A BDD based prover for mCRL2

Engelen, L.J.P.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

A BDD based prover for mCRL2

by

L.J.P. Engelen

Supervisor:
prof.dr.ir. J.F. Groote

Eindhoven, October 2006

Abstract

We present an automated prover based on equational binary decision diagrams (EQ-BDDs) for boolean expressions. This prover is the main component of a number of tools in the mCRL2 toolset. Induction on lists and elimination of inconsistent paths from EQ-BDDs are investigated, to increase the proving power of the prover. The prover and a number of related tools have been implemented and compared with the existing prover and tools for the μCRL toolset.

Contents

Chapter 1

Introduction

The main focus of this thesis is the construction of an automated prover based on equational binary decision diagrams (EQ-BDDs), using the method described in [1]. This method is extended, so that arbitrary boolean expressions in the format of the specification language mCRL2 can be handled. These extensions make the prover incomplete, thus making it desirable to increase the proving power of the prover. To increase the proving power, elimination of inconsistent paths from EQ-BDDs and induction on lists have been investigated.

The prover forms the basis of a number of tools in the mCRL2 toolset. The language mCRL2 combines process algebra with data and timing, and is used to specify the behaviour of systems consisting of communicating components operating in parallel. State spaces of these systems can be created if such a system has a finite state space. The tools based on the prover can, for instance, be used to reduce these state spaces and to check the invariance of certain expressions concerning these systems. A number of examples are described, showing how the elimination of inconsistent paths and induction on lists make it possible to handle specifications that could not be handled successfully before.

Chapter 2

Preliminaries

We start with a short description of the language mCRL2. The prover we constructed takes expressions of sort *Bool* in the internal mCRL2 format as input, and the tools based on the prover process mCRL2 specifications. After describing mCRL2, we give an overview of automated provers to indicate how our prover relates to other provers. Finally, we give a short explanation of the concepts involved in rewriting, as rewriting is a major part of the proving process.

2.1 mCRL2

mCRL2 is a specification language combining process algebra [2] with data and timing, that extends the language μCRL [3]. Every mCRL2 specification consists of a data specification and a process specification. The toolset of the language can be found on the mCRL2 website [4]. In time also general information about the language will be available there. We give a summary of the description of the language given in [5].

2.1.1 Data specification

In mCRL2, data is specified in a language based on higher-order abstract data types. New data types, called *sorts*, can be declared. The domain of a sort is specified using *constructors* and functions over sorts are declared using *maps*. These maps are defined using rewrite rules called *equations*. *Variables* denote arbitrary elements of a sort in the equations that define maps. The following data specification specifies a sort D with elements d_1 and d_2, and a map *inverse*. The inverse of d_1 is defined to be d_2 and vice versa. The variable d denotes an arbitrary element of sort D.

sort	D;
cons	$d_1 : D$;
	$d_2 : D$;
map	$inverse : D \to D$;
var	$d : D$;
eqn	$inverse(d_1) \approx d_2$;
	$inverse(d_2) \approx d_1$;
	$inverse(inverse(d)) \approx d$;

Besides the user defined abstract data types there are concrete data types. The concrete data types consist of *standard data types and functions,* and *type constructors*. The standard data types offer representations for positive, natural, integer and real numbers, along with a number of relations and operators on these numbers such as $<$, $\leq$, $\geq$, $>$, $-$, $+$, $*$, **div** and **mod.** There is a sort representing booleans, *Bool*, with constants *true* and *false,* and operators $\wedge$, $\vee$, $\Rightarrow$ and

$\neg$. For all sorts, including the user defined abstract data types, the operators denoting equality $\approx$ and inequality $\not\approx$ are available, as well as an if-then-else function.

mCRL2 offers type constructors that facilitate the declaration of lists, sets and bags containing elements of previously declared sorts or standard data types. A list of booleans is for instance declared as $List(Bool)$ and has constructors $[] : Bool$ and $\triangleright : Bool \to List(Bool) \to List(Bool)$. Predefined functions over lists are for example *snoc*, $\triangleleft$, and *concatenation*, $+\!\!+$. Besides the type constructors for lists, there are similar type constructors for sets and bags. Another kind of type constructor is the constructor for structured types. It can be used to specify a sort, the constructor functions of the sort, recognizer functions for the constructors and projection functions for the constructors, all at once.

2.1.2 Process specification

A process specification in mCRL2 specifies the behaviour of a system by defining the communicating processes the system consists of and the initial state of the system. The behaviour of the processes is defined by recursive definitions. The basic elements of such a definition are *actions* and *deadlock*. An action represents an atomic event and deadlock, denoted by δ, represents the absence of any activity, i.e. inaction. These basic elements can be composed into process expressions using *sequential composition* and *alternative composition*. The sequential composition of two processes expressions p and q, denoted $p \cdot q$, first executes p and then q, provided that p terminates. The alternative composition of p and q, denoted $p + q$, non-deterministically chooses between the execution of either p or q.

These process expressions can be used to form process equations. The following process specification specifies a process K, that can either execute an action a followed by an action b, or a multiaction $a \mid b$.

$$\begin{array}{ll} \textbf{act} & a, b; \\ \textbf{proc} & K = ((a \cdot b) + (a|b)) \cdot K; \end{array}$$

Processes can be combined to form new processes using *parallel composition*, *synchronisation* and *left merge*. $p \| q$, i.e. p in parallel with q, is the interleaving and synchronisation of the actions in p with those in q. $p \mid q$, i.e. the synchronisation of p and q, denotes the process that first executes the synchronisation of the first action of p with the first action of q, followed by the parallel composition of the remaining actions of p and the remaining actions of q. A special instance of the synchronisation operator takes a number of single actions and combines them into a *multiaction*. A multiaction is the execution of all of these single actions at the same time. The actions a, b and c can, for instance, be synchronised to a multiaction $a \mid b \mid c$. The left merge of p and q, denoted $p \lfloor\!\lfloor q$, denotes the process that executes the first action of p followed by the parallel composition of the remainder of p and q.

There are a number of operators that can be applied to a process to restrict its behaviour. These are the *restriction operator*, the *blocking operator*, the *renaming operator* and the *communication operator*. The restriction operator, $\nabla_V(p)$, also known as the *allow operator*, takes a set V of multiactions and a process p, and returns the process p' that results from removing all actions that are not in V from process p. The blocking operator, $\partial_H(p)$, also known as the *encapsulation operator*, takes a set H of action names and a process p, and returns the process p' that results from replacing all actions in p that occur in H by deadlock. The renaming operator, $\rho_R(p)$, takes a process p and a set R of renamings of the form $a \to b$, and replaces all actions from p that occur on the left-hand side of a renaming with the corresponding action on the right-hand side. The communication operator, $\Gamma_C(p)$, specifies which actions can communicate and will be replaced by one action representing a succesful communication when combined together with the synchronisation operator. It takes a process p and a set C of communications of the form $a_0 \mid \ldots \mid a_n \to c$, with $n \geq 1$, and replaces all occurences of the multiaction $a_0 \mid \ldots \mid a_n$ in p by c.

To allow for abstraction the following two constructs are available. The internal action or silent step τ represents unknown internal behaviour. The hiding operator takes a set I and a process P and replaces all actions in P occuring in the set I by the special multiaction τ.

Actions can be parameterized with data. An action a with data parameters of sort *Bool* and *Int* is specified as follows:

$$\textbf{act} \quad a : Bool \times Int;$$

Processes can also be parameterized with data. The behaviour of a process can be influenced by its data parameters. The conditional operator takes a data expression c of sort *Bool* and two process expressions p and q and is denoted as $c \rightarrow p \diamond q$. The process expression $c \rightarrow p \diamond q$ behaves as p if the expression c evaluates to *true*, otherwise it behaves as q. A simpler form of the conditional operator exists, taking only one process expression. It is denoted as $c \rightarrow p$ and behaves as $c \rightarrow p \diamond \delta$. The summation operator takes a variable d of sort D and a process expression p, possibly containing d. The resulting process expression, denoted as $\sum_{d:D} p$, behaves as $p[d_0/d] + \cdots + p[d_n/d]$, with $n \geq 0$, for all elements $d_i \in D$. $p[d_i/d]$ stands for p in which all free occurences of d are replaced by d_i. The following specification specifies a process P with a data parameter v of sort *Int*.

$$\begin{array}{ll} \textbf{act} & a : Int; \\ & b; \\ \textbf{proc} & P(v : Int) = \sum_{i:Int} a(i) \cdot P(i) + (v > 0) \rightarrow b \cdot P(v); \end{array}$$

An mCRL2 process specification has to be linearized to a linear process equation (LPE), before it can be processed by the tools in the mCRL2 toolset [6]. An LPE is a process equation of the form

$$\textbf{proc} \quad P(d : D) \quad = \quad \sum_{i \in I} \sum_{e_i \in E_i} c_i(d, e_i) \rightarrow a_i(f_i(d, e_i)) \cdot P(g_i(d, e_i));$$

where I is a finite index set, $c_i(d, e_j)$ a condition, $f_i(d, e_j)$ the parameter of action a_i and $g_i(d, e_j)$ the next state. The first summand, $\sum_{i \in I}$, is used as a shorthand only. In general, $\sum_{i \in I} p$ is a shorthand for $p_1 + \cdots + p_n$, where n is the size of I. Chapter 6 describes applications that manipulate LPEs using the BDD based prover.

2.2 Automated provers

The applications we describe in chapter 6 all involve constructing mCRL2 data expressions of sort *Bool* and then using a prover to check whether or not these expressions are tautologies or contradictions. An expression of sort *Bool* is a contradiction if it is equal to *false* for all valuations of the variables occuring in the formula. An expression of sort *Bool* that is equal to *true* for all valuations of the variables occuring in the formula is called a tautology.

Proof assistants, like COQ and PVS, combine a specification language with a theorem prover and can be used interactively to prove properties expressed in this specification language. Implementations of the applications in chapter 6 must check large amounts of formulas. The interactivity that is part of the proving process is a drawback of these proof assistants in this kind of situation, as it makes it unfeasible to check a large amount of formulas in a reasonable amount of time.

Checking the satisfiability of an expression can serve as a substitute for checking whether or not the expression is a tautology or a contradiction. An expression is satisfiable if it is equal to *true* for some valuation of the variables occuring in the expression. Checking whether or not an expression is a tautology can be simulated by checking if the negation of that expression is satisfiable, provided the underlying logic is closed under negation. If the negated expression is not satisfiable, the expression is a tautology. Tools that check the satisfiability of boolean expressions are often referred to as SAT solvers.

The expressions read by SAT solvers are often formulas in propositional logic. They consist of boolean variables combined using logical connectives. The data expressions of sort *Bool* that we want to check can contain equalities and inequalities concerning variables and constants of a number of different sorts, as well as functions and operators over these sorts, which makes SAT solvers unsuited for our task. SMT solvers are a variation on SAT solvers. They extend the

language of the expressions serving as input with standard sorts, functions and operators. They check whether or not expressions containing these extensions are satisfiable given one or more background theories, hence the acronym SMT, that stands for "satisfiability modulo theories". A more elaborate discussion of SMT solvers is given in chapter 8.

The prover we constructed is a fully automated prover that does not depend on human interaction. It receives expressions of sort *Bool* as input and can indicate whether or not these expressions are tautologies or contradictions. It differs from SMT solvers, because the prover itself does not offer any functionality to handle specific sorts, functions or operators. The prover can only use properties of such sorts, functions or operators in the proving process if those properties are expressed in rewrite rules.

2.3 Rewriter

The algorithm used for translating formulas to EQ-BDDs as described in chapter 3 uses a so called *rewriter*. A rewriter rewrites *terms* using a set of *rewrite rules*, according to a certain *strategy*. We give a short explanation of the concepts *term*, *rewrite rule* and *strategy*, following the definitions given in [7].

Given a set of variables and a set of functions, a term is either a variable or a function with arity n applied to n terms. A rewrite rule is a pair of terms $l \mapsto r$, where l is not a variable and all variables occuring in r occur in l, too. A substitution is a mapping from variables to terms. The term t with all variables x in t replaced by $\sigma(x)$ is denoted by t^σ. Term t can be rewritten using rewrite rule $l \mapsto r$ if t has a subterm t' for which there exists a substitution σ such that $l^\sigma = t'$. The resulting term is equal to term t with subterm t' replaced by r^σ. A term is in normal form if there are no rewrite rules that can be applied to the term.

A strategy indicates how a term is rewritten, based on the top-level function symbol of that term. Suppose we have a ternary function *if*, two nullary functions T and F, a set of variables $\{b, x, y\}$ and a set of rewrite rules $\{\alpha : if(T, x, y) \mapsto x, \beta : if(F, x, y) \mapsto y, \gamma : if(b, x, x) \mapsto x\}$. Rewriting a term $if(p, q, r)$ using rewrite rule α or β can be done without first rewriting subterms r and q respectively. Only if subterm p can not be rewritten to T or F, both q and r have to be rewritten to see whether or not rewrite rule γ is applicable. A strategy corresponding to function *if* would be to first rewrite the subterm that is the first argument of the function to a normal form and then to check whether or not rewrite rules α and β can be applied. If they cannot be applied, the second and third argument of the function have to be rewritten to a normal form to see whether or not rewrite rule γ can be applied.

The implementation of the prover we built uses a rewriter that can rewrite mCRL2 data expressions using rewrite rules in mCRL2 format. This rewriter can be set to use one of a number of different rewrite strategies. A detailed description of the implementation of this rewriter can be found in [8].

Chapter 3

Constructing equational binary decision diagrams

This chapter discusses the process of constructing a so called equational binary decision diagram corresponding to a boolean expression. In short, these boolean expressions are formed by applying boolean connectives, like $\wedge$,$\vee$,$\neg$ or $\Rightarrow$, to other boolean expressions, boolean variables, denoted by p, q, ..., or the boolean constants *true* and *false*. We start by describing and defining binary decision diagrams, and then extend these binary decision diagrams to equational binary decision diagrams. The main task of the prover we constructed is converting boolean expressions to equational binary decision diagrams. The final part of this chapter describes this process.

3.1 Reduced ordered binary decision diagrams

Binary decision diagrams (BDDs) provide a way of representing boolean expressions [9]. A BDD is a rooted directed acyclic graph. Each of the nodes of the graph has exactly two outgoing edges. The nodes are labelled with boolean variables and the leaves are labelled "*true*" and "*false*".

The value of a BDD for a valuation of the boolean variables is determined by starting at the root and traversing the BDD until a leaf is reached. The valuation of the variable that labels a node determines which edge will be taken during the traversal. If the variable is *true* in the valuation, the *true*-edge is taken. If the variable is *false*, the *false*-edge is taken. The label of the leaf that is reached at the end of this process corresponds to the value of the expression for that particular valuation of the variables.

In the figures in this report, the *true*-edge is depicted as a solid line, whereas the *false*-edge is depicted as a dashed line. We will refer to the subgraphs connected to the *true*-edge and the *false*-edge of a node as the *true*-branch, respectively the *false*-branch. Figure 3.1 shows a BDD representing the predicate $\neg(p \wedge q)$.

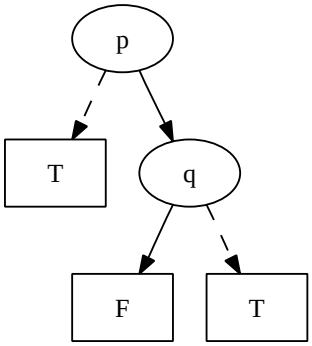

Figure 3.1: BDD representing $\neg(p \wedge q)$

Following the description above, a boolean function can be represented by more than one BDD. Figure 3.2 shows another BDD representing the same predicate. To eliminate this redundancy, the notion of orderedness is introduced, yielding Ordered BDDs (OBDDs). In an OBDD each variable in the label of a node is smaller than all the variables in the nodes below that node, for some total order on the boolean variables. Given a total order on the boolean variables $\succ$, with $q \succ p$, the BDD in figure 3.2 would not be allowed. Note that the size of an OBDD is determined by the boolean function it represents as well as the chosen order on the variables. OBDDs provide an efficient representation for many boolean functions, but the choice of a wrong order can dramatically increase the size of an OBDD.

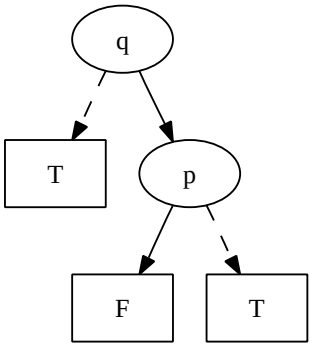

Figure 3.2: Another BDD representing $\neg(p \wedge q)$

Still, there exists more than one OBDD representing a boolean function, given a total order on the boolean variables. The *true*-branch and the *false*-branch of the node labelled "p" in figure 3.3 are the same. Replacing the node labelled "p" by one of the two branches yields an equivalent BDD.

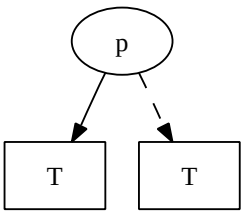

Figure 3.3: Node with two equal branches

The BDD shown in figure 3.4 is another example of an OBDD containing redundant information.

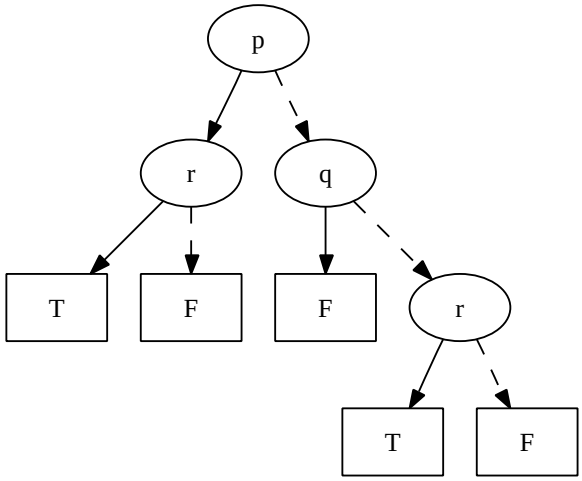

Figure 3.4: BDD with redundancy

To eliminate this redundancy, the following reduction rules are introduced, referred to as the elimination rule and the merging rule in [10].

- **Elimination Rule:** Replace all nodes with equal *true*- and *false*-branches with their *true*-branch.
- **Merging Rule:** If two nodes are labelled with the same boolean variable, their *true*-edges lead to the same node and their *false*-edges lead to the same node, then remove one of the two nodes and redirect all incoming edges of that node to the other node. If the merging rule cannot be applied to an OBDD, the OBDD is *maximally shared*.

An OBDD is called an Reduced Ordered Binary Decision Diagram (ROBDD) if the elimination rule and the merging rule cannot be applied. Figure 3.5 shows the ROBDD equivalent to the OBDD in figure 3.4.

Equivalent boolean functions are represented by the same ROBDD. Because of this property, checking if a boolean function is a tautology or a contradiction can be done in constant time. All tautologies are represented by the ROBDD consisting of one leaf labelled "true" and all contradictions are represented by the ROBDD consisting of one leaf labelled "false".

3.2 Formal definition of reduced ordered binary decision diagrams

We give a formal definition of binary decision diagrams following the definitions given in [1]. There, an approach is taken that uses terminology from term rewriting systems (TRSs) to define properties and extensions of BDDs. An introduction to term rewriting systems can be found in [11] and [12]. In short, a term rewrite system is a set of function symbols and variables, and a set of rewrite rules on terms constructed using those function symbols and variables.

We start by specifying what we mean with the boolean expressions mentioned earlier. The boolean expressions used in this definition are propositional formulas.

Definition 3.2.1. Propositional formulas are expressions satisfying the following syntax:

$$\Phi \quad ::= \quad \mathit{true} \mid \mathit{false} \mid P \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathit{ITE}(\Phi, \Phi, \Phi),$$

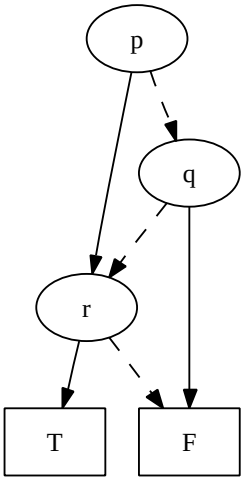

Figure 3.5: ROBDD

where P denotes a set of proposition variables and $ITE(\Phi, \Psi, \Xi)$ is an if-then-else formula, equivalent to $(\Phi \wedge \Psi) \vee (\neg\Phi \wedge \Xi)$. Elements of the set P are denoted by p, q, ... in the remainder of this text.

Given an interpretation function $J : P \to \{true, false\}$, a formula Φ evaluates to either *true* or *false*.

Definition 3.2.2. The value that Φ evaluates to given interpretation function $J : P \to \{true, false\}$, denoted by $[\![\Phi]\!]_J$, is defined by induction over the syntactic structure of Φ.

$$
\begin{aligned}
[\![false]\!]_J &= false \\
[\![true]\!]_J &= true \\
[\![p]\!]_J &= J(p) \\
[\![\neg\Phi]\!]_J &= \begin{cases} true, & \text{if } [\![\Phi]\!]_J = false \\ false, & \text{otherwise} \end{cases} \\
[\![\Phi \wedge \Psi]\!]_J &= \begin{cases} true, & \text{if } [\![\Phi]\!]_J = true \text{ and } [\![\Psi]\!]_J = true \\ false, & \text{otherwise} \end{cases} \\
[\![ITE(\Phi, \Psi, \Xi)]\!]_J &= \begin{cases} [\![\Psi]\!]_J, & \text{if } [\![\Phi]\!]_J = true \\ [\![\Xi]\!]_J, & \text{otherwise} \end{cases}
\end{aligned}
$$

A formula Φ is a *tautology* if $[\![\Phi]\!]_J = true$ for all $J : P \to \{true, false\}$. Similarly, a formula Φ is a *contradiction* if $[\![\Phi]\!]_J = false$ for all $J : P \to \{true, false\}$. If Φ is not a contradiction, i.e. if there exists a $J : P \to \{true, false\}$ such that $[\![\Phi]\!]_J = true$, it is *satisfiable*.

Definition 3.2.3. We define BDDs as expressions satisfying the following syntax:

$$B \quad ::= \quad true \mid false \mid ITE(P, B, B)$$

Given a total order on P, $\succ$, we define the notion of orderedness for BDDs.

Definition 3.2.4. A BDD is ordered if, and only if, it is a normal form w.r.t. the following term rewrite system

1. $ITE(P, T, T) \to T$.
2. $ITE(P, ITE(P, T_1, T_2), T_3) \to ITE(P, T_1, T_3)$.

3. $ITE(P, T_1, ITE(P, T_2, T_3)) \rightarrow ITE(P, T_1, T_3)$.

4. $ITE(P_1, ITE(P_2, T_1, T_2), T_3) \rightarrow ITE(P_2, ITE(P_1, T_1, T_3), ITE(P_1, T_2, T_3))$, provided $P_1 \succ P_2$.

5. $ITE(P_1, T_1, ITE(P_2, T_2, T_3)) \rightarrow ITE(P_2, ITE(P_1, T_1, T_2), ITE(P_1, T_1, T_3))$, provided $P_1 \succ P_2$.

There is no rule expressing the property of maximal sharing. We implement terms using the annotated term library [13], causing all terms to be maximally shared.

3.3 Equational binary decision diagrams

In this section, we extend BDDs with equality, yielding EQ-BDDs. Assuming a set V of domain variables, whose elements will be denoted by x, y, ..., we first extend our notion of formulas.

Definition 3.3.1. Formulas are expressions satisfying the following syntax:

$$\Phi \quad ::= \quad \mathit{true} \mid \mathit{false} \mid P \mid V = V \mid \neg\Phi \mid \Phi \wedge \Phi \mid ITE(\Phi, \Phi, \Phi),$$

where P again denotes a set of proposition variables and ITE is an if-then-else formula.

Given an interpretation function $J : P \rightarrow \{\mathit{true}, \mathit{false}\}$, a domain for the variables D and an interpretation function $I : V \rightarrow D$, a formula Φ evaluates to either *true* or *false*.

Definition 3.3.2. The value that Φ evaluates to given a domain of variables D and interpretation functions $I : V \rightarrow D$ and $J : P \rightarrow \{\mathit{true}, \mathit{false}\}$, denoted by $[\![\Phi]\!]^I_J$, is defined by induction over the syntactic structure of Φ.

$$\begin{aligned}
[\![\mathit{false}]\!]^I_J &= \mathit{false} \\
[\![\mathit{true}]\!]^I_J &= \mathit{true} \\
[\![p]\!]^I_J &= J(p) \\
[\![x = y]\!]^I_J &= \begin{cases} \mathit{true}, & \mathit{if}\ I(x) = I(y) \\ \mathit{false}, & \mathit{otherwise} \end{cases} \\
[\![\neg\Phi]\!]^I_J &= \begin{cases} \mathit{true}, & \mathit{if}\ [\![\Phi]\!]^I_J = \mathit{false} \\ \mathit{false}, & \mathit{otherwise} \end{cases} \\
[\![\Phi \wedge \Psi]\!]^I_J &= \begin{cases} \mathit{true}, & \mathit{if}\ [\![\Phi]\!]^I_J = \mathit{true}\ \mathit{and}\ [\![\Psi]\!]^I_J = \mathit{true} \\ \mathit{false}, & \mathit{otherwise} \end{cases} \\
[\![ITE(\Phi, \Psi, \Xi)]\!]^I_J &= \begin{cases} [\![\Psi]\!]^I_J, & \mathit{if}\ [\![\Phi]\!]^I_J = \mathit{true} \\ [\![\Xi]\!]^I_J, & \mathit{otherwise} \end{cases}
\end{aligned}$$

A formula Φ is *valid* in domain D if $[\![\Phi]\!]^I_J = \mathit{true}$ for all $J : P \rightarrow \{\mathit{true}, \mathit{false}\}$ and $I : V \rightarrow D$. Formula Φ is a *tautology* if it is valid in all domains D and it is *satisfiable* if for some domain D and interpretations I, J, $[\![\Phi]\!]^I_J = \mathit{true}$. A formula that is not satisfiable is a *contradiction*.

Definition 3.3.3. Guards G and EQ-BDDs B are expressions satisfying the following syntax:

$$\begin{aligned}
G &::= P \mid V = V \\
B &::= \mathit{true} \mid \mathit{false} \mid ITE(G, B, B)
\end{aligned}$$

To define the orderedness property for EQ-BDDs, we define an order on guards by taking a total order $\succ$ on $P \cup V$ and extending it lexicographically to guards.

Definition 3.3.4. The total order $\succ$ on $P \cup V$ is extended to guards as follows:

$$\begin{array}{rl} (x = y) \succ p & \textit{if, and only if}, x \succ p \\ p \succ (x = y) & \textit{if, and only if}, p \succ x \\ (x = y) \succ (u = v) & \textit{if, and only if, either } x \succ y, \textit{ or } x \equiv u \textit{ and } y \succ v. \end{array}$$

Given this total order on guards, we define the orderedness of EQ-BDDs.

Definition 3.3.5. An EQ-BDD is ordered if, and only if, it is a normal form w.r.t. 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. $ITE(y = x, T_1, T_2) \rightarrow ITE(x = y, T_1, T_2)$, provided $y \succ x$.
8. $ITE(x = y, T_1[y], T_2) \rightarrow ITE(x = y, T_1[x], T_2)$, provided $y \succ x$ and y occurs in T_1.

In [1] a proof is given of the fact that an equivalent EQ-OBDD exists for every EQ-BDD. As stated before, equivalent formulas are represented by the same ROBDD. This property, however, does not hold for EQ-OBDDs. A proof of the fact that checking whether or not a formula represented by an EQ-OBDD is a tautology or a contradiction can still be done in constant time, is also given in [1].

3.4 Translating formulas to EQ-OBDDs

The proving process of the prover we implemented is based on an algorithm descibed in [1], that translates formulas into ordered EQ-BDDs. To formally describe this algorithm, a term rewrite system *Simplify* is introduced. Simplify removes all superfluous occurences of *true* and *false* from formulas, and orients equations. An equation is oriented if the variable occuring in the left-hand side is smaller than the variable occuring in the right-hand side, for some order $\succ$ on those variables.

Definition 3.4.1. The term rewrite system *Simplify* consists of the following rules:

1. $\textit{false} \wedge x \rightarrow \textit{false}$
2. $x \wedge \textit{false} \rightarrow \textit{false}$
3. $\textit{true} \wedge x \rightarrow x$
4. $x \wedge \textit{true} \rightarrow x$
5. $\neg \textit{true} \rightarrow \textit{false}$
6. $\neg \textit{false} \rightarrow \textit{true}$
7. $ITE(\textit{true}, x, y) \rightarrow x$

8. $ITE(false, x, y) \to y$
9. $x = x \to true$
10. $y = x \to x = y$, provided $y \succ x$

We write $\Phi\downarrow$ for the normal form of Φ obtained by this rewrite system. Φ is called simplified, if $\Phi \equiv \Phi\downarrow$. Assuming that Φ is simplified, we give a recursive definition of the operator $|$. In $\Phi|_s$, s is either of the form p, $\neg p$, $x = y$ or $x \neq y$.

Definition 3.4.2. For simplified formulas Φ, $\Phi|_s$ is defined as follows, in case both Φ and s are guards:

$$
\begin{aligned}
q|_p &\equiv \begin{cases} true, & \text{if } p \equiv q \\ q, & \text{if } p \not\equiv q \end{cases} \\
q|_{x=y} &\equiv q \\
q|_{\neg p} &\equiv \begin{cases} false, & \text{if } p \equiv q \\ q, & \text{if } p \not\equiv q \end{cases} \\
q|_{x\neq y} &\equiv q \\
(u = v)|_p &\equiv u = v \\
(u = v)|_{x=y} &\equiv \begin{cases} true, & \text{if } u \equiv x \text{ and } v \equiv y \\ x = v, & \text{if } u \equiv y \text{ and } v \not\equiv y \\ u = x, & \text{if } u \not\equiv y \text{ and } u \not\equiv x \text{ and } v \equiv y \\ u = v, & \text{if } u \not\equiv y \text{ and } v \not\equiv y \end{cases} \\
(u = v)|_{\neg p} &\equiv u = v \\
(u = v)|_{x\neq y} &\equiv \begin{cases} false, & \text{if } u \equiv x \text{ and } v \equiv y \\ u = v, & \text{otherwise} \end{cases}
\end{aligned}
$$

The definition continues as follows for the remaining cases:

$$
\begin{aligned}
false|_s &\equiv false \\
true|_s &\equiv true \\
\neg\Phi|_s &\equiv \neg(\Phi|_s) \\
(\Phi_1 \wedge \Phi_2)|_s &\equiv (\Phi_1|_s) \wedge (\Phi_2|_s) \\
ITE(\Phi_1, \Phi_2, \Phi_3)|_s &\equiv ITE(\Phi_1|_s, \Phi_2|_s, \Phi_3|_s)
\end{aligned}
$$

The algorithm we describe is based on the Shannon expansion, stating for propositional logic:

$$\Phi \Leftrightarrow (\neg p \wedge \Phi|_{\neg p}) \vee (p \wedge \Phi|_p) \Leftrightarrow ITE(p, \Phi|_p, \Phi|_{\neg p})$$

The Shannon expansion still holds if p is an equation. The variant of the Shannon expansion for propositional logic with equations reads as follows:

$$\Phi \Leftrightarrow (x \neq y \wedge \Phi|_{x\neq y}) \vee (x = y \wedge \Phi|_{x=y}) \Leftrightarrow ITE(x = y, \Phi|_{x=y}, \Phi|_{x\neq y})$$

The algorithm that translates formulas to EQ-BDDs uses a function *Topdown*, that takes the smallest equation in Φ, $x = y$, oriented such that $y \succ x$ in the variable order and recursively applies the Shannon expansion. The function *Topdown* uses a function $\overline{ITE}$, taking a guard and two EQ-BDDs and returning an EQ-BDD.

Definition 3.4.3. The function $\overline{ITE}$ is defined as

$$\overline{ITE}(g, T, U) \equiv \begin{cases} T & \text{if } T \equiv U \\ ITE(g, T, U) & \text{otherwise} \end{cases}$$

Definition 3.4.4. Let Φ be a simplified formula. We define *Topdown* as follows:

$$\begin{aligned} \mathit{Topdown}(\mathit{true}) &\equiv \mathit{true} \\ \mathit{Topdown}(\mathit{false}) &\equiv \mathit{false} \\ \mathit{Topdown}(\Phi) &\equiv \overline{ITE}(g, \mathit{Topdown}(\Phi|_{g}\downarrow), \mathit{Topdown}(\Phi|_{\neg g}\downarrow), \end{aligned}$$

where g is the smallest guard in Φ.

There is no guarantee that the EQ-BDD constructed using *Topdown* is ordered, but it is proved in [1] that repeatedly applying the algorithm does lead to an ordered EQ-BDD. The final algorithm for constructing an ordered EQ-BDD from a formula Φ can now be given:

```
EQ-BDD(Φ) =
  Ψ := Φ↓
  Φ := ⊥
  while Φ ≠ Ψ do
    Φ := Ψ
    Ψ := Topdown(Ψ)
  od
  return Ψ
```

Chapter 4

Further extensions of EQ-BDDs

The implementation of the algorithm described in section 3.4 takes mCRL2 data expressions of sort *Bool* as its input. These data expressions are an extension of the formulas defined in section 3.3. They can contain function symbols and variables from a number of domains. We will not describe the complete extension, but illustrate the general principle in the following section instead. The final section describes how the order on guards is implemented.

4.1 Extending formulas

Recall the definition of formulas from section 3.3: Assuming a set V of domain variables, formulas are expressions satisfying the following syntax:

$$\Phi \quad ::= \quad true \mid false \mid P \mid V = V \mid \neg\Phi \mid \Phi \wedge \Phi \mid ITE(\Phi, \Phi, \Phi),$$

where P denotes a set of proposition variables and ITE is an if-then-else formula. EQ-BDDs B and guards G are then defined as expressions satisfying the following syntax:

$$\begin{array}{lcl} G & ::= & P \mid V = V \\ B & ::= & true \mid false \mid ITE(G, B, B) \end{array}$$

In our implementation, there is more than one set V of domain variables, and each of those sets has its own equality operator. A set of domain variables and an equality operator exists for each of the pre-defined and user-defined sorts. In addition, each pre-defined or user-defined map introduces a function, extending the notion of formulas. Sorts and maps extend the definition of formulas as illustrated in the following example. The data specification

$$\begin{array}{ll} \textbf{sort} & D; \\ \textbf{cons} & d_1, d_2 : D; \\ \textbf{var} & d : D; \\ \textbf{eqn} & d \approx d = true; \\ & d_1 \approx d_2 = false; \\ & d_2 \approx d_1 = false; \\ & inverse(d_1) \approx d_2; \\ & inverse(d_2) \approx d_1; \end{array}$$

defines a set of domain variables V_D, a function *inverse* and an equality operator $=_D$.

Definition 4.1.1. Formulas Φ are now defined as expressions satisfying the following syntax:

$$\begin{array}{lcl} E_D & ::= & V_D \mid inverse(V_D) \\ \Phi & ::= & true \mid false \mid P \mid E_D =_D E_D \mid \neg\Phi \mid \Phi \wedge \Phi \mid ITE(\Phi, \Phi, \Phi) \end{array}$$

Definition 4.1.2. EQ-BDDs B and guards G are defined as expressions satisfying the following syntax:

$$\begin{array}{lcl} G & ::= & P \mid E_D =_D E_D \\ B & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{ITE}(G, B, B) \end{array}$$

The rewrite rules

$$\begin{array}{ll} \textbf{eqn} & d \approx d = \mathit{true}; \\ & d_1 \approx d_2 = \mathit{false}; \\ & d_2 \approx d_1 = \mathit{false}; \\ & \mathit{inverse}(d_1) = d_2; \\ & \mathit{inverse}(d_2) = d_1; \end{array}$$

extend the term rewrite system *Simplify* with the following rules:

1. $d =_D d \rightarrow \mathit{true}$
2. $d_1 =_D d_2 \rightarrow \mathit{false}$
3. $d_2 =_D d_1 \rightarrow \mathit{false}$
4. $\mathit{inverse}(d_1) \rightarrow d_2$
5. $\mathit{inverse}(d_2) \rightarrow d_1$

Besides these rules that simplify guards, a rule that orients guards is added, viz.

6. $t_D =_D r_D \rightarrow r_D =_D t_D$, provided $t_D \succ r_D$,

where t_D and r_D denote expressions consisting of variables of sort D and the function *inverse*.

4.2 Implementation of the order on guards

The implementation uses the efficient annotated term library [13] to represent mCRL2 data expressions as terms. The order $\succ$ mentioned in the last rewrite rule is implemented as the lexicographical path ordering (LPO) on these terms. Using the definition of terms introduced in section 2.3, the order $\succ$ is defined as follows, following the definition of LPO in [14]:

Definition 4.2.1. $s \succ t$ holds if either

1. $s \equiv f(s_1, \ldots, s_n)$, $t \equiv x$ and the term $f(s_1, \ldots, s_n)$ contains x or
2. $s \equiv f(s_1, \ldots, s_n)$, $t \equiv g(t_1, \ldots, t_m)$, and
   (a) $s_i \succeq t$, for all i, $1 \leq i \leq n$, or
   (b) $f >_a g$ and $s \succ t_k$ for all k, $1 \leq k \leq m$, or
   (c) $f = g$, $\langle \exists_i : 1 \leq i \leq m : \langle \forall_j : 1 \leq j < i : s_j = t_j \rangle \wedge s_i \succ t_i \rangle$ and $\langle \forall_k : 1 \leq k \leq m : s \succ t_k \rangle$,

where f is a function of arity n, g is a function of arity m, and $s_1, \ldots, s_n, t_1, \ldots, t_m, s$ and t are terms. The order $\succeq$ is defined as follows for all terms u and v:

Definition 4.2.2. $u \succeq v$ holds if either $u \succ v$ or $u \equiv v$ holds.

The order $>_a$ is a total order on function symbols and variables. These function symbols and variables are represented by pointers in the efficient annotated term library used in the implementation. For all function symbols and variables x and y, $x >_a y$ holds if the memory address of the pointer representing x is larger than the memory address of the pointer representing y.

Chapter 5

Proving using EQ-BDDs

5.1 Basic functionality of a prover

Using the theory described in chapter 3, we constructed a prover that reads data expressions of sort *Bool* and transforms these to ordered EQ-BDDs. The prover then inspects the resulting EQ-BDD to indicate whether or not the original expression is a tautology or a contradiction. Because of the extensions to the input formulas made in chapter 4, the algorithm described in section 3.4 is no longer complete. If the resulting EQ-BDD is not equal to *true* or *false,* there is no way of telling whether or not the expression provided as input was a tautology or a contradiction. In this case the prover will indicate that it is undeterminable whether or not the input was tautological or contradictory. The incompleteness of the procedure can lead to so called inconsistent paths in the EQ-BDD. Chapters 7, 9 and 10 will discuss methods to reduce the number of inconsistent paths in EQ-BDDs.

Besides the data expression of sort *Bool*, the prover uses a set of data equations as its input. These data equations serve as rewrite rules for the rewriter used by the prover. The rewriter is described in section 2.3.

The resulting EQ-BDD is stored in the same representation as the input formulas. This means that an EQ-BDD is again a data expression of sort *Bool* that can be regarded as a simplified version of the original formula. For some applications it is useful to substitute data expressions of sort *Bool* with the EQ-BDD equivalent to those expressions. (See chapter 12 for an example of such an application.) To enable the use of the resulting EQ-BDDs, the prover can output them.

The incompleteness of the prover sometimes calls for user interaction, as described in chapter 7. To provide the user with information about the proving process and the resulting EQ-BDD, the prover offers a number of features. It can output the resulting EQ-BDD in order to print it in some human readable format. Furthermore, it can print so called counterexamples and witnesses. A counterexample is a conjunction of guards and negated guards that lead to a leaf labeled "false" in the resulting EQ-BDD. A witness is a conjunction of guards and negated guards that lead to a leaf labeled "true".

The prover can be given a positive integer as input, representing the maximum number of seconds to be spent on proving a single formula. Eventually, there could be a number of provers available for the mCRL2 toolset, each with its own pros and cons. If a large number of formulas needs to be checked and it is unclear which prover handles which type of formulas best, this time limit can be used to prevent that an unsuited prover spends too much time on a certain type of formula. One specific prover can be asked to check all formulas, provided that it only spends a specified number of seconds on proving a single formula. The formulas it cannot prove within the time limit can be passed on to another prover.

Chapter 6

Applications

This chapter describes a number of applications of the prover. These application form the basis of a number of tools based on the prover in the mCRL2 toolset.

6.1 Formulas

The most obvious application of the constructed EQ-BDD based prover is checking whether or not an mCRL2 data expression of sort *Bool* is a tautology or a contradiction. Data expressions can contain maps defined using rewrite rules. The implementation of the EQ-BDD based prover uses a rewriter, as described in section 2.3, which needs these rewrite rules to be able to process the data expressions. Checking data expressions given the corresponding set of rewrite rules is a straightforward application of the algorithm described in section 3.4. We give a number of examples of this application in chapter 7.

6.2 Invariants

Let P be a linear process equation defined as

$$P(d:D) \quad = \quad \sum_{i \in I} \sum_{e_i \in E_i} c_i(d, e_i) \rightarrow a_i(f_i(d, e_i)) \cdot P(g_i(d, e_i));$$

with initial state d_0 and let $\Phi(d)$ be a data expression of type Bool expressing a property of variable d of sort D. We call Φ an invariant of LPE P if the following two conditions hold:

- The property expressed by Φ holds in the initial state, i.e. $\Phi(d_0)$ is a tautology.
- The expression $\Phi(d) \wedge c_i(d, e_i) \Rightarrow \Phi(g_i(d, e_i))$ holds for all $i \in I$.

This method of checking the invariance of a data expression is not only suited for proving properties of LPEs, but can also be used to simplify LPEs. Given an LPE P and a data expression Φ that is an invariant of P, each summand of P whose condition in conjunction with the invariant is a contradiction can be removed.

6.3 Confluence

Linear process equations in mCRL2 can be instantiated to obtain so called *state spaces*. A state space is a set of states combined with a set of transitions. Transitions are pairs of states, corresponding to an action. The process of instantiating an LPE suffers from a problem known as state space explosion. The size of the state space corresponding to an LPE grows exponentially as the

complexity of the LPE increases. Even relatively simple processes can have enormous, possibly infinite, state spaces. The notion of confluence [15] can be used to reduce the size of state spaces. We give a definition of confluence for state spaces as well as LPEs and show how an automated prover can aid in the detection of confluence.

Given a state space with a set T of transitions, we denote the fact that there is a transition corresponding to action a from state s to state t in T by $s \xrightarrow{a}_T t$. Transitions corresponding to a τ action can be confluent. The transition $s \xrightarrow{\tau}_T s'$ is confluent if for all transitions $s \xrightarrow{a}_T s''$ one of the following conditions holds:

- There exists a state s''', and transitions $s' \xrightarrow{a}_T s'''$ and $s'' \xrightarrow{\tau}_T s'''$, or
- $a = \tau$ and $s' = s''$.

This definition of confluent transitions in a state space can be translated to a definition of confluent summands in an LPE. Given an LPE,

$$\begin{aligned} P(d:D) &= \cdots \\ &+ \sum_{e_i:E_i} c_i(d,e_i) \rightarrow a_i(f_i(d,e_i)) \cdot P(g_i(d,e_i)) \\ &+ \cdots \\ &+ \sum_{e_j:E_j} c_j(d,e_j) \rightarrow \tau \cdot P(g_j(d,e_j)) \\ &+ \cdots ; \end{aligned}$$

τ-summand j is called confluent if the following condition holds for all $i \in I$, for all $d:D$, for all $e_i:E_i$ and for all $e_j:E_j$

$$\begin{array}{c} (c_i(d,e_i) \wedge c_j(d,e_j)) \\ \Rightarrow \\ \left(\begin{array}{c} c_i(g_j(d,e_j),e_i) \wedge c_j(g_i(d,e_i),c_j) \\ \wedge \\ f_i(d,e_i) = f_i(g_j(d,e_j),e_i) \\ \wedge \\ g_i(g_j(d,e_j),e_i) = g_j(g_i(d,e_i),e_j) \end{array} \right) \end{array}$$

If action a_i is a τ action as well, the condition is weakened to

$$\begin{array}{c} (c_i(d,e_i) \wedge c_j(d,e_j)) \\ \Rightarrow \\ \left(\begin{array}{c} g_i(g_j(d,e_j),e_i) = g_j(g_i(d,e_i),e_j) \\ \vee \\ \left(\begin{array}{c} c_i(g_j(d,e_j),e_i) \\ \wedge \\ c_j(g_i(d,e_i),c_j) \\ \wedge \\ g_i(g_j(d,e_j),e_i) = g_j(g_i(d,e_i),e_j) \end{array} \right) \end{array} \right) \end{array}$$

These two conditions are stronger than the actual confluence conditions, but these are the best approximations when quantifiers are unavailable. They can be generated automatically given an LPE and can subsequently be passed on to an automated prover to check them. Chapter 11 describes a number of examples of this application.

Chapter 7

Optimizing the set of equations

We give a number of examples in which the EQ-BDD based prover we constructed is unable to determine whether or not an expression is a tautology or a contradiction, given an mCRL2 specification containing the definitions of the functions occuring in that expression. In some cases, adding new equations concerning the functions in those expressions to the mCRL2 specification at hand can aid the prover. The method illustrated below can be summarized as expressing one function in terms of another function, in order to increase the effectiveness of the substitutions applied during the process of translating an expression to an EQ-BDD.

7.1 Lists of odd and even length

Given the specification below, the prover is unable to determine that the expression $odd(a) \wedge even(a)$ is a contradiction, where variable a is of sort $List(D)$, and functions $odd : List(D) \to Bool$ and $even : List(D) \to Bool$ indicate whether or not lists are of odd or even length, respectively. The proving process fails because there are no rewrite rules that can be applied to the original expression or the expressions that are constructed by substituting boolean subexpressions by boolean constants during the creation of the corresponding EQ-BDD.

$$
\begin{array}{ll}
\textbf{sort} & D; \\
\textbf{cons} & d_1, d_2 : D; \\
\textbf{map} & even : List(D) \to Bool; \\
 & odd : List(D) \to Bool; \\
\textbf{var} & d : D; \\
 & l : List(D); \\
\textbf{eqn} & even([]) = true; \\
 & even(d \triangleright []) = false; \\
 & even(d \triangleright l) = odd(l); \\
 & odd([]) = false; \\
 & odd(d \triangleright []) = true; \\
 & odd(d \triangleright l) = even(l);
\end{array}
$$

The situation can be improved by adding a rewrite rule to the definitions of *odd* and *even*, relating the maps in a way that makes the process of substituting subexpressions succesful:

$$
\begin{array}{ll}
\textbf{var} & l : List(D); \\
\textbf{eqn} & odd(l) = if(even(l), false, true);
\end{array}
$$

The rewrite rule above causes the expression $odd(a) \wedge even(a)$ to be rewritten to $if(even(a), false, true) \wedge even(a)$. Replacing the subexpression $even(a)$ by *true* yields an expression that rewrites to *false*, substituting it with *false* also yields an expression that rewrites to *false*.

The corresponding EQ-BDD is now constructed correctly and is equivalent to *false,* indicating that the original expression is a contradiction.

7.2 Tails and empty lists

The expression $q \not\approx [] \Rightarrow d1 \triangleright rtail(q) = rtail(d1 \triangleright q)$ cannot be proven to be a tautology, given the following specification:

$$\begin{array}{ll} \textbf{map} & rtail : List(D) \to List(D); \\ \textbf{var} & d, e : D; \\ & l : List(D); \\ \textbf{eqn} & rtail(d \triangleright []) = []; \\ & rtail(d \triangleright e \triangleright l) = d \triangleright rtail(e \triangleright l); \end{array}$$

Using the same approach as before, we express the map *rtail* in terms of the unary map $\approx q$ that maps lists to booleans. Adding the rewrite rule below solves the problem.

$$\begin{array}{ll} \textbf{var} & d : D; \\ & l : List(D); \\ \textbf{eqn} & rtail(d \triangleright q) = if(q \approx [], [], d \triangleright rtail(q)); \end{array}$$

7.3 A failing attempt

Finding suitable additional rewrite rules in similar situations can be cumbersome, as is illustrated next. Consider the expression $n < 0 \wedge n > 0$, for integer variable n. The prover is again unable to determine that this expression is a contradiction. Expressing the operator $<$ in terms of the operator $\geq$ and subsequently expressing the operator $\geq$ in terms of the operators $>$ and $\approx$ fails.

In our first attempt at finding a set of suitable rewrite rules we investigated the results of using the following rule:

$$\begin{array}{ll} \textbf{var} & n : Int; \\ \textbf{eqn} & n < 0 = if(n \geq 0, false, true) \end{array}$$

This rule has no effect at all, because in the process of translating an mCRL2 specification to an LPE, the 0 occuring in the expression $n < 0$ that is part of the rule is rewritten to a form that differs from the form of the 0 in the subexpression $n < 0$ of the original expression $n < 0 \wedge n > 0$. Left-hand sides of rewrite rules are not rewritten to a normal form, so these left-hand sides may not match any subexpression occuring in expressions that have been rewritten to a normal form. To circumvent this unwanted rewriting behaviour, a new rewrite rule is introduced:

$$\begin{array}{ll} \textbf{var} & n, m : Int; \\ \textbf{eqn} & n < m = if(n \geq m, false, true); \end{array}$$

A rewrite rule that is part of the standard repertoire of mCRL2 causes our expression of interest to be rewritten to $n < 0 \wedge 0 < n$. The previous rewrite rule is applied to both of the conjuncts, yielding the expression $if(n \geq 0, false, true) \wedge if(0 \geq n, false, true)$. Since we again fail to express one operator in terms of the other operator, we investigate the following two rules:

$$\begin{array}{ll} \textbf{var} & n, m : Int; \\ \textbf{eqn} & (n \approx 0) \to n < m = if(n \geq m, false, true); \\ & (n \leq m) \to n < m || n \approx m; \end{array}$$

To rewrite only one of the two subexpressions, the condition $n \approx 0$ is added. The second rule expresses the operator $\leq$ in terms of the operator $<$ and equality. This rewrite rule causes the rewriter to keep rewriting the subexpression $0 < n$ infinitely.

7.4 Conclusion

In conclusion we state that the approach of expressing one function in terms of another function is not infallible. Although it provides the desired results in some cases, it often requires a large amount of human interaction. Chapters 9 and 10 describe alternative methods of increasing the provers capability to handle certain types of expressions, that do not require human interaction.

Chapter 8

SMT solvers

A formula can contain a number of functions that have specific properties. The process of checking if a formula is satisfiable given one or more theories describing the functions used in that formula is called checking satisfiability modulo theories. SMT solvers check the satisfiability of formulas modulo theories, hence the acronym SMT. Chapter 9 describes how SMT solvers are used to increase the power of the prover, by eliminating inconsistent paths from EQ-BDDs.

8.1 Review

On a yearly basis a competition for SMT solvers, called SMT-COMP, is organised. The competition is meant to promote the SMT-LIB standard and its benchmarks [16, 17]. The SMT solvers entering the competition compete in a number of divisions. Each division is associated with a sublogic of the main SMT-LIB logic and consists of a set of so called benchmarks. These benchmarks are large formulas expressed in the language defined by that sublogic. The participants must solve a number of benchmarks of the division they compete in. The rank of an SMT solver is determined by the amount of benchmarks it solves correctly and the time it takes to do so. For each benchmark a solver solves correctly, it is rewarded a point. Points are subtracted for each incorrect answer. If a solver indicates that it is unable to determine whether or not a benchmark is satisfiable it will receive no points for that benchmark. If two solvers have an equal amount of points after processing all benchmarks, the solver that spent the least time on proving benchmarks will get a higher rank.

Table 8.1 shows the rank of each participant in the divisions of the competition. The partic-

	UF	**RDL**	**IDL**	**UFIDL**	**LRA**	**LIA**	**AUFLIA**
Ario	7	5	4	4	5	3	-
BarcelogicTools	1	1	1	1	-	-	-
CVC	5	7	7	5	6	5	2
CVC Lite	4	8	6	8	7	4	3
HTP	10 (X)	10 (X)	10 (X)	10 (X)	4 (X)	8 (X)	5 (X)
Sammy	8	6	8	9 (X)	8 (X)	6	6 (X)
Sateen	-	-	5	-	-	-	-
SBT	9 (X)	-	11 (X)	6 (X)	-	9 (X)	-
Simplics	-	4	-	-	1	-	-
SVC	6	9	9	7	9	7	4
Yices	2	2	2	2	2	1	1
MathSat	3	3	3	3	3	2 (X)	-

Table 8.1: Results of SMT-COMP'05

ipants are listed to the left and abbreviations of the division names are listed at the top of the table. The benchmarks in all of the divisions contain quantifier free formulas, so the actual names of the divisions and their underlying logic are of the form "QF_" followed by the abbreviation in the table. The rank of solvers that gave one or more incorrect answers is marked with an "*X*". A "-" indicates that a solver did not enter the competition in the corresponding division.

All of the tools participating in SMT-COMP read their input in the SMT-LIB format. Chapter 9 discusses an extension of the prover that needs an SMT solver to check the satisfiability of boolean expressions. Converting these expressions to the SMT-LIB format enables the use of all the participants of SMT-COMP for checking their satisfiability. Section 8.2 gives a short description of the SMT-LIB format and the translation from data expressions in the internal mCRL2 format to benchmarks in the SMT-LIB format.

8.2 SMT-LIB

A benchmark in the SMT-LIB format is a tuple $\langle L, A, F, S, ES, EF, EP\rangle$. Elements F and A are formulas in a restriction of the many-sorted first-order logic with equality that is the underlying logic of the SMT-LIB standard. Checking a benchmark boils down to checking the satisfiability of formula F under the assumption of formula A. The element L is the name of the sublogic of the SMT-LIB logic that is used for the benchmark. The sublogic specified by L restricts the logic of the SMT-LIB standard by specifying a background theory and a language. Each theory defines a number of sorts, and a number of functions and predicates over these sorts. The theory "Ints", for instance, introduces a sort *Int,* functions representing *0, 1, unary minus, binary minus, addition* and *multiplication*, and predicates representing *less than*, *less than or equal*, *greater than* and *greater than or equal*. Each sublogic defines a language based on the sorts, functions and predicates of a certain theory. They describe how elements of these sorts, and functions and predicates over these sorts can be combined into formulas using logical connectives, the boolean constants and other constructs from the underlying logic of the standard. The element S indicates the status of a benchmark. The status of a benchmark can be either "satisfiable", "unsatisfiable" or "unknown". The elements ES, EF and EP of a benchmark define additional sorts, functions and predicates, respectively, that can be used in the formulas A and F. These sorts, functions and predicates are all uninterpreted.

We use the tools CVC Lite and ARIO in our implementation of the algorithm *Remove* described in chapter 9. These SMT solvers take a less strict version of the SMT-LIB benchmarks as their input. They are built for use with specific theories and logics and therefore disregard the element L of benchmarks. The element S indicating the status of a benchmark is also disregarded, which means that the benchmarks provided as input for CVC Lite and ARIO are tuples of the form $\langle A, F, ES, EF, EP\rangle$. The theories and logics supported by CVC Lite and ARIO are discussed in the sections 8.3 and 8.4.

The translation from data expressions in the mCRL2 format to benchmarks in the format accepted by CVC Lite and ARIO is straightforward. We will not discuss all the details of the process, but give the general principle instead. The sorts *Int* and *Real* are available in the benchmark format, so variables and constants of these sorts, as well as functions taking these sorts as arguments, do not need any special treatment during translation. The mCRL2 sorts *Pos* and *Nat*, representing positive and natural numbers, respectively, can be simulated by the sort *Int* and the addition of a clause to the translated formula. This clause is added to make the resulting formula unsatisfiable for integer valuations where only valuations using positive or natural numbers are allowed. A variable v of sort *Nat*, for example, is translated to a variable v' of sort *Int.* A clause $v' \geq 0$ is added to the resulting formula, ensuring that it is only satisfiable if v' is instantiated with a natural number. Boolean variables are not allowed in the benchmark format and are therefore simulated using unary predicates over variables of an uninterpreted sort. A boolean variable b' is thus translated to a clause $f(b')$, where b' is variable of an uninterpreted sort and f is a unary predicate over variables of that uninterpreted sort. Operators including those representing *addition*, *subtraction* and *multiplication*, and predicates including those representing *greater than*,

greater than or equal, *smaller than* and *smaller than or equal* are available in the theories and logics supported by CVC Lite and ARIO, so they are directly translated from data expressions in the mCRL2 format to the format of the benchmarks. Functions and predicates that are unavailable are translated to uninterpreted functions and predicates. If an unavailable function ranges over an unavailable sort, the translation simulates the sort using an uninterpreted sort. The same goes for the arguments of unavailable functions or predicates that are of an unavailable sort.

Note that data expressions in mCRL2 format that use higher-order constructs, such as functions returning functions, can not be translated to the benchmark format, as the format only supports first-order constructs.

8.3 CVC Lite

The SMT solver CVC Lite can be used as a command line tool and as a C library. The command line tool can read input in the format of the SMT-LIB standard, as well as input in the native CVC Lite fomat. The C library checks the satisfiability of formulas in the native format only. Translating data expressions in internal mCRL2 format to formulas in native CVC Lite format practically follows the same general procedure as described above for the case of formulas in the SMT-LIB format.

CVC Lite is an automated theorem prover, also referred to as a validity checker. It is called a validity checker because it checks the validity of a formula in a given logical context. The tool has support for integer and real numbers, booleans, uninterpreted functions and types (sorts), arrays, records and bitvectors. It supports the common logical connectives, as well as a number of operators and predicates on integer and real numbers, including those representing *addition*, *subtraction*, *multiplication*, *division*, *less than*, *less than or equal*, *greater than*, *greater than or equal*, *equality* and *inequality*. More information, an executable and the source code of the tool can be found on its website [18].

8.4 ARIO

ARIO is available as a command line tool only. It is limited in comparison to CVC Lite in terms of the formulas it can check. It supports integer and real numbers, the common logical connectives and operators and predicates on real numbers representing *addition*, *subtraction*, *less than*, *less than or equal*, *greater than* and *greater than or equal*. Problems that can be solved by both CVC Lite and ARIO, are solved in less time by ARIO, even in comparison to the C library version of CVC Lite. An executable of the tool is available for download at its website [19].

Chapter 9

Eliminating paths

Since the prover is not complete, it can produce EQ-BDDs containing so called inconsistent paths. To increase the power of the prover, we use SMT solvers to remove these inconsistent paths from EQ-BDDs. This chapter gives a definition of inconsistent paths and presents an algorithm for removing those paths. We use definitions from [1] to define inconsistent paths. Paths are represented by sequences of 0's and 1's and are typically denoted by α, β and γ. These 0's and 1's correspond to edges in the EQ-BDD leading to a *false*-branch or a *true*-branch, respectively. The empty path is denoted by ϵ.

Definition 9.0.1. The set of paths of an EQ-BDD T, $paths(T)$, is defined as follows:

$$\begin{aligned} paths(false) &= \emptyset \\ paths(true) &= \emptyset \\ paths(ITE(g, T_1, T_2)) &= \{\epsilon\} \cup \{1.\alpha \mid \alpha \in paths(T_1)\} \cup \{0.\alpha \mid \alpha \in paths(T_2)\} \end{aligned}$$

Figure 9.1 shows a BDD where the path represented by 0.0.1 is marked grey.

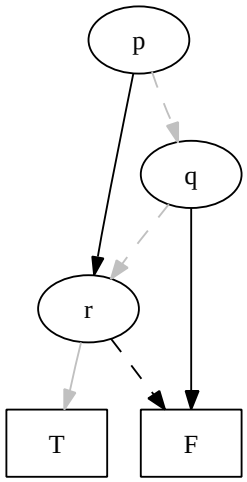

Figure 9.1: A BDD with a marked path

Definition 9.0.2. Given an EQ-BDD T, the guard at the end of path $\alpha \in paths(T)$, written as $T\mid_\alpha$, is defined by:

- $ITE(g, T_1, T_2)\mid_\epsilon = g$.
- $ITE(g, T_1, T_2)\mid_{1.\alpha} = T_1\mid_\alpha$.

- $ITE(g, T_1, T_2) \mid_{0.\alpha} = T_2 \mid_\alpha$.

Definition 9.0.3. Given an EQ-BDD T, the theory up to the node reachable by path $\alpha \in paths(T)$, written as $Th(T, \alpha)$, is defined by:

- $Th(T, \epsilon) = \emptyset$.
- $Th(T, \alpha.1) = Th(T, \alpha) \cup \{T \mid_\alpha\}$.
- $Th(T, \alpha.0) = Th(T, \alpha) \cup \{\neg T \mid_\alpha\}$.

A theory S is called inconsistent if the conjunction of all guards in S is unsatisfiable. Given an EQ-BDD T and a path $\alpha \in paths(T)$, we call α inconsistent if $Th(T, \alpha)$ is inconsistent. A path that is not inconsistent is called consistent.

A way of circumventing the incompleteness of the prover is to remove all inconsistent paths from EQ-BDDs using some method to determine whether or not a path is consistent. The function *Remove* takes an EQ-BDD T and a set of guards and negated guards P as its input and removes all paths α from T for which $Th(T, \alpha) \cup P$ is inconsistent.

Definition 9.0.4. The function *Remove* is defined as follows:

$$
\begin{aligned}
Remove(true, P) &\equiv true \\
Remove(false, P) &\equiv false \\
Remove(ITE(g, T_1, T_2), P) &\equiv \begin{cases}
Remove(T_1, P \cup \{g\}), \\
\quad \textit{if } P \cup \{\neg g\} \textit{ is inconsistent} \\
Remove(T_2, P \cup \{\neg g\}), \\
\quad \textit{if } P \cup \{g\} \textit{ is inconsistent} \\
\overline{ITE}(g, Remove(T_1, P \cup \{g\}), Remove(T_2, P \cup \{\neg g\})), \\
\quad otherwise
\end{cases}
\end{aligned}
$$

The function *Remove* performs a depth-first search of T and checks for each node whether the *true*-branch and the *false*-branch of that node are reachable. When processing a node with *true*-branch T_1, *false*-branch T_2 and label g, the guard g is added to the theory P and it is checked whether or not the resulting theory is consistent. If the theory is inconsistent, the *true*-branch T_1 is unreachable and can be removed. Since the node labelled with guard g does not encode any information if it has only one outgoing edge, the node is replaced with its *false*-branch T_2. A similiar story holds for the *false*-branch, with the exception that the negation of guard g is added to the set P.

Note that there is no case in the final equation of the definition for the situation in which both $P \cup \{g\}$ and $P \cup \{\neg g\}$ are inconsistent. This is because no path $\alpha \in paths(T)$, $|\alpha| \geq 2$, and no guard g exist, such that $Th(T, \alpha) \cup \{g\}$ and $Th(T, \alpha) \cup \{\neg g\}$ are both inconsistent. Finding such a path for $|\alpha| = 2$ boils down to finding boolean expressions a, b and c that satisfy $(c \Rightarrow \neg a) \wedge (\neg c \Rightarrow \neg b) \wedge a \wedge b$. In the BDD shown in figure 9.2, this would mean that both the *true*-branch and the *false*-branch of the node labelled with guard c would be unreachable, because the theories $\{a, b, c\}$ and $\{a, b, \neg c\}$ are both inconsistent.

It can be proved that boolean expressions satisfying the predicate $(c \Rightarrow \neg a) \wedge (\neg c \Rightarrow \neg b) \wedge a \wedge b$ do not exist, i.e. that $(\exists_{a,b,c:Bool} : (c \Rightarrow \neg a) \wedge (\neg c \Rightarrow \neg b) : a \wedge b)$ is a contradiction. The proof of this statement is straightforward and therefore omitted.

The process of translating a formula to an EQ-BDD as described in section 3.4 and subsequently removing all inconstent paths from that EQ-BDD is complete, given a complete method of determining whether or not a path is consistent. In other words, one can prove the following two theorems:

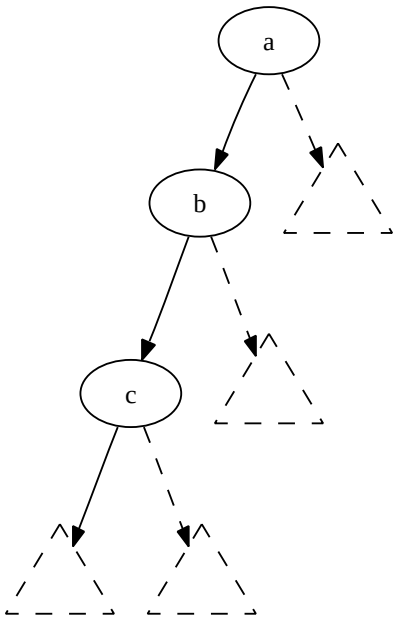

Figure 9.2: A BDD with at least three nodes

Theorem 9.0.5.

1. $\mathit{Remove}(\mathit{EQ\text{-}BDD}(\Phi), \emptyset) \equiv \mathit{true}$, if the formula Φ is a tautology.
2. $\mathit{Remove}(\mathit{EQ\text{-}BDD}(\Phi), \emptyset) \equiv \mathit{false}$, if the formula Φ is a contradiction.

Proof. As both theorems have a similar proof, only the first will be discussed. We prove the following more general statement, from which our proof obligation follows

$$P \text{ is consistent} \Rightarrow \mathit{Remove}(X, P) \equiv \mathit{true},$$

where X is an arbitrary EQ-BDD representing a tautology. If Φ is a tautology, $\mathit{EQ\text{-}BDD}(\Phi)$ can have two forms: it is either equal to *true*, or of the form $\mathit{ITE}(g, T_1, T_2)$ for some guard g, and EQ-BDDs T_1 and T_2. We prove the statement using induction on the structure of EQ-BDDs representing tautologies. This yields two proof obligations: the base case

$$P \text{ is consistent} \Rightarrow \mathit{Remove}(\mathit{true}, P) \equiv \mathit{true}$$

which is trivial, and the induction step

$$P \text{ is consistent} \Rightarrow \mathit{Remove}(\mathit{ITE}(g, T_1, T_2), P) \equiv \mathit{true}$$

The induction step is split in three parts: one where we assume that $P \cup \{g\}$ is consistent and $P \cup \{\neg g\}$ is consistent, one where we assume that $P \cup \{g\}$ is inconsistent and $P \cup \{\neg g\}$ is consistent, and one where we assume that $P \cup \{g\}$ is consistent and $P \cup \{\neg g\}$ is inconsistent. The induction hypotheses are:

- P *is consistent* $\Rightarrow \mathit{Remove}(T_1, P) \equiv \mathit{true}$ and
- P *is consistent* $\Rightarrow \mathit{Remove}(T_2, P) \equiv \mathit{true}$.

Assuming $P \cup \{g\}$ is consistent and $P \cup \{\neg g\}$ is consistent, we derive

$$\begin{array}{ll} & \mathit{Remove}(\mathit{ITE}(g, T_1, T_2), P) \\ \equiv & \{\text{Definition of Remove, } P \cup \{g\} \text{ is consistent and } P \cup \{\neg g\} \text{ is consistent}\} \\ & \overline{\mathit{ITE}}(g, \mathit{Remove}(T_1, P \cup \{g\}), \mathit{Remove}(T_2, P \cup \{\neg g\})) \\ \equiv & \{\text{Induction Hypotheses}\} \\ & \overline{\mathit{ITE}}(g, \mathit{true}, \mathit{true}) \\ \equiv & \{\text{Definition of } \overline{\mathit{ITE}}\} \\ & \mathit{true} \end{array}$$

Assuming $P \cup \{g\}$ is inconsistent and $P \cup \{\neg g\}$ is consistent, we derive

$$\begin{array}{ll} & \mathit{Remove}(\mathit{ITE}(g, T_1, T_2), P) \\ \equiv & \{\text{Definition of Remove, } P \cup \{g\} \text{ is inconsistent and } P \cup \{\neg g\} \text{ is consistent}\} \\ & \mathit{Remove}(T_2, P \cup \{\neg g\}) \\ \equiv & \{\text{Induction Hypotheses}\} \\ & \mathit{true} \end{array}$$

Finally, assuming $P \cup \{g\}$ is consistent and $P \cup \{\neg g\}$ is inconsistent, we derive

$$\begin{array}{ll} & \mathit{Remove}(\mathit{ITE}(g, T_1, T_2), P) \\ \equiv & \{\text{Definition of Remove, } P \cup \{g\} \text{ is consistent and } P \cup \{\neg g\} \text{ is inconsistent}\} \\ & \mathit{Remove}(T_1, P \cup \{g\}) \\ \equiv & \{\text{Induction Hypotheses}\} \\ & \mathit{true} \end{array}$$

□

9.1 Checking consistency

Implementations of the algorithm *Remove* only differ from each other in the way they address the problem of determining whether or not a path is inconsistent. The following subsections describe different ways of using an SMT solver to determine whether or not a path is inconsistent.

9.1.1 Naive approach

A straightforward way of checking the consistency of a path $\alpha \in \mathit{path}(T)$, for some EQ-BDD T, is to construct a conjunction of all guards and negated guards in the corresponding theory $\mathit{Th}(T, \alpha)$. This conjunction can be checked for satisfiability by an SMT solver. If the constructed conjunction is unsatisfiable, the path is inconsistent and can be removed. Two downsides to this approach are the fact that the SMT solver has to be called for each path in the EQ-BDD and the fact that the formulas that serve as input for the SMT solver grow larger as the algorithm is processing nodes deeper in the EQ-BDD.

9.1.2 Minimal sets

A method of reducing the number of calls to the SMT solver is proposed in [10]. If two paths $\alpha \in \mathit{paths}(T)$ and $\beta \in \mathit{paths}(T)$, for some EQ-BDD T, have overlapping theories, i.e. $\mathit{Th}(T, \alpha) \cap$

$Th(T, \beta) \neq \emptyset$, and $Th(T, \alpha) \cap Th(T, \beta)$ itself is inconsistent, the SMT solver does not have to be called to check the consistency of both paths. So called minimal inconsistent sets can be created, containing only those guards and negated guards of an inconsistent theory that cause the inconsistency. These minimal inconsistent sets are, in other words, sets of guards and negated guards that are inconsistent and whose strict subsets are consistent. Only the inconsistency of paths whose theories do not contain any of these minimal sets has to be checked, which decreases the numbers of calls to the SMT solver in some cases.

Constructing the minimal inconsistent sets themselves, however, increases the number of calls to the SMT solver. Each time an inconsistent path is encountered, the powerset of the corresponding theory is created and the consistency of all the elements in the powerset is checked.

9.1.3 Overlap

Adding a single guard g to a consistent theory S can only result in an inconsistent theory if the set of variables used in g and the set of variables used in S overlap. If a theory S has been proved consistent already and the consistency of $S \cup \{g\}$ has to be checked next, it is not always necessary to construct and check the complete conjunction described in section 9.1.1.

Let $Var(g)$ denote the set of all variables used in guard g and let $trans(g, S)$ denote the smallest set satisfying the following two rules:

1. $g \in trans(g, S)$
2. $t \in S \wedge (\cup_{i \in trans(g,S)} Var(i)) \cap Var(t) \neq \emptyset \Rightarrow t \in trans(g, S)$

The consistency of $S \cup \{g\}$ can be checked by checking the consistency of $trans(g, S) \cup \{g\}$. This is an improvement over the naive algorithm, because the set $trans(g, S)$ can be smaller than S. It can even be empty, if S has no variables in common with guard g.

The advantage of this method over the naive method is the fact that the SMT solver is used to check smaller formulas. This advantage will only be noticeable if an SMT solver is used that can check the satisfiability of its input faster if the input is smaller.

9.2 Benchmarks

The performance of the naive algorithm is compared with the performance of the algorithm using minimal sets in [10]. The experiments taken there show that the method using minimal sets does not outperform the naive method when removing inconsistent paths in a number of case studies. To compare the new method described in section 9.1.3 with the existing approaches, we did a number of experiments.

We started by checking the confluence, as described in section 6.3, of all hidden summands in the leader election protocol described in appendix A.1 and the case study concerning the two piles of gold described in appendix A.2. Tabel 9.1 shows the number of seconds it takes to check the confluence for the leader election protocol and tabel 9.2 shows the number of seconds it takes to check the confluence for the case study concerning the piles. In tabel 9.2, the row labelled "piles 6" shows the results for the case where there are six miners and the row labelled "piles 12" shows the results for the case where there are twelve miners. The columns labelled "naive" show the results for the case where paths are eliminated using the naive method and the columns labelled "overlap" show the results for the case where paths are eliminated using the method described in section 9.1.3. Tabel 9.1 shows that checking confluence of the leader election protocol takes less time when inconsistent paths are eliminated using the method described in section 9.1.3. This is not the case for the case study shown in table 9.2.

The difference in performance between the two cases is caused by the amount of reduction of the size of formulas checked by the SMT solvers. The size of the formulas is reduced by 40% for the leader election protocol, whereas the size of the formulas is reduced by only 4% for the

	time (s)
naive	44.436
overlap	64.406

Table 9.1: Checking confluence of the leader election protocol

	ario		cvc lite	
	naive	**overlap**	**naive**	**overlap**
piles 6	1.013	1.032	15.201	15.279
piles 12	1.891	1.930	29.321	30.036

Table 9.2: Checking confluence of the case study concerning piles

specification concerning two piles and six miners, and only 2% for the case concerning twelve miners.

Table 9.3 shows how a reduction in the size of a formula causes a reduction of the amount of time needed by CVC Lite to check that formula. The table lists the average change of the size of formulas and the average change in time spent on checking these formulas, for a number of EQ-BDDs created while checking confluence for the leader election protocol. If the reduction of the size of a formula is only small, the reduction of the amount of time needed to check that formula is also small, as is shown in tables 9.4 and 9.5. Table 9.4 lists the average change in the size of formulas and the average change in time spent by Ario on checking these formulas, for a number of EQ-BDDs created while checking confluence for the case study concerning the two piles and the miners. Table 9.5 lists the same information for CVC Lite.

	change in size (%)	**change in time (%)**
5	-52	-39
6	-44	-33
7	-47	-55
8	-43	-39
9	-47	-48
10	-50	-43
11	-43	-27
12	-47	-41
14	-35	-32
15	-43	-31
16	-43	-42
17	-35	-24

Table 9.3: Reduction of size and time for the leader election protocol

	change in size (%)	**change in time (%)**
18	0	+8
19	-16	-4
20	0	+2
21	0	-2
22	0	+2
23	0	-4
24	-10	+11

Table 9.4: Reduction of size and time for the specification concerning two piles

	change in size (%)	**change in time (%)**
18	0	-4
19	-16	-14
20	0	-3
21	0	-4
22	0	-4
23	0	+7
24	-10	-15

Table 9.5: Reduction of size and time for the specification concerning two piles

Chapter 10

Induction

This chapter introduces the principle of structural induction and shows how our automated prover can apply this principle. After a short introduction of structural induction on one variable, we give an example and show how the prover handles that example. In the final sections, we generalize the principle of structural induction to induction on two or more variables, as some cases cannot be solved using induction on one variable.

10.1 Induction on one variable

In mCRL2, a sort is defined using constructors. Elements of a sort are created by combining these constructors. The following declaration declares a sort S and its two constructors c_0 and c_1.

$$\begin{array}{ll} \textbf{sort} & S; \\ \textbf{cons} & c_0 : S; \\ & c_1 : S \to S; \end{array}$$

Elements of the sort S have the following form: they are either equal to c_0 or equal to $c_1(a)$, for some $a \in S$. To keep things simple, we illustrate the principle of structural induction using the sort S. The generalization of the following discussion and proof to sorts with more constructors or more complex constructors is straightforward.

Theorem 10.1.1.
A statement $X(s)$ holds for all s of sort S, if the following two statements hold:

$$X(c_0)$$

$$X(a) \Rightarrow X(c_1(a)), \textit{ for arbitrary } a \in S$$

The first statement is often called the *base case* and the second statement is often referred to as the *induction step*. The antecedent of the induction step is called the *induction hypothesis*. The proof of a similar claim as part of a more general description of structural induction can be found in [20].

10.1.1 An example of applying induction on one variable

For some sort D, the constructors of the sort $List(D)$ are as follows:

$$\begin{array}{lcl} [] & : & List(D); \\ \triangleright & : & D \to List(D) \to List(D); \end{array}$$

The first constructor indicates that $[]$ is an element of the sort $List(D)$. The second constructor indicates that a new element of $List(D)$ can be created given an element of the sort D and an element of the sort $List(D)$.

The following example shows the declaration of two maps:

$$\begin{array}{rcl} ++ & : & List(D) \to List(D) \to List(D); \\ reverse & : & List(D) \to List(D); \end{array}$$

The rewrite rules defining these two maps are as follows:

$$\begin{array}{rcl} [] ++ s & = & s; \\ (d \triangleright s) ++ t & = & d \triangleright (s ++ t); \\ s ++ [] & = & s; \\ reverse([]) & = & []; \\ reverse(d \triangleright s) & = & reverse(s) ++ [d]; \end{array}$$

where s and t are elements of sort $List(D)$, d is an element of sort D and $[d]$ is shorthand for $d \triangleright []$.

The prover for mCRL2 has been extended so that it can generate the expressions corresponding to the base case and the induction step mentioned in section 10.1. For all expressions of sort $Bool$ containing variables of sort $List(X)$, for some sort X, whose corresponding EQ-BDD is not equal to *true* or *false*, the prover applies structural induction. It generates the expressions corresponding to the base case and the induction step, and converts those expressions to EQ-BDDs. If these EQ-BDDs are equal to *true*, the original expression is a tautology.

We prove the property

$$\langle \forall x, y \in List(D) :: reverse(x ++ y) \approx reverse(y) ++ reverse(x) \rangle$$

of the sort $List(D)$ using structural induction on x and indicate how the prover handles these proofs, starting with the base case:

$$\begin{array}{cc} & reverse([] ++ y) \approx reverse(y) ++ reverse([]) \\ \equiv & \{\text{Definition of } reverse\} \\ & reverse([] ++ y) \approx reverse(y) ++ [] \\ \equiv & \{\text{Definition of } ++\} \\ & reverse(y) \approx reverse(y) \\ \equiv & \{\text{Reflexivity of } \approx\} \\ & true \end{array}$$

Creating the EQ-BDD corresponding to the base case is straightforward. The expression $reverse([] ++ y) \approx reverse(y) ++ reverse([])$ is transformed into *true*, as is done in the first two steps of the derivation above, simply by applying the rewrite rules defining the maps $++$, *reverse* and $\approx$.

We continue with the induction step:

$$
\begin{array}{cc}
 & reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
 & \Rightarrow \\
 & reverse((d \triangleright x) +\!\!+ y) \approx reverse(y) +\!\!+ reverse(d \triangleright x) \\
\equiv & \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
 & \Rightarrow \\
 & reverse(d \triangleright (x +\!\!+ y)) \approx reverse(y) +\!\!+ (reversex) +\!\!+ [d]) \\
\equiv & \{\text{Definition of } reverse\} \\
 & reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
 & \Rightarrow \\
 & reverse(x +\!\!+ y) +\!\!+ [d] \approx reverse(y) +\!\!+ (reverse(x) +\!\!+ [d]) \\
\equiv & \{\text{Associativity of } +\!\!+\} \\
 & reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
 & \Rightarrow \\
 & reverse(x +\!\!+ y) +\!\!+ [d] \approx (reverse(y) +\!\!+ reverse(x)) +\!\!+ [d] \\
\Leftarrow & \{\text{Monotonicity of } +\!\!+\} \\
 & reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
 & \Rightarrow \\
 & reverse(x +\!\!+ y) \approx (reverse(y) +\!\!+ reverse(x)) \\
\equiv & \{\text{Predicate calculus}\} \\
 & true
\end{array}
$$

The first two steps again consist of applying the rewrite rules defining $+\!\!+$ and *reverse*. To complete the third step, an additional rewrite rule has to be introduced, viz.

$$x +\!\!+ (y +\!\!+ z) = (x +\!\!+ y) +\!\!+ z,$$

for x, y and z of sort $List(D)$. The last two steps in the derivation are handled by the prover by converting the formula to an EQ-BDD as described in section 3.4. Taking the expression $reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x)$ as the smallest guard g and the expression

$$
\begin{array}{c}
reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
\Rightarrow \\
reverse(x +\!\!+ y) +\!\!+ [d] \approx (reverse(y) +\!\!+ reverse(x)) +\!\!+ [d]
\end{array}
$$

for Φ, both $\Phi|_g\downarrow$ and $\Phi|_{\neg g}\downarrow$ are equal to *true*, yielding an EQ-BDD equal to *true*. Note that it is important that terms are ordered according to the lexicographical path ordering described in chapter 4. If the term $reverse(x +\!\!+ y) +\!\!+ [d] \approx (reverse(y) +\!\!+ reverse(x)) +\!\!+ [d]$ is chosen as the smallest guard, the constructed EQ-BDD will not be equal to *true*.

10.1.2 Applying induction twice

The previous example does not pose any problems to the prover, as long as induction is applied on variable x. Problems do arise during the induction step if induction is applied on variable y. We give a proof of the property $\langle \forall x, y \in List(D) :: reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \rangle$ using induction on variable y and indicate why this proof is impossible to complete for the prover. The base case is straightforward, so we proceed with the induction step.

$$
\begin{array}{l}
\quad reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
\quad \Rightarrow \\
reverse(x +\!\!+ (d \triangleright y)) \approx reverse(d \triangleright y) +\!\!+ reverse(x) \\
\equiv \quad \{\text{Definition of } reverse\} \\
\quad reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\
\quad \Rightarrow \\
reverse(x +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x)
\end{array}
$$

The story seems to end abruptly for this proof, but we can continue by applying induction on variable x. The base case is again trivial, so we only discuss the induction step.

$$
\begin{array}{l}
\quad \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \Rightarrow \\ reverse(x +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) \end{array} \right) \\
\quad \Rightarrow \\
\left(\begin{array}{c} reverse((e \triangleright x) +\!\!+ y) \approx reverse(y) +\!\!+ reverse(e \triangleright x) \\ \Rightarrow \\ reverse((e \triangleright x) +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(e \triangleright x) \end{array} \right) \\
\equiv \quad \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
\quad \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \Rightarrow \\ reverse(x +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) \end{array} \right) \\
\quad \Rightarrow \\
\left(\begin{array}{c} reverse(e \triangleright (x +\!\!+ y)) \approx reverse(y) +\!\!+ reverse(x) +\!\!+ [e] \\ \Rightarrow \\ reverse(e \triangleright (x +\!\!+ (d \triangleright y))) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) +\!\!+ [e] \end{array} \right) \\
\equiv \quad \{\text{Definition of } reverse\} \\
\quad \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \Rightarrow \\ reverse(x +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) \end{array} \right) \\
\quad \Rightarrow \\
\left(\begin{array}{c} reverse(x +\!\!+ y) +\!\!+ [e] \approx reverse(y) +\!\!+ reverse(x) +\!\!+ [e] \\ \Rightarrow \\ reverse(x +\!\!+ (d \triangleright y)) +\!\!+ [e] \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) +\!\!+ [e] \end{array} \right) \\
\Leftarrow \quad \{\text{Monotonicity of } +\!\!+\} \\
\quad \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \Rightarrow \\ reverse(x +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) \end{array} \right) \\
\quad \Rightarrow \\
\left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \Rightarrow \\ reverse(x +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) \end{array} \right) \\
\equiv \quad \{\text{Predicate calculus}\} \\
\quad true
\end{array}
$$

This proof can not be completed by the prover. The last two steps in the previous proof were handled automatically by converting the formula that remained after unfolding all definitions to an EQ-BDD. This approach fails for this example, because the expression at hand is not of the form $A \Rightarrow B$, where A is the smallest guard of the expression. Instead, replacing the smallest

guard, subexpression $reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x)$, with *false* yields an expression of the form

$$\begin{array}{c} \neg(reverse(x +\!\!+ (d \triangleright y)) \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x)) \\ \Rightarrow \\ \left(\begin{array}{c} reverse(x +\!\!+ y) +\!\!+ [e] \approx reverse(y) +\!\!+ reverse(x) +\!\!+ [e] \\ \Rightarrow \\ reverse(x +\!\!+ (d \triangleright y)) +\!\!+ [e] \approx reverse(y) +\!\!+ [d] +\!\!+ reverse(x) +\!\!+ [e] \end{array} \right) \end{array}$$

that cannot be reduced any further.

10.2 Induction on two variables

We investigate another approach, based on the assumption that applying induction on both variables simultaneously will improve the situation. This alternative approach is also guided by the observation that a formula of the form $A \Rightarrow B$ is equal to *true* if A is equal to *false*. If we construct formulas of this form, such that the antecedent A is a conjunction containing the smallest guard, the construction of corresponding EQ-BDDs is simplified. When the prover constructs the EQ-BDD corresponding to such a formula, this formula will immediately reduce to *true* when the smallest guard is replaced by *false*.

Theorem 10.2.1.
A statement $Y(s_1, s_2)$ holds for all s_1 and s_2 of sort S, if the following statements hold:

$$Y(c_0, c_0) \tag{10.1}$$

$$Y(a, c_0) \Rightarrow Y(c_1(a), c_0), \textit{ for arbitrary } a \in S \tag{10.2}$$

$$Y(c_0, a) \Rightarrow Y(c_0, c_1(a)), \textit{ for arbitrary } a \in S$$

$$\left(\begin{array}{c} Y(a, b) \\ \wedge \\ Y(a, c_1(b)) \\ \wedge \\ Y(c_1(a), b) \end{array} \right) \Rightarrow Y(c_1(a), c_1(b)), \textit{ for arbitrary } a, b \in S \tag{10.3}$$

This theorem is proved by showing that the set of all pairs (x, y) for which the statement $Y(x, y)$ does not hold is empty, provided the previous statements hold.

Proof. Let T be the set of all pairs (x, y) for which the statement $Y(x, y)$ does not hold. We define the function $size : S \to \mathbb{N}$ as follows:

$$\begin{array}{rcl} size(c_0) & = & 0 \\ size(c_1(a)) & = & 1 + size(a) \end{array}$$

Suppose an element $(x, y) \in T$ exists for which there is no element $(x', y') \in T$ such that $size(x) + size(y) > size(x') + size(y')$. This element is not equal to (c_0, c_0), since $Y(c_0, c_0)$ holds according to statement 10.1. It is unequal to $(c_0, c_1(a))$, for some $a \in S$, because $size(c_0) + size(c_1(a)) > size(c_0) + size(a)$ and thus $Y(c_0, a)$ holds, yielding $Y(c_0, c_1(a))$ according to statement 10.2. A similar argument holds for the case $(c_1(a), c_0)$. The only remaining form of the element is $(c_1(a), c_1(b))$, for some $a, b \in S$. Since $size(c_1(a)) + size(c_1(b)) > size(c_1(a)) + size(b)$,

$size(c_1(a)) + size(c_1(b)) > size(a) + size(c_1(b))$ and $size(c_1(a)) + size(c_1(b)) > size(a) + size(b)$, all of the statements $Y(a, b)$, $Y(c_1(a), b)$ and $Y(a, c_1(b))$ hold. According to statement 10.3, $Y(c_1(a), c_1(b))$ holds as well. This proves that the set T is empty and $Y(a, b)$ holds for all $x, y \in S$. □

10.2.1 An example of applying induction on two variables

The following derivations show how this approach is used to prove the property $\langle \forall x, y \in List(D) :: reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \rangle$, starting with the base case:

$$
\begin{array}{ll}
 & reverse([] +\!\!+ []) \approx reverse([]) +\!\!+ reverse([]) \\
\equiv & \quad \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & reverse([]) \approx [] +\!\!+ [] \\
\equiv & \quad \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & [] \approx [] \\
\equiv & \quad \{\text{Reflexivity of } \approx\} \\
 & true
\end{array}
$$

This case is handled by the prover in the same matter as the base case discussed above.

We continue with the first two induction steps:

$$
\begin{array}{ll}
 & reverse(x +\!\!+ []) \approx reverse([]) +\!\!+ reverse(x) \\
 & \quad \Rightarrow \\
 & reverse((d \triangleright x) +\!\!+ []) \approx reverse([]) +\!\!+ reverse(d \triangleright x) \\
\equiv & \quad \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & reverse(x) \approx [] +\!\!+ reverse(x) \\
 & \quad \Rightarrow \\
 & reverse(d \triangleright x) \approx [] +\!\!+ reverse(x) +\!\!+ [d] \\
\equiv & \quad \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & reverse(x) \approx reverse(x) \\
 & \quad \Rightarrow \\
 & reverse(x) +\!\!+ [d] \approx reverse(x) +\!\!+ [d] \\
\equiv & \quad \{\text{Reflexivity of } \approx\} \\
 & true
\end{array}
$$

and

$$
\begin{array}{cc}
 & reverse([] +\!\!+\, y) \approx reverse(y) +\!\!+ reverse([]) \\
 & \Rightarrow \\
 & reverse([] +\!\!+\, e \triangleright y) \approx reverse(e \triangleright y) +\!\!+ reverse([]) \\
\equiv & \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & reverse(y) \approx reverse(y) +\!\!+ [] \\
 & \Rightarrow \\
 & reverse(e \triangleright y) \approx reverse(e \triangleright y) +\!\!+ [] \\
\equiv & \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & reverse(y) \approx reverse(y) \\
 & \Rightarrow \\
 & reverse(y) +\!\!+ [e] \approx reverse(y) +\!\!+ [e] \\
\equiv & \{\text{Reflexivity of } \approx\} \\
 & true
\end{array}
$$

These two cases are again handled like the base case, i.e. by simply applying the rewrite rules defining the maps $+\!\!+$, *reverse* and $\approx$.

We continue with the final induction step:

$$
\begin{array}{ll}
 & \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \wedge \\ reverse((d \triangleright x) +\!\!+ y) \approx reverse(y) +\!\!+ reverse(d \triangleright x) \\ \wedge \\ reverse(x +\!\!+ (e \triangleright y)) \approx reverse(e \triangleright y) +\!\!+ reverse(x) \end{array}\right) \\
 & \qquad \Rightarrow \\
 & reverse((d \triangleright x) +\!\!+ (e \triangleright y)) \approx reverse(e \triangleright y) +\!\!+ reverse(d \triangleright x) \\
\equiv & \qquad \{\text{Definition of } +\!\!+ \text{ and } reverse\} \\
 & \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \wedge \\ reverse(d \triangleright (x +\!\!+ y)) \approx reverse(y) +\!\!+ reverse(x) +\!\!+ [d] \\ \wedge \\ reverse(x +\!\!+ (e \triangleright y)) \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x) \end{array}\right) \\
 & \qquad \Rightarrow \\
 & reverse(d \triangleright (x +\!\!+ (e \triangleright y))) \approx reverse(y) +\!\!+ [e] +\!\!+ reversex) +\!\!+ [d] \\
\equiv & \qquad \{\text{Definition of } reverse\} \\
 & \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \wedge \\ reverse(x +\!\!+ y) +\!\!+ [d] \approx reverse(y) +\!\!+ reverse(x) +\!\!+ [d] \\ \wedge \\ reverse(x +\!\!+ (e \triangleright y)) \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x) \end{array}\right) \\
 & \qquad \Rightarrow \\
 & reverse(x +\!\!+ (e \triangleright y)) +\!\!+ [d] \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x) +\!\!+ [d] \\
\Leftarrow & \qquad \{\text{Monotonicity of } +\!\!+\} \\
 & \left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \wedge \\ reverse(x +\!\!+ y) +\!\!+ [d] \approx reverse(y) +\!\!+ reverse(x) +\!\!+ [d] \\ \wedge \\ reverse(x +\!\!+ (e \triangleright y)) \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x) \end{array}\right) \\
 & \qquad \Rightarrow \\
 & reverse(x +\!\!+ (e \triangleright y)) \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x) \\
\equiv & \qquad \{\text{Predicate calculus}\} \\
 & true
\end{array}
$$

The EQ-BDD corresponding to the expression

$$
\left(\begin{array}{c} reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x) \\ \wedge \\ reverse(x +\!\!+ y) +\!\!+ [d] \approx reverse(y) +\!\!+ reverse(x) +\!\!+ [d] \\ \wedge \\ reverse(x +\!\!+ (e \triangleright y)) \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x) \end{array}\right)
$$
$$
\Rightarrow
$$
$$
reverse(x +\!\!+ (e \triangleright y)) +\!\!+ [d] \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x) +\!\!+ [d]
$$

obtained after applying the rewrite rules defining the maps *reverse* and $+\!\!+$ is constructed using the algorithm described in section 3.4, by first taking $reverse(x +\!\!+ y) \approx reverse(y) +\!\!+ reverse(x)$ as smallest guard and finally $reverse(x +\!\!+ (e \triangleright y)) \approx reverse(y) +\!\!+ [e] +\!\!+ reverse(x)$. This EQ-BDD is equal to *true*.

10.3 A further generalization

The process of proving that an expression of sort *Bool* holds using structural induction on two variables can be extended to proving this fact using induction on an arbitrary number of variables.

Theorem 10.3.1.
A statement $Z(x_1, \ldots, x_n)$ holds for all $x_1, \ldots, x_n \in S$, if

$$Z(c_0, \ldots, c_0)$$

holds and all statements of the form

$$(\bigwedge H) \Rightarrow Z(y_1, \ldots, y_n)$$

hold, where y_i, $0 \leq i \leq n$, equals c_0 or $c_1(a_i)$, for some $a_i \in S$, $\bigwedge H$ holds if all elements of the set H hold, and the set of hypotheses H is the smallest set containing the following elements:

1. The statement $Z(z_1, \ldots, z_n)$, where
   (a) z_i equals a_i for all i, $0 \leq i \leq n$, for which y_i in $Z(y_1, \ldots, y_n)$ equals $c_1(a_i)$, for some $a_i \in S$, and
   (b) z_i is equal to c_0 for all remaining i, $0 \leq i \leq n$.
2. All statements $Z(z_1, \ldots, z_n)$, where
   (a) z_i equals $c_1(a_i)$ for one i, $0 \leq i \leq n$, for which y_i in $Z(y_1, \ldots, y_n)$ equals $c_1(a_i)$, for some $a_i \in S$,
   (b) z_i equals a_i for all other i, $0 \leq i \leq n$, for which y_i in $Z(y_1, \ldots, y_n)$ equals $c_1(a_i)$, for some $a_i \in S$, and
   (c) z_i is equal to c_0 for all remaining i, $0 \leq i \leq n$.

Proof. The proof of this theorem is similar to the one given in section 10.2, so we only sketch the general approach. Given the function $size : S \rightarrow \mathbb{N}$ defined in section 10.2, it can be proven that no $x_1, \ldots, x_n \in S$ exist for which $Z(x_1, \ldots, x_n)$ does not hold, if $Z(x'_1, \ldots, x'_n)$ holds for all $x'_1, \ldots, x'_n \in S$ such that $\sum_{1 \leq i \leq n} size(x_i) > \sum_{1 \leq i \leq n} size(x'_i)$.

The statement $Z(x_1, \ldots, x_n)$ follows from the statement

$$(\bigwedge H) \Rightarrow Z(y_1, \ldots, y_n)$$

and the fact that $Z(z_1, \ldots, z_n)$ holds for each $Z(z_1, \ldots, z_n) \in H$, because $\sum_{1 \leq i \leq n} size(x_i) > \sum_{1 \leq i \leq n} size(z_i)$.

□

If the prover cannot prove that an expression of sort *Bool* holds using induction on one variable, it uses the approach described above to apply induction on a larger number of variables. The prover keeps increasing the number of variables used to apply induction, until the expression at hand can be proven to hold or the maximum number of available variables has been used. If the prover is unable to prove that an expression holds using induction, it attempts to prove that the negation of the expression holds using the same method. If it can be proven using induction that an expression of sort *Bool* holds, the corresponding EQ-BDD is equal to *true*. If the negation of an expression can be proven to hold using induction, the EQ-BDD corresponding to the expression is equal to *false*.

Chapter 11

Comparing old and new

11.1 Speed

To check whether or not the prover for mCRL2 is a worthy successor of the prover for μCRL [21], we checked the confluence of two examples available in both languages and compared the results. Table 11.1 shows the results for the leader election protocol described in appendix A.1. The row labelled "plain" lists the results for the case where the following rules are missing from the specification:

$$\begin{array}{ll} \textbf{var} & d : Nat; \\ & q : List(Nat); \\ \textbf{eqn} & rtail(d \triangleright q) = if(q \approx [], [], d \triangleright rtail(q)); \\ & rhead(d \triangleright q) = if(q \approx [], d, rhead(q)); \end{array}$$

This is the case with the specification of the leader election protocol that is part of the μCRL distribution. It is impossible for both provers to prove the formulas generated by the tools without these equations and without applying induction. The row labelled "additional rules" shows the results for the situation where these rules have been added to the specifications. The columns are labelled with the names of the tools used to check confluence. The tool named "confcheck" is part of the μCRL toolset. The tool named "lpeconfcheck" is its counterpart in the mCRL2 toolset. Note that the number of summands of the LPEs generated by the two toolsets differ because of a slightly different approach in linearization and not because of a difference between the specifications.

	confcheck		**lpeconfcheck**	
	time (s)	**number of summands**	**time (s)**	**number of summands**
plain	2.318	0 out of 70	2.231	0 out of 80
additional rules	2.955	70 out of 70	2.385	80 out of 80

Table 11.1: Leader election protocol

Table 11.2 lists the results for the alternating bit protocol as described in [2]. A specification of this protocol is part of both the distribution of μCRL and the distribution of mCRL2.

confcheck		**lpeconfcheck**	
time (s)	**number of summands**	**time (s)**	**number of summands**
0.825	18 out of 22	0.763	19 out of 23

Table 11.2: Alternating Bit Protocol

The results show that the tool for the mCRL2 toolset is marginally faster than the tool from the

μCRL toolset. It is hard to pinpoint the exact cause of this increase in speed. It could be caused by any number of factors, including the difference in speed between the rewriter for μCRL and mCRL2, the difference in the internal format of the languages, and better compiler optimization enabled by the object oriented implementation of the mCRL2 toolset.

11.2 Proving power

The proving power of the prover for mCRL2 has increased in two ways compared to the prover for μCRL. It can apply induction on lists, as described in chapter 10, and it can eliminate inconsistent paths from EQ-BDDs, as described in chapter 9.

11.2.1 Induction

The ability to apply induction on lists makes it possible to prove the confluence of all τ-summands of the leader election protocol in 3.686 seconds, without the additional equations discussed in section 11.1. Proving confluence takes more time using induction than it takes with the additional rewrite rules, but this increase in time outweighs the amount of time needed for the user to find and add suitable additional rewrite rules in this case and in similar cases.

11.2.2 Eliminating inconsistent paths

The specification describing the behaviour of the miners and the two piles in appendix A.2 contains a number of hidden actions. All summands containing such hidden actions can be proven confluent when eliminating inconsistent paths. Without the elimination of inconsistent paths, the confluence of these summands cannot be proven.

Chapter 12

Conclusions and further work

The ability to apply induction on lists and the elimination of inconsistent paths using SMT solvers have proven to be useful additions to the proving process of this EQ-BDD based prover. Since all standard datatypes are defined inductively, like lists, the power of the prover can still be extended by implementing structural induction on a number of those standard datatypes. This would make the prover stronger in a number of cases and reduces the need for human interaction to minimize the effect of the incompleteness of the prover.

The integration of new SMT solvers, for instance SMT solvers that deal with lists, bags, sets or quantifiers, could enable the detection and removal of currently undetected inconsistent paths.

Since an EQ-BDD is again an expression of sort *Bool*, the process of transforming an expression into an EQ-BDD can be seen as a form of rewriting. The internal representation of sets in mCRL2 uses boolean expressions. An element is part of a certain set if the corresponding boolean expression holds for that element. Two sets can be proven equal by proving that the corresponding predicates are equal. This observation can, for instance, be used when generating state spaces of specifications involving sets.

A final subject of further work would be the implementation of lexicographical path ordering. The current implementation is rather inefficient and could be replaced by a more efficient one, as described in [14].

Bibliography

[1] J. F. Groote and J. van de Pol, "Equational binary decision diagrams," in *Logic Programming and Automated Reasoning*, 2000, pp. 161–178.

[2] J. C. M. Baeten and W. P. Weijland, *Process algebra*. Cambridge University Press, 1990.

[3] J. Groote and A. Ponse, "The syntax and semantics of μCRL," in *Algebra of Communicating Processes, Workshops in Computing, Utrecht, The Netherlands*, A. Ponse, C. Verhoef, and S. van Vlijmen, Eds., pp. 26–62.

[4] mCRL2, July 2006, http://www.mcrl2.org.

[5] J. F. Groote, A. Mathijssen, B. Ploeger, M. Reniers, M. van Weerdenburg, and J. van der Wulp, "Process Algebra and mCRL2," January 2006, IPA Basic Course on Formal Methods 2006.

[6] Y. Usenko, "Linearization in μCRL," Ph.D. dissertation, Technische Universiteit Eindhoven, 2002.

[7] J. C. van de Pol, "JITty: a Rewriter with Strategy Annotations," in *Proceedings of 13th international conference Rewriting Techniques and Applications*, ser. Lecture Notes in Computer Science, S. Tison, Ed., vol. 2378. Springer, 2002.

[8] M. van Weerdenburg, "An account of implementing applicative term rewriting," in *Proceedings of the 6th international Workshop on Rewriting Strategies (WRS '06), ENTCS*, to appear.

[9] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[10] M. Schuijers, "Integrating a BDD prover and a DPLL SAT solver for Abstract Data Types," Master's thesis, Technische Universiteit Eindhoven, 2006.

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

[12] J. W. Klop, "Term rewriting systems," in *Handbook of Logic in Computer Science*. Oxford University Press, 1991, vol. 1.

[13] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier, "Efficient annotated terms," *Software - Practice and Experience*, vol. 30, no. 3, pp. 259–291, 2000.

[14] B. Löchner, "Things to know when implementing lpo." *International Journal on Artificial Intelligence Tools*, vol. 15, no. 1, pp. 53–80, 2006.

[15] J. F. Groote and M. P. A. Sellink, "Confluence for process verification," *Theoretical Computer Science*, vol. 170, no. 1–2, pp. 47–81, 1996.

[16] S. Ranise and C. Tinelli, "The SMT-LIB standard: Version 1.1," 2005. [Online]. Available: http://goedel.cs.uiowa.edu/smtlib

[17] The Satisfiability Modulo Theories Library (SMT-LIB), July 2006, http://goedel.cs.uiowa.edu/smtlib.

[18] CVC Lite Homepage, July 2006, http://www.cs.nyu.edu/acsys/cvcl/.

[19] ARIO SMT solver, July 2006, http://www.eecs.umich.edu/ ario/.

[20] P. A. Fejer and D. A. Simovici, *Mathematical Foundations of Computer Science*. Springer, 1991.

[21] J. C. van de Pol, "A prover for the μCRL Toolset with Applications – version 0.1," CWI, Amsterdam, Tech. Rep. SEN-R0106, 2001.

[22] L. Fredlund, J. F. Groote, and H. Korver, "Formal verification of a leader election protocol in process algebra." *Theoretical Computer Science*, pp. 177:459–486, 1997.

Appendix A

Benchmarks

A.1 A leader election protocol

The folowing mCRL2 specification describes the leader election protocol by Dolev, Klawe and Rodeh [22]. It specifies four components that communicate in a ring. After a finite number of steps, one of the components declares itself the leader. All communication is hidden, leaving only the message indicating that a leader is found intact.

```
map
  submod:  Nat # Nat -> Nat;

var
  n, m:  Nat;

eqn
  submod(0, n) = Int2Nat(n - 1);
  n > 0 -> submod(n, m) = Int2Nat(n - 1);

var
  d:  Nat;
  q:  List(Nat);

eqn
  rtail(d |> q) = if(q == [], [], d |> rtail(q));
  rhead(d |> q) = if(q == [], d, rhead(q));

act
  rq, rp, sp, sq, c:  Nat # Nat;
  leader;

proc
  Relay(i:  Nat, n:  Nat) =
    sum d:  Nat.  rq(submod(i, n), d) .  sq(i, d) .  Relay(i, n);

  Dead(i:  Nat, n:  Nat) =
    delta;

  Q(i:  Nat, q:  List(Nat)) =
    sum d:  Nat.  rp(i, d) .  Q(i, d |> q) +
    (q != []) -> sp(i, rhead(q)) .  Q(i, rtail(q));
```

```
  Active(i: Nat, d: Nat, n: Nat) =
    sq(i, d) .
      (sum e: Nat. rq(submod(i, n), e) .
        ((d == e) ->
          leader . Dead(i, n)
        <>
          sq(i, e) . (sum f: Nat. rq(submod(i, n), f) .
            ((e > max(d, f)) ->
              Active(i, e, n)
            <>
              Relay(i, n)
            )
          )
        )
      );

  Spec =
    hide({c},
      allow({c, leader},
        comm({rp|sq -> c, rq|sp -> c},
          Active(0,3,4) || Q(0,[]) ||
          Active(1,1,4) || Q(1,[]) ||
          Active(2,0,4) || Q(2,[]) ||
          Active(3,2,4) || Q(3,[])
        )
      )
    );

init
  Spec;
```

A.2 Two piles

The following mCRL2 specification formally describes how nuggets of gold are moved from one pile to another by a number of miners. Initially, there is a pile *P1* of 10 nuggets and an empty pile *P2*. The first miner, *F1*, takes an amount of nuggets from pile *P1* and passes it on to the second miner, *F2*. The second miner passes them on to the third miner, the third miner passes them on the fourth miner and the fourth miner passes them on to the fifth miner. Each miner can only hold a certain amount of nuggets in his hands. Some are able to transport 3 nuggets at the same time, whereas another miner can only transport the nuggets one at a time. When a miner has passed on the nuggets in his hands to the next miner, he can receive a new load of nuggets from the miner before him, or, in the case of the first miner, he can take new nuggets from the pile *P1*. The fifth miner, *F5*, adds the nuggets to the second pile, *P2*. When pile *P2* consists of 10 nuggets, a report is sent indicating that the maximum capacity of that pile has been reached.

```
act
  increase, increase1, increase2: Int;
  decrease, decrease1, decrease2: Int;
  report_max, report_max1, report_max2;
  take, take1, take2, take3, take4, take5: Int;
  give, give1, give2, give3, give4, give5: Int;
  take_from_pile1, pass_on_1_to_2, pass_on_2_to_3, pass_on_3_to_4, pass_on_4_to_5, add_to_pile2: Int;

proc
  P(size: Int, max: Int) =
    sum n: Int. (n <= size) -> decrease(n) . P(size - n, max) +
    sum n: Int. (0 < n && size + n <= max) -> increase(n) . P(size + n, max) +
    (size == max) -> report_max . P(size, max);

  F(size: Int, content: Int) =
    sum n: Int. (0 < n && content + n <= size) -> take(n) . F(size, content + n) +
    sum n: Int. (n <= content) -> give(n) . F(size, content - n);
```

```
  P1 = rename({increase -> increase1, decrease -> decrease1, report_max -> report_max1}, P(10, 11));
  P2 = rename({increase -> increase2, decrease -> decrease2, report_max -> report_max2}, P(0, 10));
  F1 = rename({take -> take1, give -> give1}, F(3, 0));
  F2 = rename({take -> take2, give -> give2}, F(2, 0));
  F3 = rename({take -> take3, give -> give3}, F(3, 0));
  F4 = rename({take -> take4, give -> give4}, F(1, 0));
  F5 = rename({take -> take5, give -> give5}, F(2, 0));

init
  hide(
    {take_from_pile1,
     pass_on_1_to_2,
     pass_on_2_to_3,
     pass_on_3_to_4,
     pass_on_4_to_5,
     add_to_pile2
    },
    allow(
      {take_from_pile1,
       pass_on_1_to_2,
       pass_on_2_to_3,
       pass_on_3_to_4,
       pass_on_4_to_5,
       add_to_pile2,
       report_max2
      },
      comm(
        {decrease1|take1 -> take_from_pile1,
         give1|take2 -> pass_on_1_to_2,
         give2|take3 -> pass_on_2_to_3,
         give3|take4 -> pass_on_3_to_4,
         give4|take5 -> pass_on_4_to_5,
         give5|increase2 -> add_to_pile2
        },
        P1 || P2 || F1 || F2 || F3 || F4 || F5
      )
    )
  );
```