ψ be the BDD representing the formula $p \vee r$. Then the result of $\phi \downarrow \psi$ is the BDD representing the formula $(p \wedge q \wedge r) \vee q$. Hence, $guards(\phi \downarrow \psi) = \{p, q, r\} \not\subseteq \{q, r\} = guards(\phi)$.

This result is due to the 4th rule for the Constrain operator which adds the root of ψ to the result.

Property iv:

Like property ii, it can be proven using induction that this property holds for the extended Constrain operator (and therefore also for the Constrain operator defined on BDDs). This proof is given in Section B.2.2.

The results of the verification of the operators are summarized in Table 4.1. Here the results are listed for the extended Restrict and Constrain operators, as well as the ones defined on BDDs.

	property i	property ii	property iii	property iv
Restrict (BDDs)	no	yes	yes	yes
Constrain (BDDs)	no	yes	no	yes
Restrict (EQ-BDDs)	no	yes	no	yes
Constrain (EQ-BDDs)	no	yes	no	yes

Table 4.1: A summary of the properties satisfied by the simplification operators.

In the next subsection an algorithm suitable for implementing the Constrain operator will be given.

4.3.3 Algorithm

The pseudocode for the Constrain algorithm is given in Figure 4.7. This algorithm is directly based on the rules given in Section 4.3.1, but for efficiency reasons the use of a hash-table to store intermediate results is included here.

```
Constrain(φ, ψ): EQ-BDD =
1   begin
2       "initialise hash-table";
3       ω := Const-Step(φ, ψ);
4       "destroy hash-table";
5       return ω;
6   end;
```

```
Const-Step(φ, ψ): EQ-BDD =
1   begin
2       if (ψ = False) then return False;
3       if (φ = False or φ = True or ψ = True) then return φ;
4       if ("there exists an entry in the hash-table for the pair (φ, ψ)") then
5           return "value for the pair (φ, ψ) in the hash-table";
6       if (IsEQ(root(ψ)) and low(ψ) = False) then
7           ω := Const-Step(φ[lhs(root(φ)) := rhs(root(ψ))], high(ψ));
8       else if (root(φ) ≺ root(ψ)) then
9           ω := ITE(root(φ), Const-Step(high(φ), ψ), Const-Step(low(φ), ψ));
10      else if (root(φ) ≻ root(ψ)) then
11          ω := ITE(root(ψ), Const-Step(φ, high(ψ)), Const-Step(φ, low(ψ)));
12      else if (high(ψ) = False) then ω := Const-Step(low(φ), low(ψ));
13      else if (low(ψ) = False) then ω := Const-Step(high(φ), high(ψ));
14      else
15          ω := ITE(root(φ), Const-Step(high(φ), high(ψ)), Const-Step(low(φ), low(ψ)));
16      "create a hash-table entry for the pair (φ, ψ) with value ω";
17      return ω;
18  end;
```

Figure 4.7: The algorithm for the Constrain operator.

In the next section an application example of the simplification operators will be given.

4.4 Application example

An example of an application where the simplification operators presented in this chapter can be used is the tool `invelm` of the μCRL toolset. The purpose of this tool is to simplify an LPO by using an invariant to eliminate unreachable summands. The simplification operators can be used to simplify the guard of an LPO by first building an EQ-BDD ϕ for the guard of the LPO and an EQ-BDD ψ for the invariant. Once these EQ-BDDs are built $\phi \Downarrow \psi$ and/or $\phi \downarrow \psi$ can be calculated hopefully resulting in a simpler LPO.

Chapter 5

Simplification using a SMT solver

In the previous chapter several operators for simplifying EQ-BDDs (and BDDs) have been defined. These operators were defined in such a way that they can be implemented whenever a suitable data structure for BDDs is available. This is due to the fact that they only use properties concerning the structure of the BDDs under consideration. In this chapter an approach for simplifying BDDs using a *Satisfiability Modulo Theories* (SMT) solver will be introduced. SMT is the problem of determining the satisfiability of quantifier-free formulas in the context of a logical theory of interest. Like its simpler counterpart, propositional satisfiability (SAT), SMT has applications in circuit design, compiler optimization, software/hardware verification, planning, and scheduling.

Every path (not necessarily a path from the root to a leaf) in a BDD can be viewed as a conjunction of guards or their negation, depending on whether the high-branch or low-branch is part of the path. The satisfiability of such a path, if in the appropriate format, can be checked by a SMT solver. If a path is unsatisfiable, it can be removed from the BDD, hopefully resulting in a smaller one.

In the next section a general approach for using a SMT solver to simplify BDDs will be presented. This approach describes the steps involved in a general manner. After this, a practical example will be presented: it will be shown how EQ-BDDs can be simplified using the SMT solver Tera (Section 5.2). First a 'naive' version will be presented. In this version the algorithm simply makes a call to Tera for all possible paths originating from the root of the EQ-BDD until either the path has become unsatisfiable or a leaf has been reached. This can become very expensive. An improvement to this naive approach is proposed in Section 5.2.2. In this approach information about the unsatisfiable paths is stored. The idea is that this information can be used to reduce the time spent on calls to Tera.

5.1 General framework

In this section a general approach for simplifying a BDD using a SMT solver will be presented. Essentially, the process of removing unsatisfiable paths from a BDD consists of the following four steps:

Step 1: Choose a path in the BDD (conjunction of guards);

Step 2: Translate the constructed path to the required input format for the SMT solver;

Step 3: Check if the translated path is satisfiable using the SMT solver;

Step 4: Interpret the result of the satisfiability check.

To simplify a BDD as much as possible, these steps need to be repeated until no unsatisfiable path remains. In the next subsections these steps will be treated in more detail.

5.1.1 Choosing a path

To be able to remove all unsatisfiable paths in a feasible manner, paths need to be chosen according to some strategy rather than arbitrarily. Since extending an unsatisfiable path does not yield a satisfiable path, the best way to choose paths is to start with as small a path as possible and then recursively extend this path until the path becomes unsatisfiable or the path cannot be extended any further. To cover all unsatisfiable paths of a BDD using this strategy, the first paths to choose are the two paths of length one that originate from the root. Then in each following recursion new paths are chosen by adding either the high-branch or the low-branch of the node reached by a previously chosen path. When a chosen path reaches a leaf or when it appears to be unsatisfiable the process stops for this path (see also Section 5.1.4).

Note: since SMT solvers are focused on a specific logical theory, it is possible that certain guards occurring in the BDD (and therefore certain arcs) are not translatable to that theory. In this case, it is possible to skip these arcs (if a subset of a path is unsatisfiable, then also the whole path is unsatisfiable) or to translate them using some intermediate steps (see the next step).

5.1.2 Translating a path

After a path has been chosen it usually is not in the correct input format for the SMT solver of choice. Therefore the path needs to be translated first. As mentioned in the text concerning the previous step there could be cases where concepts available in the BDD have no counterpart in the logical theory of the SMT solver. Therefore, to be able to make the translation to the target format, one must first determine which concepts from the BDD have a counterpart in the logical theory, and for which concepts the translation is a bit more involved. For example, it might be possible that the SMT solver does not support certain function symbols. In this case it is possible to eliminate these function symbols, at the cost of introducing new variables and congruence constraints (this is called Ackermann's reduction [1]). In essence, subterms like $F(x)$ and $F(y)$ are replaced by fresh variables f_1 and f_2, and the functionality constraint $x = y \Rightarrow f_1 = f_2$ is added. This yields a formula which is satisfiable if, and only if, the original formula is. The constraints can also be omitted, but then the implication only holds in one direction: without the constraints the original formula is satisfiable if the resulting formula is. The other direction does not necessarily have to hold.

5.1.3 Checking a path

When a path has successfully been translated to the correct input format, its satisfiability can be checked. This involves nothing more than submitting the translated path to the SMT solver.

5.1.4 Interpreting the result

Finally, when a path has been checked for satisfiability the result of the check can be interpreted. If the result of the check is *satisfiable* the whole process continues by choosing new paths as described in Section 5.1.1. If on the other hand the result of the check is *unsatisfiable* then there is no need to continue the process with the current path: as mentioned earlier, if a path p of length $n > 0$ is unsatisfiable then a path p' of length $m > n$ which contains p is also unsatisfiable. Therefore, when a path is unsatisfiable, the node v from which the last arc originates can be safely removed and all incoming arcs of v can be redirected to the node reached by v's high-branch (if the last arc of the path is v's low-branch) or to the node reached by v's low-branch (if the last arc of the path is v's high-branch).

Example 5.1.1. Let x and y be natural numbers and let s denote the successor function. In Figure 5.1(a) an EQ-BDD is depicted which contains the path $[s(y) = x, s(x) = y]$. The triangles are used to denote arbitrary (sub-)BDDs. Because the combination $s(x) = y$ and $s(y) = x$ is impossible, the sub-BDD labelled with B_0 can never be reached. Therefore, the node labelled with $s(y) = x$

can be removed and the high-branch of the node labelled with $s(x) = y$ can be redirected to the sub-BDD labelled with B_1 (see Figure 5.1(b)).

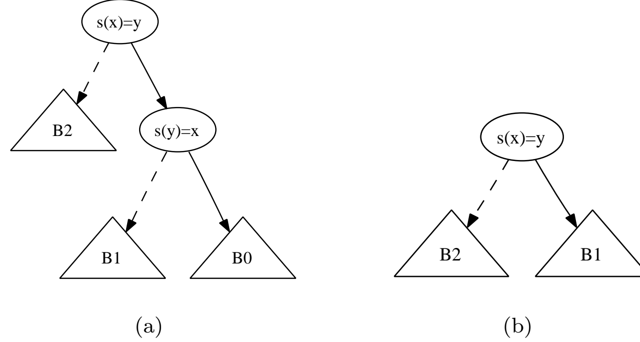

Figure 5.1: An example of removing an unsatisfiable path from a BDD.

In the next section it will be shown how unsatisfiable paths can be removed from EQ-BDDs using the SMT solver Tera.

5.2 Simplifying BDDs with equalities using Tera

In this section a practical example to the general approach presented in the previous section will be given. This example focuses on removing unsatisfiable paths from BDDs that contain equalities and is implemented in C. Here the ATerm library [9] is used to represent the BDDs. This library provides maximal subterm sharing and automatic garbage collection. The SMT solver of choice is Tera (available at `http://www.cwi.nl/~vdpol/gdpll.html`). Tera is a SMT solver for *ground term algebras* (sometimes referred to as abstract data types, or inductive data types). It's purpose is to solve decidability problems in the quantifier-free logic of equality over an infinite ground term algebra. An instance of a formula in this logic would be:

$$(x = S(y) \vee y = S(head(tail(z)))) \wedge z = cons(x, w) \wedge (x = 0 \vee z = nil).$$

The algorithm implemented by Tera can be found in [5]. This algorithm is based on unification theory [2, 23] (to deal with equality) and DPLL [14, 13] (to deal with propositional logic) and is an instance of the *generalized DPLL* (GDPLL) algorithm introduced in [5]. Currently the algorithm implemented by Tera works for constructor symbols (such as zero, successor, nil and cons) only. Extensions to destructors and recognizer predicates (such as nil?, succ?, cons?, zero?) are under development. Besides the ATerm library and the libraries of Tera also the libraries provided by the μCRL toolset [8] have been used.

5.2.1 Naive version

In this section a naive version of an algorithm for removing all unsatisfiable paths from an EQ-BDD will be given. In this version a call is made to Tera for all possible paths originating from the root of the EQ-BDD until either the path has become unsatisfiable or a leaf has been reached.

The main function is called TeraSimp and takes as argument an EQ-BDD ϕ that needs to be simplified. The result of the function will be ϕ with all its unsatisfiable paths removed. The main purpose of TeraSimp is to start the recursive process described in Section 5.1. To this end it performs a call to another function called TeraSimp-Step. This is a recursive function and takes as arguments an EQ-BDD ψ and a path p. Like ϕ that is passed to TeraSimp, ψ is an EQ-BDD that needs to be simplified, but now also the path p by which ψ was reached is passed on. The result of TeraSimp-Step will be the EQ-BDD ψ with all paths that are unsatisfiable, when combined with the path p, removed. The pseudocode for TeraSimp and TeraSimp-Step is given in Figure 5.2.

```
TeraSimp(φ): EQ-BDD =
1   begin
2       return TeraSimp-Step(φ, [ ]);
3   end

TeraSimp-Step(ψ, p): EQ-BDD =
1   begin
2       if (ψ = False or ψ = True) then return ψ;
3       if (¬ IsEQ(root(ψ))) then
4           return ITE(root(ψ), TeraSimp-Step(high(ψ),p), TeraSimp-Step(low(ψ),p));
5       new_lhs := Elim_maps(lhs(root(ψ)));
6       new_rhs := Elim_maps(rhs(root(ψ)));
7       new_root := EQ(new_lhs, new_rhs);
8       p1 := p ∪ new_root;
9       if (UNSAT(p1)) then return TeraSimp-Step(low(φ), p);
10      p2 := p ∪ ¬new_root;
11      if (UNSAT(p2)) then return TeraSimp-Step(high(φ), p);
12      return ITE(root(φ), TeraSimp-Step(high(φ),p1), TeraSimp-Step(low(φ),p2));
13   end
```

Figure 5.2: The TeraSimp and TeraSimp-Step algorithms.

The TeraSimp-Step function makes use of a number of other functions. The functions *IsEQ*, *ITE* and *EQ* are part of the libraries of the μCRL toolset. *IsEQ* is used to check whether a given term is an equality between two terms, *ITE* is used to create a non-constant EQ-BDD given a guard and two EQ-BDDs, and *EQ* is used to create a new equality between two terms.

The check in line 3 is made to make sure that the root of ψ contains an equality. If this is not the case the root is ignored since Tera can only deal with equalities or inequalities. If the root does contain an equality, the lines 5-7 are executed to eliminate all *maps* from the root. Maps are non-constructor functions. The current implementation of Tera cannot handle these. The function Eliminate_maps returns a term with all maps eliminated by applying Ackermann's reduction (without adding the additional constraints, see Section 5.1.2). The pseudocode for this function is given in Figure 5.3.

```
Elim_maps(t): Term =
1   begin
2       if (IsVar(t)) then return t;
3       if (IsMap(t)) then return Map2Var(t);
4       if (IsFunc(t)) then
5           for each a ∈ Arguments(t) do t.a := Elim_maps(a);
6           return t;
7   end
```

Figure 5.3: The Elim_maps algorithm.

Maps are eliminated from a term by performing case distinction on its type. For this the functions *IsVar*, *IsMap* and *IsFunc* are taken from the μCRL toolset and are used to check whether a term is a variable a map (non-constructor) or a function (constructor), respectively. The function Map2Var translates a term which is a map to a variable. The details of this function will not be included here. To translate each occurrence of the same map to the same variable, a hash-table needs to be used.

After eliminating the maps from the root, new paths are created by adding the root in its pos-

itive form to the previously chosen path p, or in its negative form. The satisfiability of these paths is checked by calling the UNSAT function. This function returns `true` if the path is unsatisfiable and `false` if it is satisfiable. The pseudocode for UNSAT is given in Figure 5.4.

```
UNSAT(p): {true, false} =
1  begin
2      cnf := Path2Cnf(p);
3      if (DPLL(cnf) = SAT) then return false
4      return true;
5  end
```

Figure 5.4: The UNSAT algorithm.

The function UNSAT first translates the path p to a boolean formula in *conjunctive normal form* (CNF) which is the input format for Tera. This is done by calling Path2Cnf(p). This function, which will not be treated in detail here, returns the CNF representation of p. In this case each clause of the CNF will contain a single literal, one for each (negated) guard on the path. After the path has been successfully translated to a CNF, the satisfiability of it can be checked by calling the DPLL function. This function returns either SAT or UNSAT and is part of Tera.

If one of the paths created in lines 8 and 10 of the function TeraSimp-Step is unsatisfiable, then a smaller part of ψ can be returned (see 5.1.4). Else, the root will be part of the resulting EQ-BDD and the entire process continues with the sub-BDDs $high(\psi)$ and $low(\psi)$ and the created paths.

This concludes the discussion of the naive version of the algorithm. In the next section an improvement to this version will be proposed.

5.2.2 Improving the naive version

In the TeraSimp algorithm introduced in the previous section Tera is used to check the satisfiability of all possible paths starting in the root of the EQ-BDD until either an unsatisfiable path or a leaf is found. This makes it a very costly algorithm. Reducing the time spent on calls to Tera could result in a more efficient algorithm for larger EQ-BDDs. In this section one method for accomplishing this will be proposed. This method involves storing sets of *inconsistent* guards, e.g. sets of guards for which their conjunction is unsatisfiable. Of course, the set consisting of all guards of an unsatisfiable path is inconsistent, however, since a smaller set of inconsistent guards is more likely to occur on several paths in an EQ-BDD, it is more interesting to find and store a *minimal* set of inconsistent guards for every unsatisfiable path p.

Definition 5.2.1 (Minimal set)**.** A set s of inconsistent guards of a path p is called *minimal* if, and only if, every strict subset s' of s is consistent.

Example 5.2.2. Let p be the (unsatisfiable) path $[v = w, v \neq z, w = y, x = y, x \neq z, y = z]$. Then p contains two minimal sets of inconsistent guards: $s_1 = \{v = w, v \neq z, w = y, y = z\}$ and $s_2 = \{x = y, x \neq z, y = z\}$. These sets are minimal since removing any element would result in a consistent set.

If such minimal sets of inconsistent guards are stored it may be possible to reduce the time spent on calls to Tera by first checking whether a constructed path contains one of these sets. If so, it is not necessary to make a call to Tera. Checking whether a path contains a set of inconsistent guards can be brought down to the following abstract problem:

Problem 5.2.3. *Given a set S of sets and a set P, is there a set $s \in S$ such that $s \subseteq P$?*

In this case the problem must be solved for the instance where S is the set of all minimal sets of inconsistent guards found thus far, and P is the path under consideration. For this a suitable data structure for storing these minimal sets is needed as well as a function for finding a minimal set of inconsistent guards for a given unsatisfiable path, a function for checking whether a path contains a minimal set, and a function for storing a newly found minimal set.

A proposal for a data structure for storing inconsistent sets

For storing the minimal sets of inconsistent guards a datastructure is needed which makes addition of new sets and checking whether a path contains a previously found set easy. Here a BDD-like data structure will be proposed, since a BDD can be easily built and easily traversed. The idea is that, if the internal nodes are labelled by propositions like $contains(x = y)$ and $contains(x \neq y)$, taking a high-branch of a node in the tree corresponds to the (in)equality being part of the path under consideration, and, similarly, taking the low-branch will correspond to the (in)equality not being part of the path. Checking whether a path p contains an earlier found inconsistency can then be done by starting in the root of the tree and subsequently choosing the high-branch or the low-branch (depending on the label) until a leaf is reached. If the leaf labelled with `true` is reached then p contains a previously found set of inconsistent guards. If the leaf labelled with `false` is reached p does not contain a previously found set of inconsistent guards. But it can still contain another set of inconsistent guards which has not been found yet.

Example 5.2.4. Let p be the path $[x = y, y = z, z = Z]$, let p' be the path $[x = y, x \neq z, y = Y, y = z]$ and let $\{x = y, x \neq z, x = z\}$ be the only minimal set of inconsistent guards found so far. The BDD-like tree for this set is depicted in Figure 5.5(a). Path p does not contain this set of inconsistent guards: starting at the root the high-branch is taken since $x = y$ is part of p. Then again the high-branch is taken since p also contains $y = z$. Finally, the low branch is taken since $x \neq z$ is not part of p. Thus, the leaf labelled by `false` is reached which corresponds to p not containing one of the previously found sets of inconsistent guards. Similarly, when looking at p' the leaf labelled with `true` is reached. Hence, p' contains a set of inconsistent guards.

So far only checking whether a certain path contains an earlier found set of inconsistent guards has been discussed. In this discussion the assumption was made that a BDD-like tree containing all previously found sets was already available. How such a tree can actually be built will be discussed next.

Because the tree structure proposed here is very similar to BDDs the idea is to build these trees using Bryant's Apply function (see Section 2.1). For this the order on guards needs to be extended. Let $\prec$ be a total order defined on the domain variables. This order can be extended to guards with propositions concerning equality and inequality as follows:

- $contains(x = y) \prec contains(x' = y')$, if $x \prec x'$ or $x \equiv x'$ and $y \prec y'$;
- $contains(x = y) \prec contains(x' \neq y')$, for all x, y, x' and y';
- $contains(x \neq y) \prec contains(x' \neq y')$, if $x \prec x'$ or $x \equiv x'$ and $y \prec y'$.

Using this ordering on guards, the tree T to store the inconsistent sets in can be built as follows. Initially T will be `False` indicating that no sets of inconsistent guards have been found yet. Then, when a minimal set of inconsistent guards is found for an unsatisfiable path it can be added to T by first building a tree T' for the minimal set and then computing Apply(T, T', $\vee$). The tree for the minimal set can be built by computing the conjunction of all its guards.

Example 5.2.5. Let $u \prec v \prec x \prec y \prec z$. In Figure 5.5 the trees for the minimal sets $\{x = y, x \neq z, y = z\}$ (a) and $\{u = v, u \neq z, v = y, y = z\}$ (b) are depicted, as well as the tree containing both these sets (c).

Next the version of the algorithm which makes use of this data structure will be presented. The main function of this version is simply called TeraSimp2. The TeraSimp2 function serves the same purpose as the TeraSimp function: it is used to start the recursive process. The only difference with the TeraSimp function is that it also initializes the tree that is going to be used to store the sets of inconsistent guards and the hash-table used by the Apply function (and also destroys it when finished). The pseudocode for the TeraSimp2 and TeraSimp2-Step functions is given in Figure 5.6.

The TeraSimp2-Step algorithm is in essence the same as the TeraSimp-Step algorithm. The difference is that when a new path has been constructed a check is made whether this path

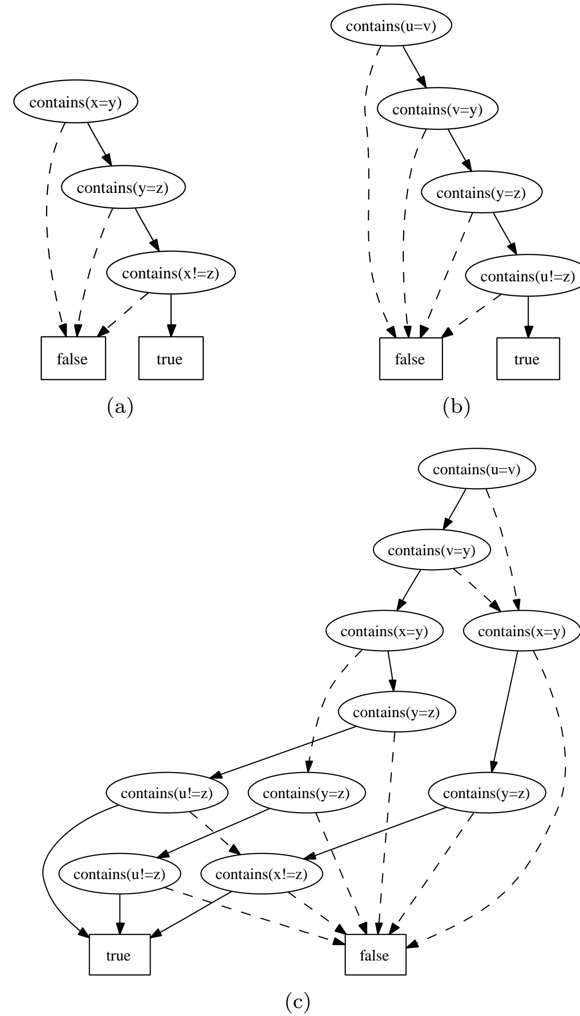

Figure 5.5: Example trees containing sets of inconsistent guards.

contains a previously found set of inconsistent guards (by calling a function Contains_inc). If this indeed is the case the result can be immediately returned without making a call to Tera. If the path does not contain a previously found set of inconsistent guards this does not mean that it does not contain such a set at all. Therefore in this case still a call to Tera must be made. If the path is unsatisfiable then a minimal set of inconsistent guards of the path is determined (Minimal_set) and stored in the tree (Add_inc).

Pseudocode for the Contains_inc function is given in Figure 5.7. As mentioned before, for checking whether a path contains a previously found set of inconsistent guards, the tree used for storing these sets is traversed. By starting at the root of the tree and for each node encountered taking the high-branch or the low-branch (depending on whether the guard with which the node is labelled is part of the path or not) eventually a leaf is reached. If this leaf is labelled with `true` then the path contains a previously found set of inconsistent guards else it does not.

The pseudocode for the Minimal_set function is given in Figure 5.8. When a path p is unsatisfiable a minimal set of inconsistent guards of p needs to be stored. To this end, first this set of guards needs to be found. This is done by the Minimal_set function which finds a minimal set of inconsistent guards for a given unsatisfiable path p. To find this set, two other sets are managed: a set with all remaining candidate guards (l) and a set with all guards for which it is known that they are in a minimal set of inconsistent guards being constructed (s). As long as the conjunction

```
TeraSimp2(φ): BDD =
1    begin
2        "initialize the hash-table used by the Apply function";
3        ic_tree := False;
4        result := TeraSimp2-Step(φ, [ ]);
5        "destroy the hash-table used by the Apply function";
6        return result;
7    end

TeraSimpStep2(ψ, p): BDD =
1    begin
2        if (ψ = False or ψ = True) then return ψ;
3        if (¬IsEQ(root(ψ))) then
4            return ITE(root(ψ), TeraSimp2-Step(high(ψ), p), TeraSimp2-Step(low(ψ), p));
5        new_lhs := Elim_maps(lhs(root(ψ)));
6        new_rhs := Elim_maps(rhs(root(ψ)));
7        new_root := EQ(new_lhs, new_rhs);
8        p1 := p ∪ new_root;
9        if (Contains_inc(p1)) then return TeraSimpStep2(low(ψ), p);
10       p2 := p ∪ ¬new_root;
11       if (Contains_inc(p2)) then return TeraSimpStep2(high(ψ), p);
10       if (UNSAT(p1)) then
11           s := Minimal_set(p1);
12           ic_tree := Add_inc(ic_tree, s);
13           return TeraSimpStep2(low(ψ), p);
14       if (UNSAT(p2)) then begin
15           s := Minimal_set(p2);
16           ic_tree := Add_inc(ic_tree, s);
17           return TeraSimpStep2(high(ψ), p);
18       return ITE(root(ψ), TeraSimpStep2(high(ψ), p1), TeraSimpStep2(low(ψ), p2));
19   end
```

Figure 5.6: The TeraSimp2 and TeraSimp2-Step algorithms.

of guards in s is still satisfiable a guard is chosen from l (Choose_guard). This can be done by simply choosing the first element of l in each iteration, which should result in a minimal set of inconsistent guards. However, because of the fact that the guards in the resulting minimal set are somehow related it seems that there are better ways to choose a guard. Here a guard is chosen if it shares at least one variable with one of the guards already in s. The pseudocode for the Choose_guard function is given in Figure 5.9.

When a guard is chosen from the list of candidates, it is checked whether insertion of this guard in s results in s being unsatisfiable. If so, s (with the chosen guard inserted) is a minimal set of inconsistent guards. If not, it could still be the case that the chosen guard is part of the minimal set that is being constructed. To this end the original path p without the chosen guard is tested for satisfiability. If p without the chosen guard is satisfiable, it is known that the chosen guard must be in the minimal set of inconsistent guards that is being constructed. Thus, the guard is inserted in s and removed from l to avoid being picked again in another iteration. If p without the chosen guard remains unsatisfiable it is known that he guard is not part of the minimal set of inconsistent guards that is constructed. Because of this it can be safely removed from p and also in this case it is removed from l. Since p is unsatisfiable, eventually the combination of guards in s will always be unsatisfiable.

The pseudocode for the Add_inc function is given in Figure 5.10. When a set of inconsistent

```
Contains_inc(p): {true,false} =
1   begin
2       f := ic_tree;
3       while (¬(f = False) and ¬(f = True)) do
4           if (τ(root(f)) ∈ p) then f := high(f);
5           else f := low(f);
6       if (f = False) then return false;
7       if (f = True) then return true;
8   end
```

Figure 5.7: The Contains_inc algorithm.

```
MinSet(p): Set =
1   begin
2       l := Prefix(p);
3       s := Last(p);
4       while (true) do
5           {P0: p = l ∪ s, P1: l ∩ s = ∅, P2: p is UNSAT, P3: l is SAT, P4: s is SAT}
6           g := Choose_guard(l, s);
7           if (UNSAT(Insert(s, g))) then return Insert(s, g);
8           if (¬UNSAT(Remove(p, g))) then
9               s := Insert(s, g);
10              l := Remove(l, g);
12          else
13              p := Remove(p, g);
14              l := Remove(l, g);
17   end
```

Figure 5.8: The Minimal_set algorithm.

guards has been found it can be quite easily added to the tree of previously found inconsistent sets. First a tree for the new set of inconsistent guards has to be made. This is done by starting with a tree $f =$ `True` and subsequently applying the boolean connective AND on f and every guard in the set (and assigning the result to f). When the tree for the single set of inconsistent guards has been built it can be added to the already existing tree with previously found inconsistent sets by applying the boolean connective OR on both trees.

5.2.3 Experimental results

In this section some results of experiments with the two algorithms for simplifying BDDs with equalities using Tera will be presented. These experiments have been performed on a number of examples of industrial systems provided by Jaco van de Pol. These examples included a video-on-demand server, a railroad-crossing system and a splice software architecture. All of these

```
Choose_guard(p, s): Term =
1   begin
2       for each g ∈ p do
3           if (Contains_var_from_list(g, s)) then return g;
4   end
```

Figure 5.9: The Choose_guard algorithm.

```
Add_inc(s) =
1  begin
2      f := True;
3      for each g ∈ s do
4          f := Apply(f, ITE(g, True, False), ∧);
5      ic_tree := Apply(ic_tree, f, ∨);
6  end
```

Figure 5.10: The Add_inc algorithm.

examples consisted of fair number of confluence formulae to be checked. A confluence formula expresses that a τ-summand of an LPO commutes with all summands of the LPO. These formulae can be automatically generated using the `confcheck` tool of the μCRL toolset.

Each of the generated formulae has first been checked using the tool `formcheck` of the μCRL toolset. If the result of such a check was “Don’t know”, then both algorithms for removing the unsatisfiable paths were applied to it. It turned out that applying these algorithms solved 240 previously unsolved formulae of a total of around 2500 formulae. Furthermore, the algorithms also reduced the number of nodes of some of the EQ-BDDs of other unsolved formulae. For example, the size of the EQ-BDD of one of the formulae was reduced from 147 nodes to 28 nodes. To illustrate the difference in size, the EQ-BDD with 147 nodes is depicted in Figure 5.11 and the EQ-BDD with 28 nodes is depicted in Figure 5.12. For the sake of clarity the labels have been removed from the EQ-BDD with 147 nodes.

Other relevant results from the experiments are the following:

- There was no significant difference in the running time of the naive version of the algorithm and the one that stores minimal sets of inconsistent guards;
- The average size of the minimal sets of inconsistent guards was two.

5.2.4 Future work

Currently the minimal sets of inconsistent guards are determined after a path has proven to be unsatisfiable by Tera. In the future it might be possible to change Tera such that the minimal sets of inconsistent guards are provided by Tera.

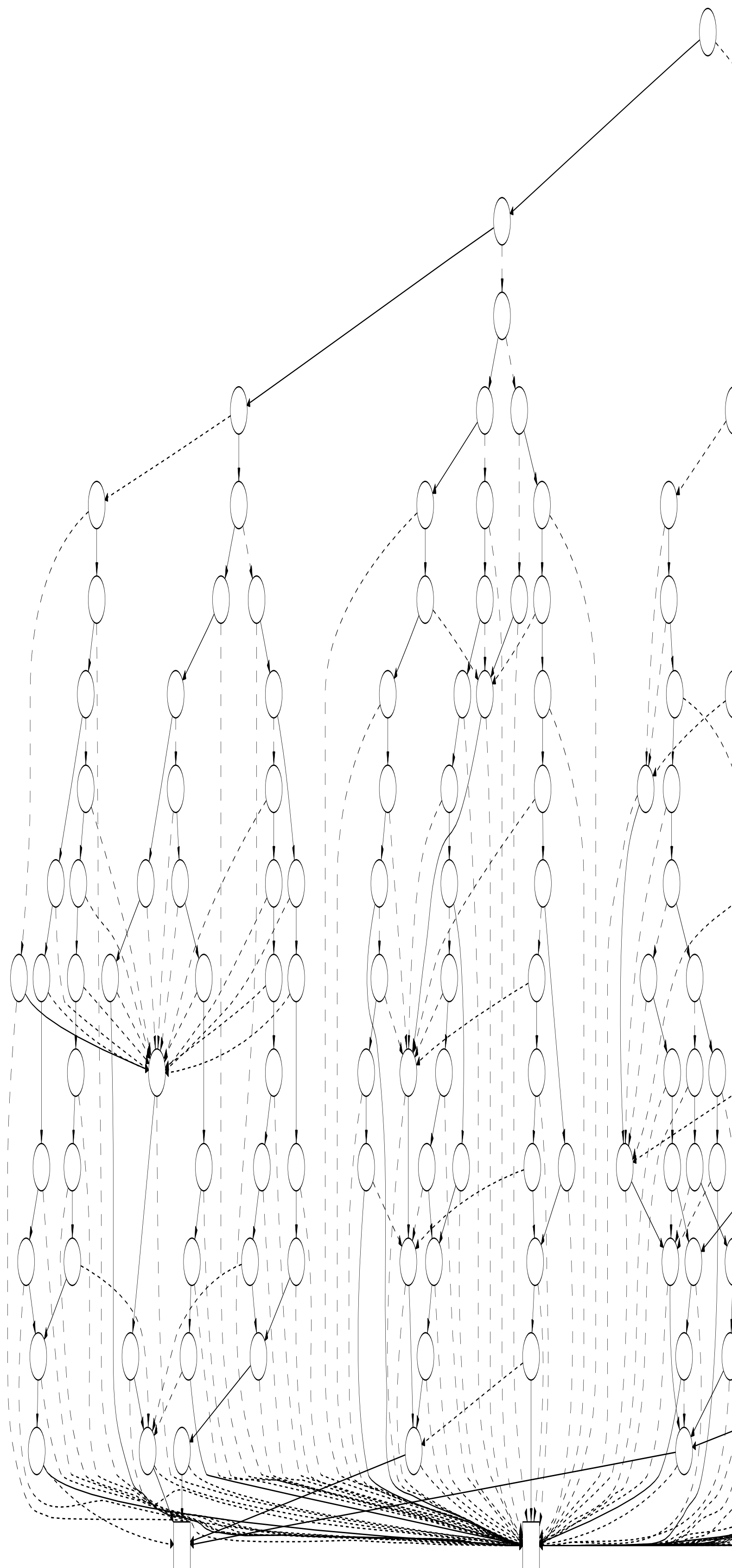

Figure 5.11: A BDD created by `formcheck`.

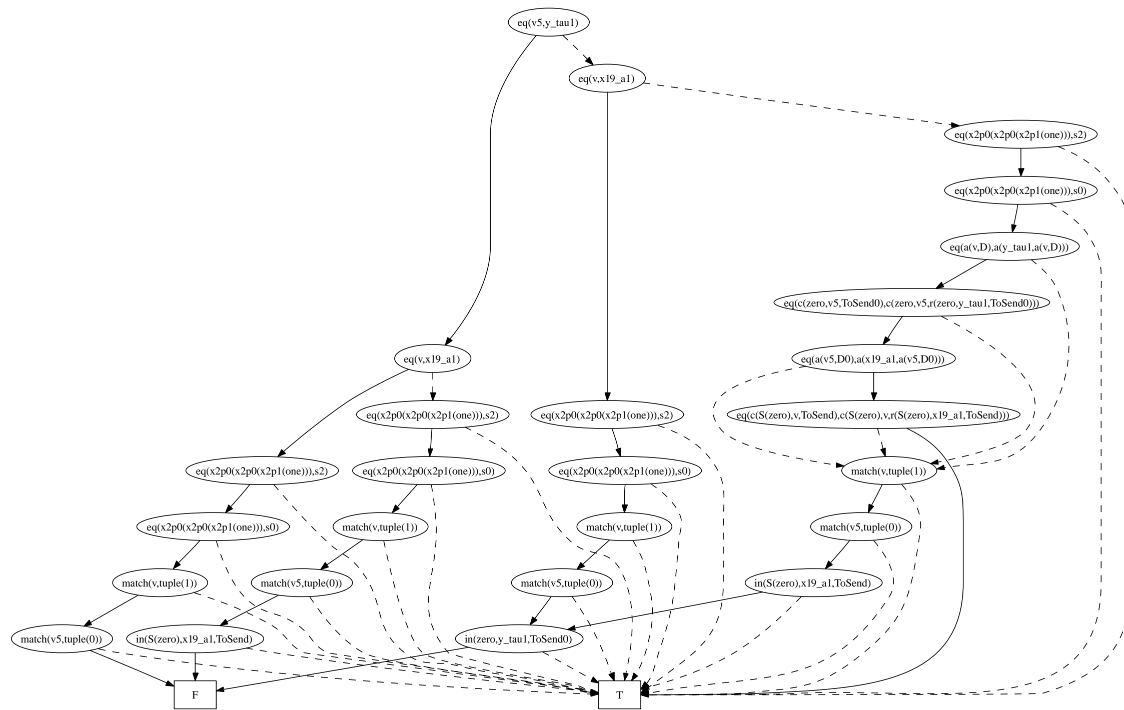

Figure 5.12: The BDD from Figure 5.11 after removing its unsatisfiable paths.

Chapter 6

Prover Shell

In this chapter the tool that was created during the project will be discussed. This shell, called prover shell (or `prover_sh` for short) was created for multiple purposes. The first purpose was to provide a uniform interface through which a user can perform a number of different tasks concerning the μCRL toolset. Currently all tools provide their own interface and need to be used separately. Perhaps in the future a tool like `prover_sh` can be included in the toolset. Besides providing a uniform interface, the shell also provides a nice platform for testing new techniques.

In Section 6.1 the user interface of the shell will be discussed. In this section the commands that are currently available to the user will be listed and two small example scenarios illustrating the usage of these commands will be given. In Section 6.2 the libraries that were created for the shell will be discussed. Besides these newly created libraries the shell also uses other libraries like the libraries of the μCRL toolset and the ATerm library. These will also be discussed in Section 6.2.

6.1 The user interface

Like its name suggests, `prover_sh` provides the user with an environment in which he or she can enter a number of commands using the keyboard and a command line interface. In the next subsection the available commands will be listed categorized by type of interaction. Some of these commands are based on functionality available through the libraries of the μCRL toolset (see also section 6.2). Arguments for the prover, the rewriter and the ATerm library can be passed to the shell itself, these will be automatically passed on when the initialization of either three takes place.

6.1.1 Commands

The commands that are available in the shell can be divided into categories corresponding to their type of interaction. The following categories can be identified:

- Reading and writing from/to the file system;
- Defining formulae for use in the shell;
- Defining EQ-BDDs for use in the shell;
- Querying certain information;
- Other;

For each of these categories the available commands will be listed next. The division given here is not strict: some commands could be listed in multiple categories.

Reading and writing from/to the file system

There are several commands available for reading and writing from/to the file system. These include loading a μCRL data specification to initialize the prover with, and processing and saving scripts. A full list of I/O commands is given in Figure 6.1.

load <file> :
: Loads a μCRL specification from <file> and initializes the prover with the arguments that were passed when starting the shell.

process <file> :
: Processes the commands listed in <file> like they were being entered one by one using the commandline. Each line of <file> should contain exactly one command.

save <file> :
: Saves the list of previously entered commands to <file>. Commands that will not be saved are listdefs, print, picture, save and help.

convert <input> <output> :
: Converts the file <input> generated by a tool from the μCRL toolset (like confcheck) to a script file <output> that can be processed by the shell.

Figure 6.1: List of I/O commands of prover_sh.

Defining formulae

There are several commands available for defining formulae in the shell. First of all there is a command for declaring (μCRL) variables that can be used to define formulae on. When such variables have been declared a formulae can be defined on these variables by either reading it from a file or from the input given by the user. It is also possible to create new formulae by manipulating previously defined ones. A full list of commands for defining formulae is given in Figure 6.2.

var v(<X_0>, <Sort>) :
: Declares the (μCRL) variable <X_0> of type <Sort>. The keyword var may be followed by a list of declarations of the form v(<X_i>, <Sort>) separated by commas.

def <P> = <file> :
: Defines <P> to be the formula that is written between square brackets in <file>.

def <P> = "<form>" :
: Defines <P> to be the formula that is written between double quotes.

def <P> = REWRITE <P_0> :
: If the formula <P_0> is defined, <P> is defined to be the rewritten version of <P_0>, using the strategy with which the rewriter was initialized.

def <P> = INDUCT <P_0> <V> :
: If the formula <P_0> is defined, <P> is defined to be <P_0> with induction performed on the variable <V>.

Figure 6.2: List of commands of prover_sh for defining formulae.

Defining EQ-BDDs

There are several commands for creating an EQ-BDD in the shell. These include commands for building an equivalent EQ-BDD for a previously defined formula and commands for manipulating a previously defined EQ-BDD. A number of commands of this last group is based on the techniques presented in this thesis. A full list of commands for defining EQ-BDDs is given in Figure 6.3.

def <B> = TOPDOWN <P> :
If the formula <P> is defined, <B> is defined to be the ordered EQ-BDD equivalent to <P>. In this case <B> is built using a topdown approach.

def <B> = BOTTOMUP <P> :
If the formula <P> is defined, <B> is defined to be the ordered EQ-BDD equivalent to <P>. In this case <B> is built using a bottomup approach.

def <B> = TFGUARDS <B_0> :
If the EQ-BDD <B_0> is defined, <B> is defined to be the EQ-BDD representing the set of true -and false-guards of <B_0>. See Chapter 3.

def <B> = RESTRICT <B_0> <B_1> :
If the EQ-BDDs <B_0> and <B_1> are defined, <B> is defined to be <B_0> restricted to the domain covered by <B_1>. See Chapter 4.

def <B> = CONSTRAIN <B_0> <B_1> :
If the EQ-BDDs <B_0> and <B_1> are defined, <B> is defined to be the generalized cofactor of <B_0> with respect to <B_1>. See Chapter 4.

def <B> = SIMPTERA <B_0> :
If the EQ-BDD <B_0> is defined, <B> is defined to be <B_0> with all its unsatisfiable paths, that is, paths that have been proven to be unsatisfiable using the SMT solver Tera, removed. See Chapter 5.

def <B> = SIMPTERA2 <B_0> :
If the EQ-BDD <B_0> is defined, <B> is defined to be <B_0> with all its unsatisfiable paths, that is, paths that have been proven to be unsatisfiable using the SMT solver Tera, removed. In this case information about unsatisfiable paths is stored to which hopefully leads to a decrease in the number of calls to Tera. See Chapter 5.

def <B> = SIMPINEQ <B_0> :
If the EQ-BDD <B_0> is defined, <B> is defined to be <B_0> with all its unsatisfiable paths, that is, paths that have been proven to be unsatisfiable using the inequalities in the EQ-BDD, removed.

def <B> = REWRITE <B_0> :
If the EQ-BDD <B_0> is defined, <B> is defined to be the rewritten version of <B_0>, using the strategy with which the rewriter was initialized.

def <B> = INDUCT <B_0> <V> :
If the EQ-BDD <B_0> is defined, <B> is defined to be <B_0> with induction performed on the variable <V>.

Figure 6.3: List of commands of `prover_sh` for defining EQ-BDDs.

Query commands

There are several commands for 'querying' certain information defined in the shell. For example there is a command for displaying all names of the defined formulae and EQ-BDDs and commands for displaying a defined formula or EQ-BDD. A full list of query commands is given in Figure 6.4.

`listdefs` :
: Prints a list of names of the currently defined formulae and BDDs.

`print <X>` :
: If the formula or EQ-BDD `<X>` is defined, it is printed in ASCII format.

`picture <B>` :
: If the EQ-BDD `<B>` is defined, it is displayed using GhostView (GV), if available.

`help` :
: Prints a list of all available commands of the shell with a short description of their purpose.

Figure 6.4: List of commands of `prover_sh` for querying information.

Other commands

Besides all the previously listed commands the shell contains one other command that does not fit in any of the previous categories. This is the `exit` command. When this command is executed, the shell is exited after deleting the temporary files that were created during the session (by the `picture` command).

Next the usage of the shell, and its commands, will be illustrated by means of an example scenario.

6.1.2 An example scenario

In this section a small example of a session with the shell will be given. Although this is only a simple example, and one of many possible scenarios, it serves well as a means of illustrating the use of the shell.

1. Starting the shell.
   The first thing to do, obviously, is to start the shell. In this case, the arguments `-pr-reverse` and `-pr-lpo` are used.

   ```
   > prover_sh -pr-reverse -pr-lpo
   muCRL prover shell, test version, 2005
   --------------------------------------
   muCRL>
   ```

2. Loading a data specification.
   After starting the shell the first thing to do is to load a data specification. In this case, the specification from the file `standard.tbf` is loaded. This file contains a μCRL specification in LPO format and was created using the tool `mcrl` (see Chapter 2). The source file `standard.mcrl` is given in Appendix C. Because no arguments for the rewriter have been used when starting the shell, the rewriter is initialized with the default strategy strictleft.

   ```
   muCRL> load standard.tbf
   prover_sh: JITty Rewriter initialised with hashing (strategy strictleft)
   muCRL>
   ```

3. Declaring some variables.
   Before formulae can be defined, first a number of variables (to be used in the formulae) need to be declared. In this case, five variables will be declared: three of sort `Bool` and two of sort `List`.

```
muCRL> var v(x,Bool),v(y,Bool),v(z,Bool),v(l2,List),v(l1,List)
muCRL>
```

4. Defining some formulae.
   After declaring some variables, it is possible to define some formulae on these variables. In this case, three formulae are defined. Two formulae on the boolean variables, and one formula on the list variables. This last formula also contains an equality.

```
muCRL> def p1 = "and(x,or(y,z))"
muCRL> def p2 = "implies(not(y),not(z))"
muCRL> def q1 = "eq(length(append(l1,l2)),plus(length(l1),length(l2)))"
muCRL>
```

5. Defining some EQ-BDDs.
   Now that there are some formulae defined, it is possible to build EQ-BDDs for these formulae. In this case, two EQ-BDDs are built using a topdown approach. As it turns out, both EQ-BDDs result in "Don't know" messages. This is due to the fact that both formulae are neither tautological nor contradictory.

```
muCRL> def b1 = TOPDOWN p1
p1: Don't know
muCRL> def b2 = TOPDOWN p2
p2: Don't know
muCRL>
```

6. Simplifying an EQ-BDD (and printing the result).
   Because the previously defined EQ-BDDs have some guards in common, it might be possible to simplify one EQ-BDD given that the other one would hold. In this case, assume that it is known that the EQ-BDD for the formula $\neg y \Rightarrow \neg z$ holds. Then it is possible to use this knowledge to simplify the EQ-BDD for the formula $x \wedge (y \vee z)$ using the Restrict operator. The result of this operation is the EQ-BDD for the formula $x \wedge y$.

```
muCRL> def b3 = RESTRICT b1 b2
muCRL> print b3
BDD: (node): atom --> (high) , (low)
------------------------------------
(0): T
(1): F
(2): y --> (0) , (1)
(3): x --> (2) , (1)
------------------------------------
Total number of nodes: 4
muCRL>
```

7. Performing induction on a variable of a formula.
   Recall the previously defined formula assigned to `q1`. This formula expresses that the length of a list obtained by appending two lists together is equal to the sum of the lengths of the individual lists. This formula obviously has to hold for every pair of arbitrary lists `l1` and `l2`. This can be proven by performing induction on `l1`.

```
muCRL> def q2 = INDUCT q1 l1
muCRL> print q2
and(or(not(eq(length(append(l0,l2)),plus(length(l0),length(l2)))),
eq(length(append(cons(d0,l0),l2)),plus(length(cons(d0,l0)),length(l2)))),
eq(length(append(nil,l2)),plus(length(nil),length(l2))))
muCRL> def c1 = TOPDOWN q2
q2: True
muCRL>
```

8. Saving a script.
   After having entered the previous commands, it might be desirable to save a script containing these commands. This makes it possible to repeat them (with the exception of certain tasks) in the future, using the `process` command.

```
muCRL> save scenario.script
muCRL> exit
>
```

This concludes the scenario. In the next section the libraries that were created for the shell, and the most important other libraries that are used by the shell will be discussed.

6.2 The libraries

The shell uses a number of different libraries to incorporate functionality that is not based on interaction with the user, like the actual initialization of the prover, the creation of a BDD, etc. Not all of these libraries were created during the project, for example the ATerm library and the libraries of the μCRL toolset were already available. In the next sections these libraries will be discussed. First the libraries that were created during the project will be treated. After that the libraries that were already available will be treated. This will be done in less detail than with the newly created libraries, only the most important functions that are used by the shell will be mentioned.

6.2.1 Libraries created for the shell

In this section the libraries that were created during the project will be discussed. The following libraries were created:

- `psh_data.h`: provides functionality for creating and manipulating lists of strings;
- `psh_bdd.h`: provides functionality for manipulating and simplifying BDDs;
- `psh_tera.h`: provides functionality for eliminating inconsistent paths from BDDs using the SMT solver Tera.

For each of these libraries the available functions will be listed and described next.

psh_data.h

The `psh_data.h` library contains functionality for creating and manipulating lists of strings. The shell uses such a list of strings to store the names of the defined formulas and BDDs, and to store the list of previously entered commands. A list of strings (or StringList) is defined as an aggregate data structure as follows:

```
typedef struct {
  char **strings;
  int max;
  int count;
  int stepsize;
} StringList;
```

Here `strings` is an array containing the strings in the StringList, `max` is the maximum number of strings that can be stored in the StringList, `count` is the number of strings stored in the StringList, and `stepsize` is the number with which the StringList is expanded when the maximum number of strings is reached.

function: `SLcreate`

Summary: Create a StringList.
Declaration: `StringList * SLcreate(int initial_size, int step_size);`
Description: This function creates a StringList given an initial size and a step size. Whenever the maximum number of strings is exceeded (which is detected when a new string is added using `SLappend`), the StringList is automatically expanded to `max` + `step_size`.

function: `SLdestroy`

Summary: Destroys a StringList.
Declaration: `void SLdestroy(StringList * list);`
Description: When a StringList is no longer needed, the user should release the resources allocated by the StringList by calling `SLdestroy`.

function: `SLgetLength`

Summary: Return the length of a StringList.
Declaration: `int SLgetLength(StringList * list);`

function: `SLappend`

Summary: Return `list` with `s1` appended to it.
Declaration: `StringList * SLappend(StringList * list, char * s1);`

function: `SLindexOf`

Summary: Return the index of a string in a StringList.
Declaration: `int SLindexOf(StringList * list, char * s1);`
Description: Return the index where `s1` can be found in `list`. Returns `-1` if `s1` is not in `list`.

function: `SLelementAt`

Summary: Return a specific string of a StringList.
Declaration: `char * SLelementAt(StringList * list, int index);`

function: `SLremoveElementAt`

Summary: Return `list` with the string at `index` removed.
Declaration: `StringList * SLremoveElementAt(StringList * list, int index);`

function: `SLreplace`

Summary: Return `list` with the string at `index` replaced by `s1`.
Declaration: `StringList * SLreplace(StringList * list, char * s1, int index);`

psh_bdd.h

The `psh_bdd.h` library contains functionality for BDD manipulation and BDD simplification. This includes functionality to make it easier to refer to certain aspects of a BDD (for example its root) and functionality for simplifying BDDs using the Restrict and Constrain operators. It might be desirable, in the future, to move the former functionality to a library of the μCRL toolset, like `signature.h`.

function: `root`

Summary: Return the root element of a BDD.
Declaration: `ATerm root(ATerm b);`
Description: Returns `NULL` if `b` is not a BDD in the if-then-else format.

function: `high`

Summary: Return the BDD located at the high-branch of the root of a BDD.
Declaration: `ATerm high(ATerm b);`
Description: Returns `NULL` if `b` is not a BDD in the if-then-else format.

function: `low`

Summary: Return the BDD located at the low-branch of the root of a BDD.
Declaration: `ATerm low(ATerm b);`
Description: Returns `NULL` if `b` is not a BDD in the if-then-else format.

function: `lhs`

Summary: Return the lefthand side of an equality.
Declaration: `ATerm lhs(ATerm t);`
Description: Returns `NULL` if the term `t` is not an equality.

function: `rhs`

Summary: Return the righthand side of an equality.
Declaration: `ATerm rhs(ATerm t);`
Description: Returns `NULL` if the term `t` is not an equality.

function: `Restrict`

Summary: Return `b1` restricted to the domain covered by `b2`.
Declaration: `ATerm Restrict(ATerm b1, ATerm b2);`
Description: See chapter 4.

function: `Constrain`

Summary: Return the generalized cofactor of `b1` with respect to `b2`.
Declaration: `ATerm Constrain(ATerm b1, ATerm b2);`
Description: See chapter 4.

function: `TFguards`

Summary: Return a BDD representing the set of true -and false-guards of a BDD.
Declaration: `ATerm TFguards(ATerm b);`
Description: The result represents the conjunction of all true -and false-guards. See chapter 3.

psh_tera

The `psh_tera.h` library contains functionality for removing the unsatisfiable paths from a BDD using the SMT solver Tera.

function: `SimpTera`

Summary: Return a BDD with all its unsatisfiable paths removed.
Declaration: `ATerm SimpTera(ATerm b);`
Description: This function uses the SMT solver Tera is for checking whether a path in `b` is satisfiable or not. See chapter 5.

function: `SimpTera2`

Summary: Return a BDD with all its unsatisfiable paths removed.
Declaration: `ATerm SimpTera2(ATerm b);`

Description: This function is in essence the same as the function `SimpTera`. The difference is that in this function sets of inconsistent guards that are found during the process are stored and used to (hopefully) decrease the number of calls to Tera. See chapter 5.

In the next section the other libraries used by the shell will be discussed. These are libraries of the μCRL toolset, the SMT solver Tera and the ATerm library.

6.2.2 Other libraries used

In this section the libraries that are used by the shell and that were not created during the project will be discussed. The following libraries are used:

- `aterm2.h`: provides functionality for creating and manipulating ATerms, lists of ATerms, tables of ATerms, etc.;
- `mcrl.h`: provides functionality for accessing a linearized μCRL specification;
- `rw.h`: provides functionality for a special rewriter which is used in combination with the `mcrl` library;
- `prover.h`: provides functionality for transforming formulas into equivalent EQ-BDDs, simplifying EQ-BDDs using inequalities, and printing EQ-BDDs in ASCII or dot format;
- `signature.h`: provides functionality for representing μCRL data types as ATerms and should be seen as part of the `prover` library;
- `tera.h`: provides functionality for creating and manipulating formulas in CNF format, and checking their satisfiability.

For more information about the ATerm library (`aterm2.h`) the reader is referred to [22]. More information about the libraries of the μCRL toolset (`mcrl.h` and `rw.h`) can be found in [31].

Chapter 7

Conclusions and directions for further work

In this thesis a number of different techniques that can be used in the process of theorem proving and solving satisfiability problems have been presented. Special attention in this has been given to satisfiability problems in the (quantifier-free) logic of equality over abstract data types. The basis of the development of these techniques has been an existing, incomplete, EQ-BDD prover.

The first technique presented in this thesis focused on finding special kinds of guards of an EQ-BDD. These guards have the property that their truth-value is invariant whenever the EQ-BDD evaluates to `true`. It turned out that finding these guards is useful for eliminating sum variables from an LPO, resulting in a simpler LPO.

The second technique presented focused on simplifying an EQ-BDD given an invariant represented as an EQ-BDD. Two different operators with different properties were defined for this. These operators can be used to eliminate summands from an LPO that have become unreachable because of the invariant. Moreover, the guards in other summands can be simplified by this operation.

Finally, the last technique focused on using a SMT solver to remove unsatisfiable paths from an EQ-BDD. For this a general approach has been presented that can be applied for different SMT solvers and different logics. This general approach has been applied in practice. Tera, a solver for abstract data types, has successfully been integrated with the prover. It turned out that, by using Tera to remove unsatisfiable paths from EQ-BDDs created by the prover, more formulae could be solved. The experiments, which were all of moderate size, did not result in a significant difference in running time between the naive version and the version with the proposed improvements. Perhaps in larger examples there are more occurrences of certain inconsistencies which results in a better running time. Promising is the average size of inconsistencies so far, which is two.

In addition to the development of these techniques, a tool has been created during the project. This tool integrates the old prover with the techniques presented in this thesis and adds the possibility of performing induction. It provides a commandline based interface through which the user can, interactively, perform different steps in the process of solving a formula. Furthermore, the tool proved to be quite useful for testing the different techniques.

Next a number of possible directions for further work will be presented.

Conversion of the code to mCRL2

Currently a successor language for μCRL, called mCRL2, is being developed at the Eindhoven University of Technology. Experience obtained from use of μCRL to model and analyse realistic systems showed that the language would benefit from some changes. Because of these changes it is likely that the current implementations of the techniques presented in this paper cannot directly be used in the case of mCRL2.

Extensions for the prover shell

Although the shell turned out to be very useful during the project, there are some things that could be added to it. For example, it could be useful to add the possibility to force a particular ordering on the variables at any time during a session. Or the possibility to re-initialize the prover. Furthermore, the addition of a graphical user interface can be considered.

Adjustments to the Tera SMT solver

With the algorithms presented in this thesis for simplifying EQ-BDDs using the SMT solver Tera the minimal set of inconsistent guards for an unsatisfiable path is computed after the satisfiability check. This seems to be unnecessary, after all, during a satisfiability check for a path Tera is already using such information to come to its conclusion. Therefore, it should be possible to let Tera come up with this information after a path has turned out to be unsatisfiable.

Incorporate the use of more SMT solvers

With the integration of the existing prover with the Tera SMT solver there are currently two methods available for removing unsatisfiable paths from an EQ-BDD. A future extension could be the addition of a SMT solver for the logic of uninterpreted functions (or other logics dealing with equalities between terms).

Bibliography

[1] W. Ackermann. *Solvable Cases of the Decision Problems*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] B. Badban and J. C. van de Pol. An algorithm to verify formulas by means of (0,S,=)-BDDs. In *Proceedings of the 9th Annual Computer Society of Iran Computer Conference (CSICC)*, 2004.

[4] B. Badban and J. C. van de Pol. Zero, sucessor and equality in BDDs. In *Annals of Pure and Applied Logic*, volume 133/1-3, pages 101–123, 2005.

[5] B. Badban, J. C. van de Pol, O. Tveretina, and H. Zantema. Generalizing DPLL and satisfiability for equalities. 2004.

[6] J. C. M. Baeten and W. P. Weijland. Process algebra. In *Cambridge Tracts in Theoretical Computer Science*, volume 18. Cambridge University Press, 1990.

[7] J. P. Billon. Perfect normal forms for discrete functions. *BULL Research Report No.87019*, 1987.

[8] S. C. C. Blom, W. J. Fokkink, J. F. Groote, I. A. van Langevelde, B. Lisser, and J. C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings 13th Conference on Computer Aided Verification (CAV '01)*, LNCS 2102, pages 250–254. Springer-Verlag, 2001.

[9] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software - Practice & Experience 30*, pages 259–291, 2000.

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

[11] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean function vectors. In L. Claesen, editor, *Applied Formal Methods for Correct VLSI Design. Proceedings*, volume 1, pages 111–128. Elsevier, 1989.

[12] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronuous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems. Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, 1989.

[13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM 5*, 7:394–397, 1962.

[14] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery 7*, 3:201–215, 1960.

[15] W. J. Fokkink. Introduction to process algebra. In *Texts in Theoretical Computer Science.* Springer-Verlag, 2000.

[16] J. F. Groote. The syntax and semantics of timed μCRL. Technical Report SEN-R9709. CWI, Amsterdam, The Netherlands, 1997.

[17] J. F. Groote, F. Monin, and J. C. van de Pol. Checking verifications of protocols and distributed systems by computer. In D. Sangiorgi and R. de Simone, editors, *Proceedings 9th Conference on Concurrency Theory (CONCUR'98)*, LNCS 1466, pages 629–655. Springer, 1998.

[18] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. van Vlijmen, editors, *Algebra of Communicating Processes, Workshops in Computing*, pages 26–62. Springer, 1995.

[19] J. F. Groote and J. C. van de Pol. Equational binary decision diagrams. In *Logic Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 2000.

[20] J. F. Groote and J. C. van de Pol. State space reduction using partial τ-confluence. Technical Report SEN-R0008. CWI, Amsterdam, The Netherlands, 2000.

[21] F. Harary. *Graph Theory.* Addison-Wesley, 1969.

[22] H. de Jong and P. Olivier. Aterm library user manual. CWI, Amsterdam, The Netherlands, 2000.

[23] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[24] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[25] J. Loeckx, H.-D. Ehrich, and M. Wolf. Specification of abstract data types. Wiley/Teubner, 1996.

[26] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design, OBDD - Foundations and Applications.* Springer-Verlag, 1998.

[27] Y. S. Usenko. Linearization in μCRL. PhD thesis. Eindhoven University of Technology, 2002.

[28] J. C. van de Pol. A prover for the μCRL toolset with applications - version 0.1. Technical Report SEN-R0106. CWI, Amsterdam, The Netherlands, 2001.

[29] J. C. van de Pol and O. Tveretina. A BDD-representation for the logic of equality and uninterpreted functions. In J. Jedrzejowicz and A. Szepietowski, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 3618 of *Lecture Notes in Computer Science*, pages 769–780. Springer, 2005.

[30] R. J. Wilson and J. J. Watkins. *Graphs: an Introductory Approach.* Wiley, 1990.

[31] A. G. Wouters. Manual for the μCRL toolset (version 2.8.2). Technical Report SEN-R0130. CWI, Amsterdam, The Netherlands, 2001.

Appendix A

Graph Theory

It is assumed that the reader of this thesis is acquainted with graph theory. However, to freshen up the memory and to prevent confusion caused by the fact that different definitions circulate in the literature for the same terminology, some basic definitions of graph theory are presented here. This appendix is not meant as a complete introduction to graph theory, for such an introduction the reader is referred to [21, 30]. Here only the terminology that is used throughout the thesis is treated.

A.1 Basic definitions

A *graph* $G = (V, E)$ consists of a set V of nodes, also called vertices, and a set E containing multisets of V of cardinality two. The elements of E are called edges. An edge $\{i, j\}$ connects nodes i and j. The edge set E may contain loops, which are edges from a node to itself. Figure A.1(a) depicts a graph $G = (V, E)$, with $V = \{1, 2, 3\}$ and $E = \{\{1,1\}, \{1,2\}, \{1,3\}, \{2,3\}\}$. The edge $\{1, 1\}$ is a loop.

If the edges of a graph are directed, it is called a *directed* graph (or *digraph* for short). Formally, a digraph $D = (V, A)$ consists of a set V of nodes and a set A of arcs, which are ordered pairs of nodes. An arc (i, j) is directed from node i to node j. Analogously to undirected graphs, the arc set A may contain loops. Figure A.1(b) gives a digraph obtained from Figure A.1(a) by orienting the edges. More precisely, the digraph is defined by $V = \{1, 2, 3\}$ and $A = \{(1,1), (1,2), (3,1), (3,2)\}$.

For an edge $e = \{i, j\}$ in an undirected graph $G = (V, E)$, it is said that e is incident with i and j, and that i is adjacent to j. The number of edges incident with a node i, is called the degree of node i. For an arc $a = (i, j)$ in a digraph $D = (V, A)$, it is said that a is incident from i and incident to j. Furthermore, i and j are again called adjacent. The number of arcs incident from a node i is called the outdegree of node i, while the number of arcs incident to node i is called its indegree. For example, node 3 in the digraph of Figure A.1(b) has outdegree two and indegree zero while the opposite is true for node 2.

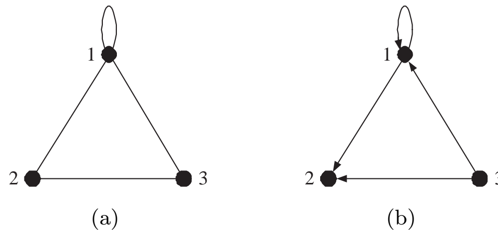

Figure A.1: Examples of (a) a graph and (b) a digraph.

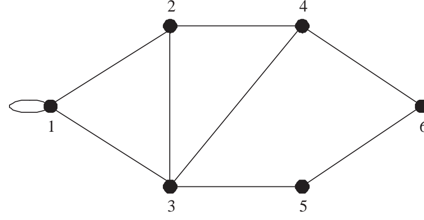

Figure A.2: A graph containing cycles.

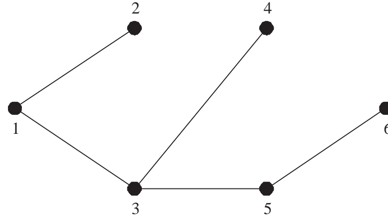

Figure A.3: A spanning tree of the graph of Figure A.2.

Let $G = (V, E)$ be a graph. A walk of length n is a sequence $(v_1, v_2, \ldots, v_n)$ of n nodes, such that $\{v_i, v_{i+1}\} \in E$ for all i with $1 \leq i < n$. If $v_1 = v_n$ and $n > 1$, then the walk is said to be closed. For example, the sequence (1,2,4,2,3) defines a walk in the graph of Figure A.2 of length five and (1,2,4,2,3,1) defines a closed walk in this graph of length six. A walk in which all nodes are distinct is called a path, and a closed walk in which all nodes are distinct except for the first and last one is called a cycle. These definitions easily translate to digraphs.

If graph $G' = (V', E')$ can be obtained from graph $G = (V, E)$ by removing some nodes and/or edges, i.e., if $V' \subseteq V$ and $E' \subseteq E$, then G' is said to be a subgraph of G. For instance, the graph of Figure A.1(a) is a subgraph of the graph of Figure A.2. Similarly, a subgraph of a digraph can be defined. A graph is said to be connected if it contains a path between any two nodes. If this is the case in a digraph, it is called strongly connected. If only the underlying graph of a digraph D is connected, then D is called weakly connected, where the underlying graph of D is obtained by replacing all its arcs by edges. All example graphs given in this appendix are connected. The digraph of Figure A.1(b), however, is only weakly connected since it contains no path from nodes 1 and 2 to node 3 or from node 2 to node 1.

A tree is a connected graph $T = (V, E)$ that does not contain any cycles. Finally, if the tree T is a subgraph of a graph G that contains all nodes of G, then it is called a spanning tree of G. Figure A.3 shows a spanning tree of the graph of Figure A.2.

A.2 Representations of graphs

There are two standard ways to represent a graph: as a collection of adjacency lists or as an adjacency matrix. Both representations are applicable to directed and undirected graphs. The graph algorithms presented in this thesis all assume that an input graph is represented in adjacency list form. Therefore the adjacency matrix will not be discussed here.

The *adjacency list* representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each node in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all nodes v for which there is an edge $\{u, v\} \in E$. That is, $Adj[u]$ consists of all nodes adjacent to u in G. The nodes in each adjacency list are typically stored in an arbitrary order. Adjacency lists for directed graphs are defined in a similar way.

For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of arcs, since an arc of the form (u, v) is represented by the appearance of v in $Adj[u]$. For an undirected

graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges, since if $\{u, v\}$ is an edge, then u appears in the adjacency list of v and vice versa. The adjacency list representation requires $\Theta(V+A)$ memory for a directed graph $D = (V, A)$ and, similarly, $\Theta(V+E)$ memory is required for an undirected graph $G = (V, E)$.

A.3 Depth-first search

To solve many problems dealing with graphs efficiently, the nodes and arcs of a directed graph need to be visited in a systematic fashion. *Depth-first search* is one important technique for doing so.

As its name implies, the strategy followed by depth-first search is to search "deeper" in the graph whenever possible. In depth-first search, edges are explored out of the most recently discovered node v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all nodes that are reachable from the original source node have been discovered. If any undiscovered nodes remain, then one of them is selected as a new source and the search is repeated from this source. This entire process is repeated until all nodes have been discovered.

Whenever a node v is discovered during the scan of the adjacency list of a previously discovered node u, depth-first search records this event by setting v's predecessor field $\pi[v]$ to u. By doing so, depth-first search produces a so called *predecessor subgraph.* Because the search can be continued from multiple sources, this subgraph may be composed of several trees. For a graph $G = (V, E)$ the predecessor subgraph is defined as $G_\pi = (V, E_\pi)$, where $E_\pi = \{(\pi[v], v) : v \in V \wedge \pi[v] \neq \mathsf{nil}\}$. The predecessor subgraph of a depth-first search forms a *depth-first forest* composed of several *depth-first trees.* The edges in E_π are called *tree edges.*

During the search nodes are colored to indicate their state. Each node is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when all nodes in its adjacency list have been examined completely. This coloring technique guarantees that each node occurs in only one depth-first tree, resulting in disjoint trees.

Besides creating a depth-first forest, depth-first search also timestamps each node. Each node v has two timestamps: the first timestamp $d[v]$ records when v is first discovered (and grayed), and the second timestamp $f[v]$ records when the search has finished examining the adjacency list of v (and blackens v). These timestamps are used in many graph algorithms and are generally helpful in reasoning about the behavior of depth-first search.

Definition A.3.1 (Depth-First Search Algorithm)**.** The basic depth-first search algorithm is depicted in Figure A.4. The variable *time* is a global variable that is used for timestamping.

The procedure `DFS` works as follows. Lines 1-3 colors all nodes white and initializes their π fields to nil. Line 4 resets the global variable *time.* Lines 5-6 check each node in V and, when a white node is found, visit it using `DFS-Visit`. Every time `DFS-Visit` is called in line 6, node u becomes the root of a new tree in the depth-first forest. When `DFS` is finished, every node u has been assigned a discovery time $d[u]$ and a finishing time $f[u]$.

In each call `DFS-Visit`(u), node u is initially white. Line 1 colors u gray, line 2 increments the global variable *time*, and line 3 records the new value of *time* as the discovery time $d[u]$. Lines 4-7 examine each vertex v adjacent to u and recursively visit v if it is white. After each node in the adjacency list of u has been examined, lines 8-9 color u black and record the finishing time in $f[u]$.

What is the running time of `DFS`? The loops in lines 1-3 and lines 5-6 of `DFS` take $\Theta(V)$ time, not counting the time it takes to execute the calls to `DFS-Visit`. The procedure `DFS-Visit` is called exactly once for each node $v \in V$, since `DFS-Visit` is only invoked on white nodes and the first thing it does is color the node gray. During an execution of `DFS-Visit`(v), the loop in lines 4-7 is executed $|Adj[v]|$ times. Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$, the total cost of executing lines 4-7 of `DFS-Visit` is $\Theta(E)$. Therefore the running time of `DFS` is $\Theta(V + E)$.

```
DFS(G)
1    for each u ∈ V[G] do
2        color[u] := white;
3        π[u] := nil;
4    time := 0;
5    for each u ∈ V[G] do
6        if (color[u] = white) then DFS-Visit(u);

DFS-Visit(u)
1    color[u] := gray;
2    time := time + 1;
3    d[u] := time;
4    for each v ∈ Adj[u] do
5        if (color[v] = white) then
6            π[v] := u;
7            DFS-Visit(v);
8    color[u] := black;
9    f[u] := time := time + 1;
```

Figure A.4: The depth-first search algorithm.

A.3.1 Classification of edges

An interesting property of depth-first search is that the search can be used to classify the edges of the input graph $G = (V, E)$. This edge classification can be used to gather important information about a graph. For example, in the next section, this classification will be used to determine which nodes of an undirected graph are so called *articulation points*.

Four edge types can be defined in terms of the depth-first forest G_π produced by a depth-first search on a graph G:

1. *Tree edges* are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).

2. *Back edges* are those edges (u, v) connecting a node u to an ancestor v in a depth-first tree. Self-loops, are considered to be back edges.

3. *Forward edges* are those nontree edges (u, v) connecting a node u to a descendant v in a depth-first tree.

4. *Cross edges* are all other edges. These can go between nodes in the same depth-first tree, as long as one node is not an ancestor of the other, or they can go between nodes in different depth-first trees.

The DFS algorithm can be modified to classify edges as it encounters them. The idea is that each edge (u, v) can be classified by the color of the node v that is reached when the edge is first explored (except that forward and cross edges are not distinguished):

1. white indicates a tree edge;

2. gray indicates a back edge; and

3. black indicates a forward or a cross edge.

The first case immediately follows from the algorithm. For the second case, observe that the gray nodes always form a chain of descendants corresponding to the stack of active DFS-Visit calls. Exploration always proceeds from the deepest gray node in te depth-first forest, so an edge that

reaches another gray node reaches an ancestor. The third case handles the remaining possibility; such an edge (u, v) is a forward edge if $d[u] < d[v]$ and a cross edge if $d[u] > d[v]$.

In an undirected graph, there can be some ambiguity in the type classification, since (u, v) and (v, u) are the same edge. There is no distinction between forward and back edges for undirected graphs. Therefore they are all referred to as back edges. Furthermore, it can be shown that cross edges never occur in the depth-first search of an undirected graph.

In the next section a problem will be given that can be solved using the classification of edges discussed in this section.

A.4 Articulation points

Definition A.4.1 (Articulation Point)**.** Let $G = (V, E)$ be a connected undirected graph. An *articulation point* of G is a node whose removal disconnects G.

For example, nodes 1, 3 and 5 are articulation points of the graph depicted in Figure A.3.

The problem of finding articulation points is one of the simplest of many problems concerning the connectivity of graphs. Here a simple depth-first search algorithm will be given. This algorithm uses the tree structure provided by the search to determine which nodes are articulation points. Let G be a connected undirected graph. Because G is connected the depth-first forest G_π will consist of one tree. The following observations can be made about the nodes in this tree:

1. The root of G_π is an articulation point if, and only if, it has two or more children. In this case the removal of the root will disconnect the subtrees rooted at its children;

2. A node u of G_π (other than the root) is an articulation point if, and only if, there exists a subtree rooted at a child v of u such that there is no back edge from any node in this subtree to a proper ancestor of u. In this case the removal of u will disconnect the subtree rooted at v from the rest of the graph.

The first observation is easy to check. Checking the second observation is a bit harder, but the structure of the depth-first tree can be exploited for this. The basic thing to check is whether there is a back edge from some subtree to an ancestor of a given node. This can be done by keeping track of the back edge that goes highest in the tree (in the sense of going closest to the root). If any back edge goes to an ancestor of a given node, this one will. To find out how close a back edge goes to the root the discovery times can be used: when travelling from a node u towards the root, the discovery times of the ancestors of u get smaller and smaller (with the root having the smallest discovery time of 1). So the back edge (v, w) that has the smallest value of $d[w]$ is recorded.

Define $Low[u]$ to be the minimum of $d[u]$ and $\{d[w] |$ where (v, w) is a back edge and v is a descendent of $u\}$, where v may be equal to u. Intuitively, $Low[u]$ is the closest to the root that you can get in the tree by taking one back edge from u or one of its descendents. "Low" means low discovery time, not low in the tree. When performing `DFS-Visit` on node u, $Low[u]$ can be computed as follows:

- Initially $Low[u] = d[u]$;

- If the edge (u, v) is a back edge, then $Low[u] = \mathsf{min}(Low[u], d[v])$. If the newly found back edge goes to a lower d value than the previous back edge, this must be made the new value for $Low[u]$.

- If the edge (u, v) is a tree edge, then $Low[u] = \mathsf{min}(Low[u], Low[v])$. Because v is in the subtree rooted at u any back edge leaving the tree rooted at v is a back edge for the tree rooted at u.

Once $Low[u]$ has been computed for all nodes u, testing whether a given nonroot node u is an articulation point can be done as follows: u is an articulation point if and only if it has a child v in the depth-first tree for which $Low[v] \geq d[u]$ (since if there were a back edge from either v or one of its descendents to an ancestor of v then $Low[v]$ would have been smaller than $d[u]$).

Definition A.4.2 (Articulation Points Algorithm)**.** The algorithm for computing articulation points is depicted in Figure A.5. It uses the same main procedure `DFS` is as the basic depth-first search algorithm from the previous section.

```
DFS(G)
1   for each u ∈ V[G] do
2       color[u] := white;
3       π[u] := nil;
4   time := 0;
5   for each u ∈ V[G] do
6       if (color[u] = white) then ArtPt(u);

ArtPt(u)
1   color[u] := gray;
2   time := time + 1;
3   Low[u] := d[u] := time;
4   for each v ∈ Adj[u] do
5       if (color[v] = white) then
6           π[v] := u;
7           ArtPt(v);
8           Low[u] := min(Low[u], Low[v]);
9           if (π[v] = nil) then
10              if ("this is u's second child") then
11                  "Add u to the set of articulation points";
12            else if (Low[v] ≥ d[u]) then
13                "Add u to the set of articulation points";
14        else if (v ≠ π[u]) then
15            Low[u] = min(Low[u], d[v]);
```

Figure A.5: The articulation points algorithm.

As with all algorithms based on depth-first search, the running time of the algorithm is $\Theta(V + E)$.

Appendix B

Proofs of properties of the simplification operators

In this appendix proofs of properties of the simplification operators defined in chapter 4 will be given. All of these proofs are by induction. Because the operators defined on EQ-BDDs differ only on one additional rule from the operators defined on propositional BDDs, the proofs for these operators will be combined. If the property holds for an operator defined on EQ-BDDs, then it certainly holds for an operator defined on propositional BDDs.

B.1 The Restrict operator

In this section it will be shown that the Restrict operator defined in section 4.2 preserves logical equivalence, and that the paths in the resulting BDD are consistent with the BDD that was used for the simplification.

B.1.1 Preserving logical equivalence

Proof. Let ϕ and ψ be reduced and ordered (EQ-)BDDs. Under the assumption that ψ evaluates to `true` it must be shown that $(\phi \Downarrow \psi) \equiv \phi$, for all ϕ and ψ. The proof is by induction on the pair $(|\phi|, |guards(\psi)|)$, where $|guards(\psi)|$ denotes the number of distinct guards in ψ, and $(x_1, y_1) < (x_2, y_2)$ if and only if $x_1 < x_2$ or $x_1 = x_2 \wedge y_1 < y_2$.

Base (I): $\phi =$ `False` or $\phi =$ `True`. Assuming that ψ evaluates to `true`, the following derivation can be made for the case where $\phi =$ `False`:

$$
\begin{array}{ll}
 & \phi \Downarrow \psi \\
= & \{\ \phi = \texttt{False}\ \} \\
 & \texttt{False} \Downarrow \psi \\
= & \{\ \text{rules for } \Downarrow\ \} \\
 & \texttt{False} \\
= & \{\ \phi = \texttt{False}\ \} \\
 & \phi \\
\equiv & \{\ \text{logic}\ \} \\
 & \phi
\end{array}
$$

Similarly, assuming that ψ evaluates to `true`, $(\phi \Downarrow \psi) \equiv \phi$ can be derived for the case where $\phi =$ `True`.

Base (II): $\psi =$ `True` or $\psi =$ `False`. The following derivation can be made for the case where $\psi =$ `True`:

$$
\begin{array}{ll}
& \phi \Downarrow \psi \\
= & \{ \ \psi = \texttt{True} \ \} \\
& \phi \Downarrow \texttt{True} \\
= & \{ \text{ rules for } \Downarrow \} \\
& \phi \\
\equiv & \{ \text{ logic } \} \\
& \phi
\end{array}
$$

If $\psi = \texttt{False}$ then $\psi \Rightarrow ((\phi \Downarrow \psi) \equiv \phi)$ trivially holds, since `False` represents the boolean function `false` and `false` implies everything. This concludes the base of the proof for the Restrict operator. Next the induction hypothesis will be formulated and the induction step will be taken.

Induction Hypothesis (IH): $\psi \Rightarrow (\phi \equiv (\phi \Downarrow \psi))$ for all ϕ and ψ such that $(|\phi|, |guards(\psi)|) < (m, n)$ where $m, n > 0$.

Induction Step: Let $|\phi| = m$ and $|guards(\psi)| = n$. Then case distinction can be performed on ϕ and ψ.

- Case $root(\phi) < root(\psi)$.
  Assuming that ψ evaluates to `true`, the following derivation can be made:

$$
\begin{array}{ll}
& \phi \Downarrow \psi \\
= & \{ \ root(\phi) < root(\psi), \text{ rules for } \Downarrow \} \\
& ITE(root(\phi), high(\phi) \Downarrow \psi, low(\phi) \Downarrow \psi) \\
\equiv & \{ \text{ Induction Hypothesis, } |high(\phi)| < |\phi|, \ |low(\phi)| < |\phi| \ \} \\
& ITE(root(\phi), high(\phi), low(\phi)) \\
= & \{ \text{ definition of a BDD } \} \\
& \phi
\end{array}
$$

- Case $root(\phi) > root(\psi)$.
  Assuming that ψ evaluates to `true`, it is known that either $high(\psi)$ or $low(\psi)$ evaluates to `true` (or both). The following derivation can be made:

$$
\begin{array}{ll}
& \phi \Downarrow \psi \\
= & \{ \ root(\phi) > root(\psi), \text{ rules for } \Downarrow \} \\
& \phi \Downarrow (high(\psi) \vee low(\psi)) \\
\equiv & \{ \text{ Induction Hypothesis, } |guards(high(\psi) \vee low(\psi))| < |guards(\psi)| \ \} \\
& \phi
\end{array}
$$

  Note: $|guards(high(\psi) \vee low(\psi))| < |guards(\psi)|$ because ψ is reduced and ordered.

- Case $root(\phi) = root(\psi)$ and $high(\psi) = \texttt{False}$.
  Assuming that ψ evaluates to `true` and given that $high(\psi) = \texttt{False}$, it is known that $root(\psi)$ evaluates to `false` and that $low(\psi)$ evaluates to `true`. The following derivation can be made:

$$
\begin{array}{ll}
& \phi \Downarrow \psi \\
= & \{ \ root(\phi) = root(\psi), \ high(\psi) = \texttt{False}, \text{ rules for } \Downarrow \} \\
& low(\phi) \Downarrow low(\psi) \\
\equiv & \{ \text{ Induction Hypothesis, } |low(\phi)| < |\phi| \ \} \\
& low(\phi) \\
\equiv & \{ \ root(\phi) \text{ evaluates to } \texttt{false}, \text{ property of a BDD } \} \\
& \phi
\end{array}
$$

- Case $root(\phi) = root(\psi)$ and $low(\psi) = \texttt{False}$.
  Assuming that ψ evaluates to `true` and given that $low(\psi) = \texttt{False}$, it is known that $root(\psi)$ evaluates to `true` and that $high(\psi)$ evaluates to `true`. The following derivation can be made:

$$\begin{aligned}
&\phi \Downarrow \psi \\
=&\ \{\ root(\phi) = root(\psi),\ low(\psi) = \texttt{False},\ \text{rules for} \Downarrow\ \} \\
&high(\phi) \Downarrow high(\psi) \\
\equiv&\ \{\ \text{Induction Hypothesis},\ |high(\phi)| < |\phi|\ \} \\
&high(\phi) \\
\equiv&\ \{\ root(\phi)\ \text{evaluates to}\ \texttt{true},\ \text{property of a BDD}\ \} \\
&\phi
\end{aligned}$$

- Case $root(\phi) = root(\psi)$, $high(\psi) \neq \texttt{False}$ and $low(\psi) \neq \texttt{False}$.
  Assuming that ψ evaluates to `true` and given that $high(\psi) \neq \texttt{False}$ and $low(\psi) \neq \texttt{False}$, it is known that both $high(\psi)$ and $low(\psi)$ evaluate to `true`, since ψ is reduced and ordered. The following derivation can be made:

$$\begin{aligned}
&\phi \Downarrow \psi \\
=&\ \{\ root(\phi) = root(\psi),\ high(\psi) \neq \texttt{False},\ low(\psi) \neq \texttt{False},\ \text{rules for} \Downarrow\ \} \\
&ITE(root(\phi), high(\phi) \Downarrow high(\psi), low(\phi) \Downarrow low(\psi)) \\
\equiv&\ \{\ \text{Induction Hypothesis},\ |high(\phi)| < |\phi|,\ |low(\phi)| < |\phi|\ \} \\
&ITE(root(\phi), high(\phi), low(\phi)) \\
=&\ \{\ \text{definition of a BDD}\ \} \\
&\phi
\end{aligned}$$

- **Extra for EQ-BDDs:** Case $root(\psi) = (y = x)$ and $low(\psi) = \texttt{False}$.
  Assuming that ψ evaluates to `true` and given that $low(\psi) = \texttt{False}$, it is known that $root(\psi)$ evaluates to `true`. The following derivation can be made:

$$\begin{aligned}
&\phi \Downarrow \psi \\
=&\ \{\ root(\psi) = (y = x) \wedge low(\psi) = \texttt{False},\ \text{rules for} \Downarrow\ \} \\
&\phi[\text{y} := \text{x}] \Downarrow high(\psi) \\
\equiv&\ \{\ \text{Induction Hypothesis},\ |guards(high(\psi))| < |guards(\psi)|\ \} \\
&\phi[\text{y} := \text{x}] \\
\equiv&\ \{\ y = x\ \} \\
&\phi
\end{aligned}$$

For all ϕ and ψ it has been shown that, using the rules for $\Downarrow$ as defined in section 4.2.1, if ψ evaluates to `true` then $(\phi \Downarrow \psi) \equiv \phi$. This concludes the proof for the Restrict operator.

B.1.2 Path-consistency

Proof. Let ϕ and ψ be reduced and ordered (EQ-)BDDs. For all ϕ and $\psi \neq \texttt{False}$, it must be shown that for every path p in the (EQ-)BDD resulting from $\phi \Downarrow \psi$ there exists a valuation υ such that $[\![p \wedge \psi]\!]_\upsilon$. Like the previous proof, the proof is by induction on the pair $(|\phi|, |guards(\psi)|)$.

Base (I): $\phi = \texttt{False}$ or $\phi = \texttt{True}$. If $\phi = \texttt{False}$ then, using the rules defined in section 4.2.1, it must be shown that for every path p in `False` there exists a valuation υ such that $[\![p \wedge \psi]\!]_\upsilon$. Since `False` contains no paths, this trivially holds. Similarly, the same can be shown for $\phi = \texttt{True}$.

Base (II): $\psi = \texttt{True}$ or $\psi = \texttt{False}$. If $\psi = \texttt{True}$ then, using the rules defined in section 4.2.1, it must be shown that for every path p in ϕ there exists a valuation function υ such that $[\![p]\!]_\upsilon$. Since ϕ is reduced and ordered, such a valuation υ exists.

This concludes the base of the proof for the Restrict operator. Next the induction hypothesis is formulated and the induction step is taken.

Induction Hypothesis (IH): $(\phi \Downarrow \psi) \simeq \psi$ for all ϕ and ψ such that $(|\phi|, |variables(\psi)|) < (m, n)$ where $m, n > 0$.

Induction Step: Let $|\phi| = m$ and $|variables(\psi)| = n$. Then case distinction can be performed on ϕ and ψ.

- Case $root(\phi) < root(\psi)$.
  In this case, using the rules of section 4.2.1, it must be shown that for every path p in $ITE(root(\phi), high(\phi) \Downarrow \psi, low(\phi) \Downarrow \psi)$ there exists a valuation υ such that $[\![p \wedge \psi]\!]_\upsilon$. This is equivalent to showing (a) that for every path p' in $high(\phi) \Downarrow \psi$ there exists a υ' such that $[\![p' \wedge root(\phi) \wedge \psi]\!]_{\upsilon'}$, and (b) that for every path p'' in $low(\phi) \Downarrow \psi$ there exists a υ'' such that $[\![p'' \wedge \neg root(\phi) \wedge \psi]\!]_{\upsilon''}$. Since it is known that the root of ϕ does not occur in the result of $high(\phi) \Downarrow \psi$, not in the result of $low(\phi) \Downarrow \psi$ and not in ψ (both ϕ and ψ are reduced and ordered), the existence of a τ for every path p' in $high(\phi) \Downarrow \psi$ such that $[\![p' \wedge \psi]\!]_\tau$ implies the existence of such a υ'. Similarly, the existence of a τ' for every path p'' in $low(\phi) \Downarrow \psi$ such that $[\![p'' \wedge \psi]\!]_{\tau'}$ implies the existence of such a υ''. By the Induction Hypotheses such τ and τ' exist.

- Case $root(\phi) > root(\psi)$.
  In this case, using the rules of section 4.2.1, it must be shown that for every path p in $\phi \Downarrow (high(\psi) \vee low(\psi))$ there exists a valuation function υ such that $[\![p \wedge \psi]\!]_\upsilon$. This is equivalent to showing that for every p there exists a υ' such that $[\![p \wedge ((root(\psi) \wedge high(\psi)) \vee ((\neg root(\psi) \wedge low(\psi))]\!]_{\upsilon'}$. Since it is known that the root of ψ does not occur in the result of $\phi \Downarrow (high(\psi) \vee low(\psi))$, not in $high(\psi)$ and not in $low(\psi)$ (both ϕ and ψ are reduced and ordered), the existence of a υ'' for every p such that $[\![p \wedge (high(\psi) \vee low(\psi))]\!]_{\upsilon''}$ implies the existence of such a υ'. By the Induction Hypotheses such a υ'' exists.

- Case $root(\phi) = root(\psi)$ and $low(\psi) = \texttt{False}$.
  In this case, using the rules of section 4.2.1, it must be shown that for every path p in $high(\phi) \Downarrow high(\psi)$ there exists a valuation function υ such that $[\![p \wedge \psi]\!]_\upsilon$. This is equivalent to showing that for every p there exists a υ' such that $[\![p \wedge root(\psi) \wedge high(\psi)]\!]_{\upsilon'}$. Since it is known that the root of ψ does not occur in the result of $high(\phi) \Downarrow high(\psi)$ and not in $high(\psi)$ (both ϕ and ψ are reduced and ordered), the existence of a υ'' for every p such that $[\![p \wedge high(\psi)]\!]_{\upsilon''}$ implies the existence of such a υ'. By the Induction Hypotheses such a υ'' exists.

- Case $root(\phi) = root(\psi)$ and $high(\psi) = \texttt{False}$.
  In this case, using the rules of section 4.2.1, it must be shown that for every path p in $low(\phi) \Downarrow low(\psi)$ there exists a valuation function υ such that $[\![p \wedge \psi]\!]_\upsilon$. This is equivalent to showing that for every p there exists a υ' such that $[\![p \wedge \neg root(\psi) \wedge low(\psi)]\!]_{\upsilon'}$. Since it is known that the root of ψ does not occur in the result of $low(\phi) \Downarrow low(\psi)$ and not in $low(\psi)$ (both ϕ and ψ are reduced and ordered), the existence of a υ'' for every p such that $[\![p \wedge low(\psi)]\!]_{\upsilon''}$ implies the existence of such a υ'. By the Induction Hypotheses such a υ'' exists.

- Case $root(\phi) = root(\psi)$, $high(\psi) \neq \texttt{False}$ and $low(\psi) \neq \texttt{False}$.
  In this case, using the rules of section 4.2.1, it must be shown that for every path p in $ITE(root(\phi), high(\phi) \Downarrow high(\psi), low(\phi) \Downarrow low(\psi))$ there exists a valuation function υ such that $[\![p \wedge \psi]\!]_\upsilon$. This is equivalent to showing (a) that for every path p' in $high(\phi) \Downarrow high(\psi)$ there exists a υ' such that $[\![p' \wedge root(\psi) \wedge high(\psi)]\!]_{\upsilon'}$, and (b) that for every path p'' in $low(\phi) \Downarrow low(\psi)$ there exists a υ'' such that $[\![p'' \wedge \neg root(\psi) \wedge low(\psi)]\!]_{\upsilon''}$. This has already been shown in the two previous steps of the induction.

- **Extra for EQ-BDDs:** Case $root(\psi) = (y = x)$ and $low(\psi) = \texttt{False}$.
  In this case, using the rules of section 4.2.1, it must be shown that for every path p in $\phi[y := x] \Downarrow high(\psi)$ there exists a valuation function υ such that $[\![p \wedge \psi]\!]_\upsilon$. For this it is sufficient to show that for every p there exists a υ' such that $[\![p \wedge root(\psi) \wedge high(\psi)]\!]_{\upsilon'}$. Since it is known that the root of ψ does not occur in the result of $\phi[y := x]$ and not in $high(\psi)$ (ψ

is ordered and reduced), the existence of a υ'' for every p such that $[\![p \wedge high(\psi)]\!]_{\upsilon''}$ implies the existence of such a υ'. By the Induction Hypotheses such a υ'' exists.

For all ϕ and ψ it has been shown that, using the rules for $\Downarrow$ as defined in section 4.2.1, the paths in the (EQ-)BDD resulting from $\phi \Downarrow \psi$ are consistent with ψ. This concludes the proof for the Restrict operator.

B.2 The Constrain operator

In this section it will be shown that the Constrain operator defined in section 4.3 preserves logical equivalence, and that the paths in the resulting BDD are consistent with the BDD that was used for the simplification. Because the Constrain operator only differs in one rule from the Restrict operator, only the proofs for this rule will be given here. The proofs for the other rules can be found in the previous section.

B.2.1 Preserving logical equivalence

Proof. Let ϕ and ψ be reduced and ordered (EQ-)BDDs. Under the assumption that ψ evaluates to `true` it must be shown that $(\phi \downarrow \psi) \equiv \phi$, for all ϕ and ψ. Like with the Restrict operator, the proof is by induction on the pair $(|\phi|, |guards(\psi)|)$. This proof goes in exactly the same way, except for the case in the Induction Step where $root(\phi) > root(\psi)$. In this case the following derivation can be made:

$$
\begin{aligned}
& \phi \downarrow \psi \\
= & \{ \ root(\phi) > root(\psi), \text{ rules for } \downarrow \ \} \\
& ITE(root(\psi), \phi \downarrow high(\psi), \phi \downarrow low(\psi)) \\
\equiv & \{ \ \text{Induction Hypothesis}, |guards(high(\psi)| < |guards(\psi)|, \\
& \quad |guards(low(\psi)| < |guards(\psi)| \ \} \\
& ITE(root(\psi), \phi, \phi) \\
= & \{ \ \text{elimination} \ \} \\
& \phi
\end{aligned}
$$

For the other cases the reader is referred to the proof for the Restrict operator (section B.1.1). This concludes the proof for the Constrain operator.

B.2.2 Path-consistency

Proof. Let ϕ and ψ be reduced and ordered (EQ-)BDDs. For all ϕ and $\psi \neq$ `False`, it must be shown that for every path p in the (EQ-)BDD resulting from $\phi \downarrow \psi$ there exists a valuation υ such that $[\![p \wedge \psi]\!]_{\upsilon}$. In case $\psi =$ `False`, it is trivial that the result of $\phi \downarrow \psi$ contains no paths that are inconsistent with ψ. Like with the Restrict operator, the proof is by induction on the pair $(|\phi|, |guards(\psi)|)$. This proof goes in exactly the same way, except for the case in the Induction Step where $root(\phi) > root(\psi)$. In this case, using the rules of section 4.3.1, it must be shown that for every path p in $ITE(root(\psi), \phi \downarrow high(\psi), \phi \downarrow low(\psi))$ there exists a valuation function υ such that $[\![p \wedge \psi]\!]_{\upsilon}$. This is equivalent to proving (a) that for every path p' in $\phi \downarrow high(\psi)$ there exists a υ' such that $[\![p' \wedge root(\psi) \wedge high(\psi)]\!]_{\upsilon'}$, and (b) that for every path p'' in $\phi \downarrow low(\psi)$ there exists a υ'' such that $[\![p'' \wedge \neg root(\psi) \wedge low(\psi)]\!]_{\upsilon''}$. Since it is known that the root of ψ does not occur in the result of $\phi \downarrow high(\psi)$, not in the result of $\phi \downarrow low(\psi)$ and not in ϕ (both ϕ and ψ are reduced and ordered), the existence of a τ for every path p' in $\phi \downarrow high(\psi)$ such that $[\![p' \wedge high(\psi)]\!]_{\tau}$ implies the existence of such a υ'. Similarly, the existence of a τ' for every path p'' in $\phi \downarrow low(\psi)$ such that $[\![p'' \wedge low(\psi)]\!]_{\tau'}$ implies the existence of such a υ''. By the Induction Hypotheses such τ and τ' exist.

For the other cases the reader is referred to the proof for the Restrict operator (section B.1.2). This concludes the proof for the Constrain operator.

Appendix C

Contents of `standard.mcrl`

```
sort Bool
func T,F :-> Bool
map not: Bool -> Bool
    and,or,implies,eq: Bool#Bool -> Bool
    if: Bool#Bool#Bool -> Bool

var x,y:Bool
rew and(T,x) = x
    and(F,x) = F
    and(x,T) = x
    and(x,F) = F
    or(T,x) = T
    or(x,T) = T
    or(x,F) = x
    or(F,x) = x
    implies(F,x) = T
    implies(T,x) = x
    implies(x,F) = not(x)
    implies(x,T) = T
    not(T) = F
    not(F) = T
    not(not(x)) = x
    if(T,x,y) = x
    if(F,x,y) = y
    if(x,y,y) = y
    eq(x,x) = T
    eq(T,F) = F
    eq(F,T) = F

sort Nat
func s: Nat -> Nat
     0: -> Nat

map eq,lt,le,gt,ge: Nat#Nat -> Bool
    if: Bool#Nat#Nat -> Nat
    plus: Nat#Nat -> Nat
var
  n,m: Nat
  x: Bool
```

```
rew
  eq(0,0)=T
  eq(s(n),0)=F
  eq(0,s(n))=F
  eq(s(n),s(m))=eq(n,m)

  le(0,n)    = T
  le(s(m),0) = F
  le(s(m),s(n)) = le(m,n)

  lt(m,n) = le(s(m),n)
  ge(m,n) = le(n,m)
  gt(m,n) = le(s(n),m)

  plus(0,n) = n
  plus(n,0) = n
  plus(s(n),m) = s(plus(n,m))
  plus(n,s(m)) = s(plus(n,m))

% extra rewrites

  not(le(m,n)) = le(s(n),m)
  eq(m,m) = T
  le(m,m) = T

  if(T,m,n) = m
  if(F,m,n) = n
  if(x,m,m) = m

sort D
map eq: D#D -> Bool
    if: Bool#D#D -> D
var y,z:D x:Bool
rew eq(y,y) = T     % tool assumes that eq is equality function
    if(T,y,z) = y
    if(F,y,z) = z
    if(x,y,y) = y

sort List
func cons: D#List -> List
     nil: -> List
map eq: List#List -> Bool
    append: List#List -> List
    length: List -> Nat
var d,e: D
    l,m: List
rew eq(nil,nil) = T
    eq(nil, cons(d,l)) = F
    eq(cons(d,l),nil) = F
    eq(cons(d,l),cons(e,m)) = and(eq(d,e),eq(l,m))
    append(nil,l) = l
    append(cons(d,l),m) = cons(d,append(l,m))
    length(nil) = 0
    length(cons(d,l)) = s(length(l))
```

```
% Finally, some uninterpreted function symbols
map a,b,c: -> D
    f,g: D -> D
    h: D#D -> D

    f,g: Nat -> Nat
    h: Nat#Nat -> Nat

init delta
```