Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

A Prover for the µCRL Toolset with Applications
-- Version 0.1 --

J.C. van de Pol

Software Engineering (SEN)

SEN-R0106 April 30, 2001

Report SEN-R0106
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

A Prover for the μCRL Toolset with Applications — Version 0.1 —

Jaco van de Pol
Jaco.van.de.Pol@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
This document describes an automated theorem prover, based on an extension of binary decision diagrams. The prover transforms quantifier-free formulae into equivalent BDD-forms, w.r.t. to some algebraic data specification. The prover is used by four tools for the symbolic analysis of distributed systems specified in μCRL (i.e. process algebra plus algebraic data types). The main techniques are invariants and confluence. Two case studies are reported: the DKR leader election protocol [13], and SPLICE [5], a coordination architecture of industrial origin. In both cases using confluence information leads to a reduced state space.

2000 Mathematics Subject Classification: 03B35, 68M14, 68Q60, 68Q65, 68T15, 68W30
Keywords and Phrases: Binary Decision Diagrams (BDD), Automated Theorem Proving, Verification, Distributed Systems, Confluence, Invariants, Linear Process Operators
Note: Research carried out in SEN 2, with financial support of the "System Validation Center".

Table of Contents

1 Introduction . . . 2
2 Preliminaries . . . 3
2.1 Algebraic Specifications . . . 3
2.2 Formulae and Binary Decision Diagrams . . . 3
2.3 Theorem Proving by Ordering BDDs . . . 5
2.4 Process Specifications and Linear Process Operators . . . 6
3 Description of the Prover . . . 8
3.1 Assumptions on Data Specification . . . 8
3.2 Interface of the Prover . . . 9
3.3 Algorithm . . . 11
3.4 Ordering . . . 12
3.5 Rewriter . . . 14
4 Description of the Tool Suite . . . 15
4.1 Checking a Formula: `formcheck` . . . 16
4.2 Checking an Invariant: `invcheck` . . . 17
4.3 Simplification using Invariants: `invelm` . . . 19
4.4 Checking Confluence: `confcheck` . . . 20
4.5 Error Messages . . . 23
5 Applications . . . 23
5.1 DKR Leader Election Protocol . . . 23
5.2 Splice Coordination Architecture . . . 27
6 Conclusions and Directions for Future Work . . . 30
References . . . 31

1. Introduction

This document is an introduction to a prover supporting the analysis of distributed systems specified in μCRL [20], and four analysis tools based on that prover. The prover is implemented as a library in C and is meant for developers of analysis tools that need theorem prover assistance. The four tools are meant for end-users who want to analyze a particular distributed system specified in μCRL. The prover uses a separate component, the rewriter. Moreover, a subcomponent dealing with a many-sorted signature for open terms can be used separately. The architecture of the system is depicted in Figure 1. The tools (version 0.1) are distributed with the μCRL toolset (version 2.9.0).

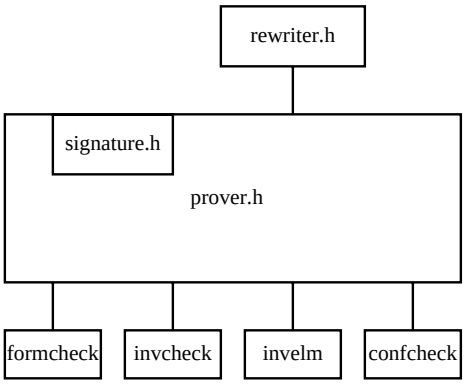


Figure 1: Architecture of the Tool Suite

The main functionality of the prover is to transform a formula into an equivalent binary decision diagram (BDD) w.r.t. some algebraic data specification. The resulting BDD can be `T`, `F`, or something different. In the first two cases, the original formula was a tautology, or a contradiction, respectively. In the third case, we know nothing, due to the incompleteness of the prover. In that case, counter examples and witnesses can be computed from the BDD for diagnostic purposes.

The four tools analyze and modify a μCRL specification in linear process operator format (LPO). An LPO consists of a finite number of summands, each of which is a symbolic representation of a number of state transitions, in terms of action, condition and effect triples. The analysis tools generate certain formulae from an LPO, that represent invariants or confluence properties and check these formulae by means of the prover. Depending on the result they modify the LPO accordingly.

An LPO can be used to generate the full state space. If a summand of the LPO is confluent (i.e. commutes with all other summands), a much smaller state space can be generated [22, 19, 4]. Sometimes, invariant properties are helpful in proving confluence properties. We demonstrate the effectiveness of the prover on confluence and invariance formulae, and the effect of this analysis on the resulting state space, by a number of applications.

In the sequel of this paper we first shortly introduce some preliminaries, such as algebraic specifications (2.1), binary decision diagrams (2.2), and linear processes (2.4). Then we describe the library providing the prover (3). Subsequently, the four analysis tools are described (4). Finally, some applications are described, in order to show and evaluate the applicability of the tools (5). This report is a tutorial and a reference manual at once. The following partial readings are explicitly supported:

Objective	Sections to be read
Tutorial of the tools	2, 4, 5
Tutorial of the prover	2.1, 2.2, 2.3, 3.1, 3.2
Manual of prover	2.1, 2.2, 2.3, 3

2. Preliminaries

All tools work in the context of a μCRL specification, which consists of an algebraic data specification and a process specification. We refer to [20, 16, 28] for a complete description of the μCRL specification language.

2.1 Algebraic Specifications

A multi-sorted algebraic data specification consists of:

- a set of sorts,
- a set of function symbols (divided in constructor and defined symbols),
- a set of equations.

We assume familiarity with the usual rules for term formation and typing, as well as the notions of variables, open terms and closed terms. Figure 2 contains an algebraic specification of the sorts `Bool`, `D` and `Queue` in the language μCRL.

In μCRL some additional requirements and a particular interpretation are put forward. First of all, the sort `Bool` is required, with `T`(rue) and `F`(alse) as its only constructors. It is assumed that `T` and `F` are different.

There is a difference in the interpretation of constructors (functions defined as `func`) and defined symbols (functions defined as `map`). The only assumption is that the domain for sort S is generated by the constructors for sort S. This corresponds to the usual "no junk"-interpretation of the initial algebra. However, opposed to the initial-algebra interpretation, it is not assumed that terms which are not provably equal are actually different.

In particular, it is not assumed that different constructors yield unequal terms, except for `T` and `F`. One way to specify that a and b are distinct, is to have an equation $eq(a, b) = F$. In combination with reflexivity $eq(x, x) = T$, it follows that $a \neq b$. As in the μCRL linearizer, we will assume that if a function `eq:S#S -> Bool` exists, then it denotes equality on sort S.

2.2 Formulae and Binary Decision Diagrams

A *formula* is any term of type `Bool`, possibly containing free variables. An example of a formula in the signature of Figure 2 is: `and(eq(x,in(D1,empty)),if(ne(x),b,not(b)))`, with free variables `b:Bool` and `x:Queue`.

We define an *atom* to be a minimal formula different from `T` and `F`. So, given `b,c:Bool` and `x,y:Queue`, the atoms of `and(b,or(T,c))` are `b, c`, and `and(eq(x,y),eq(x,if(ne(x),x,y)))` contains the atoms `ne(x)` and `eq(x,y)`. In the last example, `eq(x,if(ne(x),x,y))` is not an atom, because it is not minimal.

From now on, we assume that there is some symbol `if:Bool#Bool#Bool->Bool`, satisfying the following equations: `if(T,x,y) = x` and `if(F,x,y) = y`, for all `x,y:Bool`. We also assume that functions with the name `eq` denote equality predicates.

An (extended) *binary decision diagram* (BDD) is defined as a finite ordered binary DAG, whose leaves are labeled with `T` or `F`, and whose internal nodes are labeled by *atoms*. A BDD can be represented as a term of sort `Bool`, by representing the leaves as `T` and `F`, and representing all other nodes by a term `if(atom,bdd,bdd)`. Such terms can be stored in maximally shared form, to get back the DAG.

The BDDs defined above are a generalization of the BDDs introduced by Bryant [7]. The main difference is that Bryant's BDDs are for propositional logic, hence the only atoms allowed are boolean variables. In [18] an extension with equality is introduced, called EQ-BDDs. Here equations between

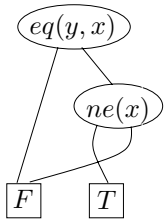
```
sort D
func D1,D2:->D
map  eq:D#D->Bool
     if:Bool#D#D->D
var  d,d1,d2:D
rew  eq(d,d)=T
     if(T,d1,d2) = d1
     if(F,d1,d2) = d2

sort Bool
func T,F:-> Bool
map  and:Bool#Bool -> Bool
     or:Bool#Bool -> Bool
     not:Bool -> Bool
     if:Bool#Bool#Bool->Bool
var  b,c:Bool
rew  and(T,b)=b
     and(F,b)=F
     or(T,b)=T
     or(F,b)=b
     not(F)=T
     not(T)=F
     if(T,b,c)=b
     if(F,b,c)=c

sort Queue
func empty:->Queue              % empty queue
     in:D#Queue->Queue          % add in front
map  ne:Queue->Bool             % non-empty
     toe:Queue->D               % last element
     untoe:Queue->Queue         % queue without last element
     eq:Queue#Queue->Bool
     if:Bool#Queue#Queue->Queue
var  d,e:D q,q1,q2:Queue
rew  ne(empty)=F
     ne(in(d,q))=T
     toe(in(d,empty))=d
     toe(in(d,in(e,q)))=toe(in(e,q))
     untoe(in(d,empty))=empty
     untoe(in(d,in(e,q)))=in(d,untoe(in(e,q)))

     eq(q,q) = T
     eq(empty,in(d,q)) = F
     eq(in(d,q),empty) = F
     if(T,q1,q2) = q1
     if(F,q1,q2) = q2
```

Figure 2: An algebraic data specification in μCRL

variables are also allowed as atoms. This is a special case of the BDDs in this report, which arises when `eq` and `if` are the only function symbols.

The formula `and(ne(x),not(eq(x,y)))`, with `x,y:Queue`, can be represented as a BDD in the following equivalent forms:

1. As the boolean formula: `if(eq(y,x),F,if(ne(x),T,F))`
2. In a picture:



3. In a textual form, as output by the library and the tools of the following sections:

```
BDD: (node): atom --> (high) , (low)
-------------------------------------
(0): F
(1): T
(2): ne(x) --> (1) , (0)
(3): eq(y,x) --> (0) , (2)
-------------------------------------
Total number of nodes: 4
```

Given a model for the algebraic specification and an interpretation of the free variables in that model, such a BDD can be evaluated to either `T`(rue) or `F`(alse). Hence a BDD can be seen as a representation of a set of interpretations, viz. those for which it evaluates to `T`. In order to evaluate the BDD above in a particular model, start in the root node (3), labeled with `eq(y,x)`. If $x = y$ in the model, proceed to node 0, and the BDD evaluates to `F`; otherwise, if $x \neq y$, proceed to node 2. Now if `ne(x)` holds, the BDD evaluates to `T` (node 1), otherwise to `F`.

2.3 Theorem Proving by Ordering BDDs

BDDs have been successfully applied in propositional logic. The key fact found by Bryant [7] is that reduced ordered BDDs are unique. Here *reduced* means that the BDD is maximally shared, and no superfluous tests of the form `if(b,x,x)` exist. A BDD is *ordered* w.r.t. a given total order on atoms, if along each path in the BDD, the atoms occur in strictly increasing order. The consequence of uniqueness of ROBDDs (reduced ordered BDDs) is that equivalence, tautology, contradiction and satisfiability can be checked in constant time on ROBDDs. This can be applied to propositional formulae, by transforming these into ROBDD-form w.r.t. some arbitrary order on the atoms. For propositional logic, this method is complete, irrespective of the chosen order, but the size of the resulting ROBDD is very sensitive to the chosen order.

In EQ-BDDs, uniqueness is spoilt: [18] gives an example of two different but equivalent ordered EQ-BDDs. But it is shown that by imposing some conditions on the ordering between equalities, and applying some equational laws, it is guaranteed that all paths in EQ-ROBDDs (reduced ordered EQ-BDDs) are consistent (see [18] for details). These equational laws are that atoms of the form `eq(x,x)` are forbidden, other atoms of the form `eq(x,y)` are oriented such that $x < y$ in some total

ordering on variables. Finally, in `if(eq(x,y),p,q)`, y may not occur in p. This can be achieved by substituting x for y in p.

As a consequence of consistent paths, the only tautological EQ-ROBDD is `T`, and the only contradictory EQ-ROBDD is `F`. Tautology, satisfiability and contradiction can still be checked in constant time, but the equivalence check is lost. For EQ-BDDs this still gives a complete decision method for tautology, contradiction and satisfiability.

Moreover, in case the formula is not a tautology or a contradiction, one can compute a counterexample and a witness by finding any path in the resulting EQ-ROBDD from the root to the `F` and `T` leaves, respectively. For both paths there exists a model, because all paths are satisfiable. Now the path to the `F` leaf can be seen as a *counterexample* that the formula always holds, and the path to the `T` leaf can be seen as a *witness* that the formula sometimes holds.

In this document, EQ-BDDs are extended by allowing arbitrary function (and relation) symbols, restricting their interpretation by an algebraic data specification. To this specification, we associate a term rewriting system (TRS), by orienting the rules of the algebraic specification. Besides the restrictions on EQ-ROBDDs, we additionally add the requirement that all atoms in an ordered extended BDD are in normal form w.r.t. to the TRS.

At this moment we can only provide experimental support for these extended BDDs, and this notion of ordering. In particular, we have no proof that ordered extended BDDs always exist. Moreover, the paths in these ordered extended BDDs are not always satisfiable. As a consequence, the method is not complete for tautology checking. This shouldn't come as a surprise, because this problem is already undecidable for equations over arbitrary equational theories (known as the word problem). But the method is sound, in the sense that if the resulting BDD is `T` or `F`, then the original formula is indeed tautological or contradictory, respectively. So we have a sound algorithm for tautology and contradiction checking.

As a consequence of the lack of completeness, we don't provide a sound satisfiability check. The witnesses and counterexamples for ordered extended BDDs may be invalid, because there may not exist a model in which they hold. However, if there is such a model in which the counter example holds, then this model makes the whole formula false.

2.4 Process Specifications and Linear Process Operators

In the process part of a μCRL specification, several components can be defined by recursive equations. The main operators are $+, \sum$ (non-deterministic choice), $\cdot$ (sequential composition), $\lhd b \rhd$ (test on boolean). The system is composed by parallel composition ($||$), and hiding/encapsulating internal behaviour (∂,τ). As an example, Figure 3 shows the process specification of two communicating unbounded queues.

In the μCRL toolset [28], a specification will first be linearized. Linearization typically extends the data specification, and transforms the process specification to an LPO (linear process operator). We refer to [28] for an introduction to linearization, and to [21] for a theoretical description and justification. The LPO connected to this example, as generated by the μCRL toolset can be found in Figure 4. It can be reproduced by the following two commands, for linearization and pretty printing:

```
> mcrl -tbfile -regular queue
> pp queue.tbf
```

After linearization, the process specification becomes a single recursive specification with the following components:

- a state vector (a list of variables with their sorts);
- a list of summands, each consisting of:
  - a list of local variables with their sorts,

```
act r,s,r',s',c:D
comm s' | r' = c
proc Q(q:Queue)=
       sum(d:D, r(d) . Q(in(d,q)))
     + s(toe(q)).Q(untoe(q)) <| ne(q) |> delta

     Q1 = rename({s->s'},Q(empty))
     Q2 = rename({r->r'},Q(empty))
     Q  = hide({c}, encap({s',r'}, Q1 || Q2))

init Q
```

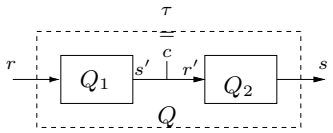


Figure 3: Process specification of parallel queues

```
proc X(q:Queue,q0:Queue) =

sum(d:D,r1(d).
  X(in(d,q),q0) <| T |> delta)+

s2(toe(q0)).
  X(q,untoe(q0)) <| ne(q0) |> delta+

tau.
  X(untoe(q),in(toe(q),q0)) <| and(ne(q),T) |> delta

init X(empty,empty)
```

Figure 4: LPO generated for the parallel queues

  - an action name,
  - a list of action parameters,
  - a next state vector,
  - a condition;

- and an initial state (a list of data terms of the right sorts).

A common naming convention for LPOs is as follows:

$$X(d:D) = \ldots + \sum_{e_i:E_i} a_i(f_i(d,e_i)).X(g_i(d,e_i)) \triangleleft b_i(d,e_i) \triangleright \delta + \ldots$$

with state vector $d : D$, and an i-th summand for the action with (port) name a_i, local variables $e_i : E_i$, action label $f_i(d, e_i)$, next state $g_i(d, e_i) : D$ and condition $b_i(d, e_i) : Bool$. The intended meaning is that in state d, the i-th summand can be executed if there exists some $e_i : E_i$ such that $b_i(d, e_i)$ holds. In this case, the executed action will be $a_i(f_i(d, e_i))$ and the next state will be $g_i(d, e_i)$.

Remarks:

1. In principle, LPOs can have terminating summands, where the next state is missing. Like the existing tools in the toolset, the tools reported in this document neglect this possibility.

2. On default, the linearizer applies so-called *clustering* of summands by default. The algorithm for this introduces new enumerated sorts. The prover doesn't perform case splits on these sorts, so it will miss a number of obvious conclusions. This can be easily solved by providing one of the following options to `mcrl`:

   - `-nocluster`: this avoids clustering at all.
   - `-binary`: this does clustering, but uses boolean encodings instead of enumerated sorts.

   So the "favourite" call to the linearizer becomes `mcrl -tbfile -regular2 -binary spec_file`.

3. Description of the Prover

The main functionality of the prover is to transform a formula into an equivalent BDD according to some algebraic data specification.

3.1 Assumptions on Data Specification

The prover makes some assumptions on the data specification, that the user has to obey. These are the following:

1. The user must specify the following boolean functions, with their usual meaning:

   - `and:Bool#Bool->Bool`
   - `or:Bool#Bool->Bool`
   - `not:Bool->Bool`

2. The user must specify some function with type `Bool#Bool#Bool->Bool`. It is assumed by the prover that this function denotes the if-then-else function. Sometimes this function is called `if`, or, in automatically generated specifications, `C2-Bool`. Use option `-sig-verbose` to see which function symbol is taken as if-then-else.

3. All functions with the name `eq`, and only these, are treated as semantic equality. They should denote the identity-relation in the intended model. The prover implicitly uses reflexivity and symmetry laws, and the substitution rule for `eq`-symbols, even when not present in the original specification.

It is easy to construct specifications that don't satisfy these requirements. Especially, it can not be checked whether the intended interpretation of `eq` is the equality predicate. If this is not the case, the prover will give wrong answers. The presence of the boolean connectives is checked, but it is currently not checked whether the rewrite rules for them respect the intended meaning.

3.2 Interface of the Prover

The μCRL toolset implements data terms as `ATerm`s, defined in the `ATerm`-library [6]. This C-library implements terms, featuring maximal sharing and automatic garbage collection and it also provides hash tables indexed by `ATerms`. The `ATerm` data structure is also used for other data structures in the μCRL toolset, such as (lists of) variable declarations, signatures, data specifications and even a whole linearized specification is represented as a single `ATerm`.

Because `ATerms` are present everywhere, the `ATerm` interface must be included before the prover interface can be included. Similarly, the `ATerm` library must be initialized before the prover library can be initialized.

The most important functions provided by the prover are stated below. The prover is initialized once with a certain data type; this must be the first call to the prover. Free variables can be declared incrementally. The prover can transform formulae into BDD-form, where the formulae are constructed from previously declared variables, and function symbols from the data type.

```
void Init_prover(ATerm datatype,int argc, char *argv[]);
void Declare_vars(ATermList vars);
ATerm Prove(ATerm formula);
ATerm Simplify(ATerm term);
```

Init_prover: initializes the prover with a certain datatype, and passes the options from the UNIX command line to the prover. It automatically calls `RWinitialiseRewriter` and `Init_signature`. The `ATerm` library must be initialized before initializing the `prover`. Re-initialization of the `prover` is not intended, although it might work as expected, except causing memory leaks.

The datatype specification is conform the current μCRL toolset (version 2.9.0), i.e. an `ATerm` of the form `d(<signature>, <equations>)`, where the signature is of the form `s(<sorts>,<funcs>,<maps>)` and the equations are a list of items of the form `e(<vars>,<lhs>,<rhs>)`.

The prover recognizes the following options from `argv`, mainly useful for debugging purposes:

`-pr-test`	writes all case splits (tedious for large examples)
`-pr-statistics`	collects and writes statistic information to `stderr`
`-pr-verbose`	reports information on the number of iterations to `stderr`
`-pr-print`	prints the intermediate BDDs after each iteration to `stderr`

Declare_vars: is used to declare new free variables. Its argument is an `ATermList` of the form `[v(`v_1`,`S_1`), ... ,v(`v_n`,`S_n`),]`, where v_i are variable names with sorts S_i, respectively. S_i must be declared as sorts in the data type, and the variable names must not coincide with constants from the data specification. A variable may be redeclared with a different sort, in which case the old declaration is overwritten.

Prove: is used to transform formulae into BDD-form. when `Prove(formula)` terminates, it yields an equivalent formula in BDD form. The argument is any term of type `Bool`, possibly containing free variables, which should have been declared with `Declare_vars`.

Simplify: is used to simplify a formula by means of rewriting. At the moment it just calls `RWrewrite` of the rewriter with the full specification, but this might change in future versions, by applying only simplifying rules.

The library provides the following functions to inspect a BDD. Their first argument is a BDD, typically constructed by `Prove`. The last argument is the file where the output is written:

```
void print_BDD(ATerm bdd, FILE*);
void count_BDD(ATerm bdd, FILE*);
void print_example(ATerm bdd,ATerm polarity,FILE*);
```

The first prints a BDD, in a format which is linear to the shared size of the BDD. The second function counts and reports the size of a BDD. In `print_example`, the polarity is `T` or `F`, and this function writes some path from the root of the BDD to the leaf indicated by polarity (i.e. either a counterexample or a witness).

Remark. All mentioned functions use terms and declarations in internal toolset format. This format can be created and inspected by the parse- and prettyprint-functions of `signature.h`.

3.2.1 `signature.h`

The representation of μCRL data terms by `ATerms` is not completely straightforward, for instance due to the fact that μCRL allows overloading. The details of this representation are encapsulated in the interface `signature.h`. Its implementation depends on the so-called 2nd-generation format of the μCRL toolset (version 2.9.0).

This interface should be seen as a part of the interface of the prover. Figure 5 lists this interface. Note that `Declare_vars` is in fact part of `signature.h`.

First, there is the initialization `Init_signature`, which is called automatically by `Init_prover`; so it is only needed when `signature.h` is used as a separate component. The following option, from the command-line is recognized by `signature` and can be used to find out the designated if-then-else symbol:

`-sig-verbose` writes info to `stderr`

The signature is initialized once, and free variables can be declared incrementally (see the previous section). Several functions for constructing terms and retrieving information from them are provided. They all work in the context of the initial signature and the current set of declared variables.

The functions that retrieve information on the head symbol of a term are:

- `Sort_of` returns the sort of a symbol
- `Is_var`, `Is_func` and `Is_map` find out whether the head symbol of their argument is a variable, constructor, or defined symbol, respectively. If so, they return the sort of their argument; otherwise they return NULL.
- `Is_eq` and `Is_ite` find out whether the symbol is an equality function or the designated if-then-else symbol, respectively. They return 0 (false) or 1 (true).

```
void Init_signature(ATerm signature, int argc,char *argv[]);
void Declare_vars(ATermList vars);

ATerm Sort_of(ATerm t);
ATerm Is_var(ATerm t);
ATerm Is_func(ATerm t);
ATerm Is_map(ATerm t);
char Is_eq(ATerm t);
char Is_ite(ATerm t);

ATerm prettyprint(ATerm t);
ATerm parse(ATerm t);
ATermList prettyprint_decls(ATermList t);
ATermList parse_decls(ATermList t);

extern ATerm TRUE;
extern ATerm FALSE;
extern ATerm BOOL;

ATerm EQ(ATerm t, ATerm s);
ATerm AND(ATerm t, ATerm s);
ATerm OR(ATerm t, ATerm s);
ATerm IMPLIES(ATerm t, ATerm s);
ATerm NOT(ATerm t);
ATerm ITE(ATerm e, ATerm high, ATerm low);

ATermList generate_reflexivities();
```

Figure 5: Interface `signature.h`

Also some functions to convert terms and variable declarations from input format to internal format are provided. As an example, the external representation `and(b,eq(x,y))` has as its internal counterpart: `"and#Bool#Bool"("b#","eq#Queue#Queue"("x#","y#"))`. Now `prettyprint` transforms a term from internal to external format, and `parse` transforms the other way around. The `prettyprint_decls` and `parse_decls` provide the same functionality on variable declarations.

The sort `Bool` and its constants, and functions for formula formation are provided. Here the `EQ`-function takes into account the type of its arguments. `ITE` is the boolean if-then-else found at initialization.

Finally, a list of reflexivity axioms of the form `e([v(x,sort)],eq(x,x),T)`, for each equality symbol `eq:sort#sort->Bool` can be generated. These rules are needed by the prover.

Remark. Currently, functions to construct signatures, variable declarations, and arbitrary function applications are missing. Such a library should be designed for the whole toolset, not just for the prover.

3.3 Algorithm

We now sketch the algorithm to construct a binary decision diagram from a formula, which is an extension of the algorithm in [18]. The main idea is to create the if-then-else structure by repeated

case splitting. The goal is to generate an *ordered* BDD (see Section 2.3), which is achieved by doing the case-splits in a certain order. Moreover, we require the atoms to be in normal form w.r.t. the TRS associated to the algebraic data type. The actual order on atoms and terms, and the strategy used for term rewriting are parameters of the main algorithm. These issues will be dealt with in subsequent sections.

Given a formula Φ, it is first normalized according to the rewrite system associated with the current data specification, and the equations occurring in Φ are oriented such that in `eq(s,t)`, $s < t$ in a given term ordering. It is assumed that the TRS rewrites every closed formula to `T` or `F`, and that it has at least the following rules:

```
if(T,x,y) = x
if(F,x,y) = y
eq(x,x) = T
```

Moreover, we orient equations, such that `eq(t,s)` rewrites to `eq(s,t)` for all $s < t$.

The result of rewriting and orienting is called $\Phi\downarrow$. Let A be the smallest atom occurring in $\Phi\downarrow$, in the given ordering on atoms. A will be a good candidate for the root of the ordered BDD. Next, two new formulae are formed, $\Phi\downarrow|_A$ and $\Phi\downarrow|_{\neg A}$. Basically, these are obtained by replacing A in $\Phi\downarrow$ by `T` and `F`, respectively. However, if A is of the form $eq(s,t)$, $\Phi\downarrow|_A$ is obtained by replacing t by s. The resulting formulae $\Phi\downarrow|_A$ and $\Phi\downarrow|_{\neg A}$ are recursively transformed into BDDs B_1 and B_2. If $B_1 = B_2$, the test A was redundant. So the complete algorithm becomes:

$$BDD(\Phi) = ITE(A, B_1, B_2),$$

where

$A =$ the smallest atom in $\Phi\downarrow$,
$B_1 = BDD(\Phi\downarrow|_A)$
$B_2 = BDD(\Phi\downarrow|_{\neg A})$
$\Psi\downarrow =$ The normal form of Ψ in the TRS, with equations `eq(s,t)` oriented such that $s < t$.

$$ITE(A, B, C) = \begin{cases} B & \text{if } B = C \\ if(A, B, C) & \text{otherwise} \end{cases}$$

In fact, due to the substitution in case A is an equation, new atoms can emerge, which may be even smaller than A. As a consequence, the resulting BDD is not always ordered. Therefore, the function $BDD()$ is repeated, until a fixed point is reached. In most cases, this fixed point is reached in a small number of iterations, typically 1 or 2. For EQ-BDDs termination is guaranteed. Note that $BDD(\Phi)$ is a formula itself, so $BDD(BDD(\Phi))$ is well-defined.

The algorithm only replaces subterms by equal subterms, so the resulting BDD is always logically equivalent to the original formula w.r.t. the algebraic specification. (Strictly speaking: enhanced with reflexivity, symmetry, transitivity, congruence rules for eq-symbols, and case splits on booleans).

3.4 Ordering

A parameter of the algorithm above is the ordering on atoms and terms. The particular order influences completeness, but it doesn't influence soundness of the prover. We found some effective ordering by experimentation. No theoretical completeness result exists until now. In particular, choosing a different ordering influences both completeness and termination of the algorithm. We now describe the orderings currently used in the `prover` in detail.

3.4.1 Ordering on atoms

This ordering is built from a number of different partial orders. Below we list these orders. They are tried in sequence, until one of them applies. More formally, given $>_1$ and $>_2$, their combination is

defined as:

$$s >_1 t \vee (s \not>_1 t \wedge t \not>_1 s \wedge s >_2 t)$$

In fact this is only well-defined if the constituent orders are total preorders.

1. By type: boolean variables are smaller than equations between variables, which are smaller than other equations, which are smaller than other boolean terms.
2. Equality $eq(a, b)$ is smaller than equality $eq(c, d)$, if either $a < c$ in the term ordering, or $a = c$ and $b < d$ in the term ordering.
   `THIS FEATURE IS CURRENTLY COMMENTED OUT IN THE PROVER.` (See Section 3.4.3).
3. By address: $s < t$ if s viewed as `ATerm`-address is smaller than t as `ATerm`-address. Note that pointer comparison on terms is possible due to maximal sharing and because the `ATerm` garbage collector never relocates a term.

3.4.2 Ordering on terms

This is also a sequential combination of the following sub-orderings:

1. By subterm: $t < s$ if t is a subterm of s.
2. By type: terms starting with a constructor (func) are smaller than terms starting with a defined symbol (map), which are smaller than variables.
3. By address: $s < t$ if the `ATerm`-address of s is the smallest.

This ordering on terms is used to orient a single equality. An equality is ordered, if the right-hand side is greater than the left-hand side. Such an equality is always used from right to left, so in substitutions, the greater term is replaced by the smaller.

There is a problem, as the subterm-ordering is not a total preorder. So in fact it the smallest extension of the subterm-ordering should be taken, which in fact orders terms by depth. As a consequence this relation is not transitive (which is not problematic for orienting equations). If the lexicographic combination wasn't commented out, the term ordering would also be used in comparing two equalities, which would be problematic. We view a neat treatment of this ordering as future work.

3.4.3 Rationale

This section gives a motivation for the chosen orderings. It is mainly based on experiments, and we have no fundamental results yet, except [18] in the case of EQ-BDDs.

Subterm. Consider the equation $eq(f(x), x)$. This would be used to replace all x's by $f(x)$, which doesn't terminate. So the desired orientation is $eq(x, f(x))$. Hence $f(x)$ must be bigger than x. For termination, the subterm ordering must also be the first check in the list of orderings.

In combination with the lexicographic ordering on equations, this has another nice effect: Consider equations $eq(s, t)$ and $eq(s, C[t])$. Since $t < C[t]$, also $eq(s, t) < eq(s, C[t])$. So $eq(s, t)$ occurs above $eq(s, C[t])$, and a substitution gives rise to $eq(s, C[s])$, which rewrites to false for many contexts $C[]$.

Variables bigger than other terms. Substituting a closed term for a variable in a formula eliminates that variable, which makes the formula smaller in some sense. Because the formula becomes more specific, new rewrite rules may match. Typically, if x is replaced by 0 in some formula, some more rewrite rules are applicable. Consider as an example: $x = 0 \rightarrow y + x = y$. Replacing x by 0 triggers the rule $y + 0 = y$.

Distinction constructor/defined symbols. In order to find inconsistencies like `length(l)=2` $\wedge$ `length(l)=3`, we want to replace `length(l)` (defined) by 2 or 3 (constructor). So we orient it like `2=length(l)` $\wedge$ `3=length(l)`. After substitution we get `2=3`, which probably reduces to `F`. This is based on the observation that in many specifications different constructors yield unequal terms.

Furthermore, many rewrite rules match on constructors, so replacing a map by a func probably fires some rewrite rule.

Lexicographic ordering of Equalities. Ordering equalities lexicographically is motivated by [18], where completeness is shown in case equality is the only function symbol. Turning this off can have bad results, like in: `[]=l --> []=untoe(in(d,l))`. In order to use the equality `[]=l`, it must occur above (i.e. be smaller than) `[]=untoe(in(d,l))`.

However, the lexicographic ordering can have bad effects on efficiency for certain formulae, that frequently occur as invariants. Consider a formula of the following shape:

$$(0 = x \vee 1 = x) \wedge (0 = y \vee 1 = y) \wedge (0 = z \vee 1 = z) \wedge \ldots$$

Now, if $0 < 1$ and $x < y < z$, the lexicographic ordering on these guards is:

$$0 = x < 0 = y < 0 = z < 1 = x < 1 = y < 1 = z < \ldots$$

This is exactly the ordering in which the formula above has an exponential BDD! There is a much more efficient ordering, which yields a linear BDD:

$$0 = x < 1 = x < 0 = y < 1 = y < 0 = z < 1 = z < \ldots$$

For this reason, the lexicographic ordering on terms is NOT implemented in the current prover. This makes the prover quicker on a number of examples. The unfortunate effect is that the prover is not complete even when we only have equality as a function symbol.

ATerm addresses. In the end we need a total ordering. The address where a term is stored is unique during a run of the program, and it is easy to compute. Using addresses makes the application sensitive to small changes, like the chosen platform (sgi/i686/sun), or small changes in the formulae. The recursive path ordering over the textual order of function declarations in the specification may be a good alternative, which is currently not implemented.

3.5 Rewriter

3.5.1 Characteristics

The prover uses its own rewriter, and not the rewriter that was available in the μCRL toolset (`libruw`). Compared to `libruw` the new rewriter has the following characteristics:

- It is an interpreter:
  - \+ it has a quick startup;
  - \+ it is easy to add new rules on the fly (this is currently not used);
  - – on the long run it is slower per step than a compiler.
- It uses hashing (memoization): old rewrite results are kept:
  - \+ This is essential for highly shared terms, such as BDDs;
  - – This could result in unpredictably high memory demands.

- It uses a left-sequential strategy:

  + this gives a better termination behaviour, especially for if-then-else terms.

The last reason was regarded important enough to construct a new rewriter. In this strategy, it is first determined which arguments are needed by each rule (due to matching or non-left-linearities). Then, given a term, its arguments are evaluated from left to right in principal, but a rewrite rule is tried as soon as its needed arguments are normalized. See [27] for a complete description and justification of this strategy. Here we explain it with an example:

```
if(T,x,y) = x
if(F,x,y) = y
and(F,x) = F
and(T,x) = x
```

In order to normalize `if(b,s,t)`, the left-sequential strategy first rewrites `b`. In case this rewrites to `T`, `s` is rewritten and the result will be returned, avoiding steps in `t`. If `b` rewrites to `F`, only `t` will be rewritten and the result will be returned, avoiding steps in `s`. Only in case `b` might normalize to some `b'` different from `T` or `F` (probably since it is an open term), both `s` and `t` are rewritten, and the answer `if(b',s',t')` is returned. Similarly, if the first argument of a conjunction rewrites to `F`, the right argument will not be normalized.

Besides saving a number of rewrite steps, this strategy allows rules of the following form:

```
div(x,y) = if(gt(y,x),0,1+div(minus(x,y),y))
```

which is non-terminating with innermost rewriting, but terminating on closed terms with left-sequential rewriting, provided standard definitions of `gt` and `minus` are given.

3.5.2 Interface

Currently, `rewriter.h` provides the following interface:

```
int RWinitialiseRewriter(ATerm datatype, long rewritelimit, int casefunctions);
ATerm RWrewrite(ATerm t);
ATermList RWrewriteList(ATermList l);
void RWflush();
```

The `rewritelimit` and the `casefunctions`-switch are not used, but provided for compatibility with `librw`. `RWflush()` can be used to explicitly empty the hash-table used for memoization, in case memory-usage becomes too high. This is currently done by the prover after each call to `Prove`. Other functions of `librw`, such as those connected to substitutions, are not implemented by the `rewriter`.

4. Description of the Tool Suite

The tool suite currently consists of four tools:

`formcheck`	Check a formula in a data specification
`invcheck`	Check an invariant for an LPO
`invelm`	Simplify an LPO by using an invariant
`confcheck`	Find confluent τ-summands

In the next sections, a description of these tools follows. For all tools, we give a short description, explain the possible options, and give a number of examples. These examples have been tested in the μCRL toolset version 2.9.0. Finally, we give an overview of the error messages in Section 4.5.

4.1 Checking a Formula: `formcheck`

This tools checks whether a certain formula is true in a given data specification.

4.1.1 Usage

```
formcheck [-formula file] [-spec file.tbf] [options]
```

At least one of the `-formula` or `-spec` arguments is required. If only one is given, the other input stream defaults to `stdin`. The following options are recognized:

`-print`	prints the resulting BDDs
`-counter`	prints counterexamples
`-witness`	prints witness for the formulae
`-version`	prints the version number
`-help`	prints a help message

The formula `file` should contain an alternating list of variable declarations and formulae. A variable declaration is of the form `[v(`v_1, S_1`), ... ,v(`v_n, S_n`)]`, where each S_i is a sort in the data type, and the v_i are variable names. The formulae are terms of type `Bool`, whose free variables should be declared in the preceding variable declaration. The specification `file.tbf` should contain a linearized μCRL specification. Only the data part of `file.tbf` is used. A report is written to `stderr`.

4.1.2 Description

This tool checks each formula in the formula `file`, by transforming it into an ordered BDD. It reports for each formula a `True`, `False` or `Don't know` message. The first two messages are generated when the resulting BDD is `T` or `F`. Unlike in propositional BDDs, or in EQ-BDDs, a non-terminal BDD doesn't imply that the formula is satisfiable.

In case of a `Don't know` answer, the resulting BDD can be inspected. With `-print` the BDD is printed, which is not recommended for large BDDs, although the output is linear in the shared size of the BDD. In larger cases `-counter` can be used, which prints a single counterexample to the formula, that can be used for diagnostic purposes. The counterexample is a path in the BDD leading to `F`, consisting of atoms and negated atoms. It is guaranteed that the original formula is false in a model where all elements of the counterexample hold. However, it is not guaranteed that some model of the counterexample exists. Such counterexamples ("false negatives") are manifestations of the incompleteness of the prover. Similarly, `-witness` prints a path to the `T` node.

4.1.3 Limitations

The tool has all limitations of the prover, see Section 3. To mention a few: it is not complete, doesn't apply induction and might not terminate. (The problem is undecidable anyway). Due to incompleteness, counterexamples may be invalid. Possible reactions to a counterexample are: change the formula or add laws to the specification.

4.1.4 Examples

Given the specification of `D`, `Bool` and `Queue` of Figure 2, the formula file can have the following form:

```
[[v(q,Queue),v(d,D)],
 or(not(eq(q,empty)),eq(untoe(in(d,q)),empty)),
 [v(d,D),v(e,D),v(f,D)],
 or(not(or(eq(d,e),eq(d,f))),eq(toe(in(d,empty)),e))]
```

The following dialogue is possible, where > denotes the prompt.

```
> formcheck -spec queue.tbf -formula form
1: True
2: Don't know
> formcheck -counter -spec queue.tbf -formula form
1: True
2: Don't know

FALSE SITUATION:
not(eq(d,e))
eq(d,f)
F
```

Here in the `Don't know` case, a counterexample is printed. The counterexample prints one of the paths to `F`. We see that the formula doesn't hold in case $d = f$ and $d \neq e$. It is easy to construct a model for this situation, so we know that the formula is not valid.

4.2 Checking an Invariant: `invcheck`

`invcheck` checks whether a formula is an invariant of an LPO.

4.2.1 Usage

```
invcheck -invariant file [options]
```

The following options are recognized:

`-generate`	generates formulae to `stdout`, without checking them
`-all`	doesn't terminate as soon as an invariant-violation is found
`-verbose`	indicates to `stderr` for which summands the invariant holds
`-print`	prints the resulting BDDs
`-counter`	prints counterexamples
`-version`	prints the version number
`-help`	prints a help message

When `-generate` is absent, `-verbose` is turned on automatically, otherwise there would be no output at all. Furthermore, `-generate` cannot be used in combination with `-print` or `-counter`.

This tool reads an LPO from `stdin`, and a conjectured invariant from `file`. This file should contain exactly one formula, i.e. a term of type `Bool`. The formula may contain the parameters of the LPO as free variables, but no other variables. `invcheck` reports on `stderr` for which summands the invariant holds.

4.2.2 Description

This tool tries to prove that a given formula is an invariant of the LPO. Let the LPO be given by the following summands

$$X(d:D) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(d,e_i)).X(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta$$

Let the initial state be $init$, and the formula be $\Phi(d)$. Then this tool tries to prove $\Phi(init)$ and for all $i \in I$, the universal closure of $\Phi(d) \wedge b_i(d,e_i) \rightarrow \Phi(g_i(d,e_i))$. The first proves that the invariant

initially holds, the second that it is preserved by all transitions. The applicability of invariants to LPOs dates back to [2].

As output to `stderr`, the tool indicates whether the initial state and which summands satisfy the invariant. The tool terminates as soon as a counterexample is found, unless the `-all` flag is given. The counterexample can be inspected by the `-print` or `-counter` options.

When `-generate` is on, `invcheck` doesn't check the invariance formulae, but merely generates and writes them to `stdout`, in the format understood by `formcheck` (Section 4.1). The main reason is that the formula could be translated and submitted to other theorem provers, like PVS [25] or Isabelle/HOL [26]. Note that this also requires a translation of the data specification.

In default operation the prover is called directly from within `invcheck` for the following reasons: first, the file with all formulae can be very large because the textual representation has no subterm sharing, and second, `invcheck` can stop as soon as a non-conforming summand has been found.

On default, `invcheck` terminates as soon as a summand is detected that doesn't preserve the invariant. With the flag `-all` the tool continues for the other summands. This is useful if one wants to see which summands violate the invariant, or if one desires counterexamples for all summands.

4.2.3 Limitations

Besides the limitations of the prover, it should be noted that `invcheck` doesn't perform induction loading. The invariant should hold "as is". Invariants that need a simultaneous induction proof should be put in conjunction in one formula-file. Furthermore, it is not possible to express invariants containing quantifiers. These can often be avoided by adding more functions to the specification. For instance, the invariant on list L: $\forall l.\, l \in L \rightarrow l < 5$ could be replaced by $\max(L) < 5$, after specifying the function max which returns the maximum value in a list.

4.2.4 Examples

Consider the specification of the queues in Figures 2 and 3. Notice that the LPO (Figure 4) has parameters q and $q0$. We now try to prove the wrong statement that $q0$ is always empty, so we put the formula `eq(q0,empty)` in the file `queue_inv`. Then we get the following dialogue:

```
> mcrl -regular -tbfile queue
> invcheck -counter -invariant queue_inv < queue.tbf
Init OK ..X
No! (summand 3 violates invariant)

FALSE SITUATION:
eq(empty,q0)
ne(q)
F
```

First the specification is linearized (`mcrl`). Then the invariant is checked. It holds initially, and it is preserved by the first two summands, but not by the third one. The counterexample is valid. It shows exactly under which conditions the invariant is violated.

Below an alternative dialogue is given, where the formulae are first generated by `invcheck`, and then checked by `formcheck` and a diagnostic BDD is required. Note that in this case `formcheck` finds the formulae on `stdin`.

```
> mcrl -regular -tbfile queue
> invcheck -generate -invariant queue_inv < queue.tbf | \
     formcheck -print -spec queue.tbf
1: True
2: True
```

```
3: True
4: Don't know

BDD: (node): atom --> (high) , (low)
------------------------------------
(0): F
(1): T
(2): ne(q) --> (0) , (1)
(3): eq(empty,q0) --> (2) , (1)
------------------------------------
Total number of nodes: 4
```

This shows that the counterexample above was the only branch to the `F` leaf.

4.3 Simplification using Invariants: `invelm`

This tool uses an invariant to eliminate unreachable summands. It can also be used to transform the guards into BDD-form.

4.3.1 Usage

```
invelm [options]
```

The following options are recognized:

`-invariant file`	an invariant is used, but NOT checked
`-simplify`	rewrites guards into simplified form
`-summand <n>`	only eliminates/simplifies the n'th summand
`-witness`	provides a witness for summands that seem to be reachable
`-silent`	no output on `stderr`
`-version`	prints the version number
`-help`	prints a help message

This tool reads an LPO from `stdin` and writes an LPO to `stdout`. It optionally reads a formula from an invariant-file. This formula should be a term of type `Bool` in the signature of the input LPO, which may contain parameters of the LPO as free variables, but no other free variables. It is assumed that this formula is an invariant of the LPO.

If `-invariant` is missing, it is set to `T`(rue) by default. This is especially useful with the flag `-simplify`.

4.3.2 Description

When the guard of some summand is false, it will never be executed. But even when the guard of a summand is satisfiable, it might happen that all states where the guard holds are unreachable. Such a summand will never be executed either. Given an invariant Φ, approximating the set of reachable states, `invcheck` can eliminate such summands. Given an LPO

$$X(d:D) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(d,e_i)).X(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta$$

and an invariant Φ, it removes all summands i such that $\Phi(d) \wedge b_i(d,e_i)$ is provably equivalent to `F`.

If the guard is different from `F`, it is kept unaltered, except when the `-simplify` option is used. In that case, the condition $b_i(d,e_i)$ is replaced by the BDD of $\Phi(d) \wedge b_i(d,e_i)$. This might be useful, for

instance, if by the invariant a certain parameter has some constant value. After `invelm -simplify`, `constelm` [17] will detect this fact and replace this parameter by that constant value. In the case of a vacuous invariant, `invelm -simplify` just transforms all guards of an LPO into BDD-form, removing all summands with unsatisfiable guards. With `-summand <n>` the operation can be restricted to a single summand.

Remark. Note that it is not checked whether the provided invariant actually holds. Application of `invelm` is only sound after checking the invariant, for instance by using `invcheck` (Section 4.2).

4.3.3 Limitations

This tool is incomplete, because the invariant might also be true for some unreachable states. Even with an appropriate invariant, the prover might not find the contradiction, which can be inspected by using `-witness`.

4.3.4 Examples

In the queue-example, none of the summands can be removed without changing the behaviour of the system. We can however show the danger of using the invalid invariant `eq(q0,empty)`:

```
> invelm -invariant queue_inv < queue.tbf > /dev/null
.
summand 2 removed.
```

The false invariant contradicts the guard of the second summand (Figure 4). So `invelm` removes this summand. See Section 5.2 for a useful application of `invelm`.

4.4 Checking Confluence: `confcheck`

This tool checks which τ-summands are confluent w.r.t. all other summands, and marks these on request.

4.4.1 Usage

```
confcheck [options]
```

The following options are recognized

`-mark`	writes LPO, marking confluent `tau`-summands as `ctau` to `stdout`
`-generate`	generates formulae without checking to `stdout`
`-invariant file`	the formula in `file` is used as invariant
`-summand <n>`	checks confluence of n'th summand only (should be a tau)
`-all`	doesn't terminate on detection of the first non-confluence
`-silent`	suppresses information on confluent taus to `stderr`
`-print`	prints resulting BDDs
`-counter`	prints counterexamples
`-version`	prints the version number
`-help`	prints a help message

`-silent` may only be present in combination with `-mark` or `-generate`, otherwise there is no output at all. Furthermore, `-generate` and `-mark` cannot be used together, because `confcheck` cannot know which summands to mark if it doesn't check the formulae itself. Finally, `-generate` together with `-counter` or `-print` is meaningless.

This tool reads a linearized μCRL specification from `stdin`. It is assumed that for all parameter sorts of the LPO an `eq`-symbol is declared and defined to be real equality. On request, `confcheck` writes an LPO to `stdout`. It optionally reads an invariant from the invariant-file; this should be a term of type `Bool`, whose free variables are among the LPO parameters. Information on confluent τ-summands is written to `stderr`, unless `-silent`.

4.4.2 Description

`confcheck` checks which τ-summands are confluent. Optionally, with `-generate` it only generates the required formulae, to be checked by another prover. With `-mark` it outputs the same LPO, replacing the confluent `tau`-summands by `ctau`. This information can be used by a state space generator, see [4] and the applications in Section 5. Finally, invariants can be used (by `-invariant`) in order to facilitate theorem proving. These invariants must be checked separately, for instance by `invcheck` (see Section 4.2).

In default operation, `confcheck` terminates for each tau-summand, as soon as another summand is detected which is not confluent with the former. With the `-all` flag, `confcheck` will continue, which is useful if one wants to know all summands that a certain tau-summand doesn't commute with. With `-summand n` the operation can be restricted to a single summand.

Confluence was introduced in the LPO-setting in [22]. In [19], partial confluence was introduced, and used as a reduction technique on state graphs. Next, [4] considered partial confluence at LPO level, in the sense that each τ-summand can be either confluent or not. There also a method is described, to use confluence information during the generation of state spaces, resulting in smaller, but still branching bisimilar, state spaces. Given an LPO,

$$\begin{array}{lcl} X(d:D) & = & \ldots \\ & + & \sum_{e_i:E_i} a_i(f_i(d,e_i)).X(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta \\ & + & \ldots \\ & + & \sum_{e_j:E_j} \tau.X(g_j(d,e_j)) \lhd b_j(d,e_j) \rhd \delta \\ & + & \ldots \end{array}$$

τ-summand j is called *confluent*, if it commutes with all summands i (including j), see Figure 6. We approximate this by the following formula: for all $i \in I$, and $\forall d:D.\forall e_i:E_i.\forall e_j:E_j$,

$$\left(\begin{array}{ll} & Inv(d) \\ \wedge & b_i(d,e_i) \\ \wedge & b_j(d,e_j) \end{array}\right) \rightarrow \left(\begin{array}{ll} & b_i(g_j(d,e_j),e_i) \\ \wedge & b_j(g_i(d,e_i),e_j) \\ \wedge & eq(f_i(d,e_i),f_i(g_j(d,e_j),e_i)) \\ \wedge & eq(g_i(g_j(d,e_j),e_i),g_j(g_i(d,e_i),e_j)) \end{array}\right)$$

If a_i happens to be τ itself, this formula can even be weakened to the following:

$$\left(\begin{array}{ll} & Inv(d) \\ \wedge & b_i(d,e_i) \\ \wedge & b_j(d,e_j) \end{array}\right) \rightarrow \left(\begin{array}{ll} & eq(g_i(d,e_i),g_j(d,e_j)) \\ \vee & \big(\quad b_i(g_j(d,e_j),e_i) \\ & \wedge \quad b_j(g_i(d,e_i),e_j) \\ & \wedge \quad eq(g_i(g_j(d,e_j),e_i),g_j(g_i(d,e_i),e_j))\big) \end{array}\right)$$

The situation is depicted in Figure 6, where the nodes denote states and the arrows transitions, labeled with action/condition pairs.

4.4.3 Limitations

First, `confcheck` uses `eq` symbols for all parameters occurring in the state vector, so these should be declared and specified by the user. It is assumed that these symbols denote equality.

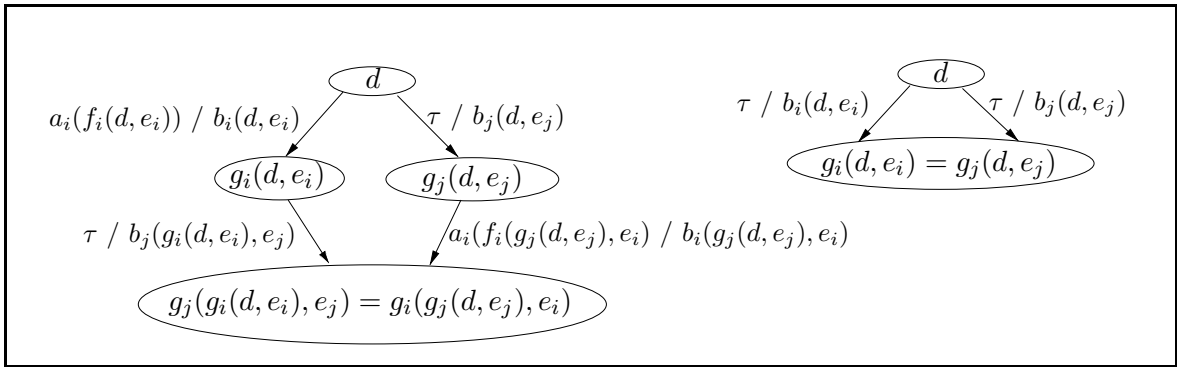


Figure 6: Confluence diagrams

Since the prover currently doesn't deal with existential quantifiers at all, the generated confluence formulae are more restricted than necessary. Ideally, the formulae have an extra existential quantification, cf. [22], as follows: $\forall d : D.\forall e_1 : E_i.\forall e_2 : E_j.\exists e_3 : E_i.\exists e_4 : E_j$.

$$\begin{pmatrix} & Inv(d) \\ \wedge & b_i(d,e_1) \\ \wedge & b_j(d,e_2) \end{pmatrix} \rightarrow \begin{pmatrix} & b_i(g_j(d,e_2),e_3) \\ \wedge & b_j(g_i(d,e_1),e_4) \\ \wedge & eq(f_i(d,e_1),f_i(g_j(d,e_2),e_3)) \\ \wedge & eq(g_i(g_j(d,e_2),e_3),g_j(g_i(d,e_1),e_4)) \end{pmatrix}$$

4.4.4 Examples

Given the specification of Figure 2 and 3, we can try to prove confluence as follows:

```
> confcheck <queue.tbf
tau summand  3:Not confluent with summand 1
> confcheck -counter < queue.tbf
tau summand  3:Not confluent with summand 1

FALSE SITUATION:
eq(in(toe(q),q0),in(toe(in(d_a1,q)),q0))
not(eq(in(d_a1,untoe(q)),untoe(in(d_a1,q))))
ne(q)
F
```

Confluence of the τ-summand could not be proved, but the given counterexample is not convincing. Of course, for non-empty q, we have: `untoe(in(d,q)) = in(d,untoe(q))`, but the prover doesn't notice. In fact, this law can be proved from the specification only by using induction. As the prover doesn't use induction, we have to specify the information above algebraically:

```
toe(in(d,q)) = if(ne(q),toe(q),d)
untoe(in(d,q)) = if(ne(q),in(d,untoe(q)),empty)
```

After adding the above rules to the specification, we try again:

```
> mcrl -tbfile -regular queue_rules
> confcheck < queue_rules.tbf
tau summand  3:...Confluent with all summands!
```

Note that we now proved confluence of an infinite state protocol, as the queue is unbounded.

4.5 Error Messages

Below is a list of error messages that occur in one or more of the tools described in this section.

1. `Specify at least one of -formula or -spec`
2. `Invariant file required '-invariant file'`
3. `The -formula file should contain a list of declarations/formulae`
4. `Variable declaration list [v(x,sort),...]  expected`
5. `Variable declaration v(x,sort) expected.`
6. `readFromTextFile:  parse error at line -, col -.`
   This denotes a syntax error in the formula file (e.g. forgotten bracket).
7. `Function or variable f#type not found.`
   The symbol `f` with arguments of type `type` has not been declared.
8. `No if-then-else symbol found.`
   There must be some symbol of type `Bool#Bool#Bool -> Bool`, which is a requirement of the prover (see Subsection 3.2.1).
9. `Equality on sort "type" is not declared`
   `confcheck` needs the equality on all types of LPO parameters
10. `Summand <n> is not a tau-summand.`
    For confcheck, the `-summand <n>` option must indicate a tau-summand.

5. Applications

In this section we show how the prover can be embedded in the verification methodology of the μCRL toolset. In the current approach, a μCRL specification is first linearized to an LPO (Linear Process Operator). The operational semantics of such an LPO gives rise to an LTS (Labeled Transition System), representing the whole state space. At the level of an LTS various verification techniques exist, such as visual inspection, or model checking [9, 14].

However, the LTS is often too large to generate and investigate; it may even become infinite. Therefore, several analysis tools at the level of the LPO exist that simplify the LPO with techniques from static analysis [17]. The simplified LPO leads to a branching bisimilar, but hopefully smaller LTS. By providing theorem proving assistance, we add powerful tools for this reduction. By using `confcheck` one can mark the confluent τ-summands. The enhanced instantiator (along the lines of [4]) uses this information in order to generate a smaller but branching bisimilar state space. See Figure 7 for an overview of this verification methodology.

In the subsequent sections, we show the success of this combination of symbolic and brute force verification approaches, by showing the different sizes of the state spaces generated before and after symbolic analysis of the LPO, for two examples.

5.1 DKR Leader Election Protocol

In this section, we apply the confluence technique in the verification of the leader election protocol of Dolev, Klawe and Rodeh [13]. The full specification of DKR is in the μCRL toolset distribution. A correctness proof, involving manual linearization, was given in [15].

When trying `confcheck` on the file `DKRleader2`, as included in the μCRL standard distribution, the tool will complain about the absence of an if-then-else function and appropriate eq-functions.

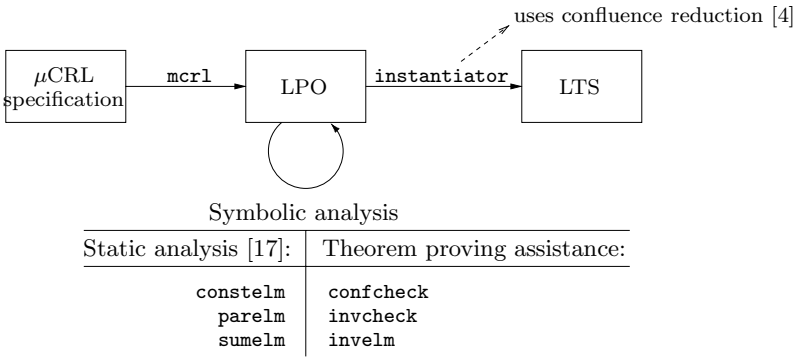


Figure 7: Verification methodology of μCRL toolset

Therefore we add the if-then-else on the booleans, as well as equality on `Table`, which occurs as a parameter type:

```
map  if:Bool#Bool#Bool->Bool
var  b,b1,b2:Bool
rew  if(T,b1,b2) = b1
     if(F,b1,b2) = b2
     if(b,b1,b1) = b1

map  eq:Table#Table->Bool
var  d,e,s,j:nat
     TB:Table
     q: Queue
rew  eq(emt,emt) = T
     eq(emt,in(j,d,e,s,q,TB)) = F
     eq(in(j,d,e,s,q,TB),emt) = F
```

The type `Table` is introduced during manual linearization [15]. This table consists of the values of the variables `d,e,s:Nat` and `q:Queue`, for each component `j`.

5.1.1 Adding Equations for Open Terms

Let us first try to use `confcheck` on the first summand. We get the following report:

```
> mcrl -tbfile -nocluster -regular2 DKRleader2
> confcheck -counter -summand 1 < DKRleader2.tbf
tau summand  1:Not confluent with summand 1
FALSE SITUATION:
...
eq(S(S(S(S(S(0))))),get_s(j_a1,TB))
eq(S(S(S(S(S(0))))),get_s(j_tau1,TB))
eq(S(S(S(S(S(0))))),
   get_s(j_a1,upd_q(... , j_tau1,
                upd_s(S(S(S(S(0)))),j_tau1,TB))))
not(eq(S(S(S(S(S(0))))),
   get_s(j_tau1,upd_q(... , j_a1,
                 upd_s(S(S(S(S(0)))),j_a1,TB)))))
```

```
...
F
```

The counterexample can be refuted as follows: Taking into account that `upd_q` doesn't change the `s`-value, the last two clauses are equivalent to:

```
eq(S(S(S(S(S(0))))),
   get_s(j_a1,upd_s(S(S(S(S(0)))),j_tau1,TB)))
not(eq(S(S(S(S(S(0))))),
       get_s(j_tau1,upd_s(S(S(S(S(0)))),j_a1,TB))))
```

Now there are two cases: Either `eq(j_a1,j_tau1)` holds, but then the last two clauses contradict each other. Or, alternatively, `not(eq(j_a1,j_tau1))`, but then the `upd_s` has no effect and the last two clauses can be simplified further to:

```
eq(S(S(S(S(S(0))))), get_s(j_a1,TB))
not(eq(S(S(S(S(S(0))))), get_s(j_tau1,TB)))
```

The last clause now contradicts the third clause of the counterexample shown above.

This analysis shows that the trace is not a valid counterexample. The prover couldn't detect this, because appropriate rewrite rules relating the various `get_x` and `upd_y` functions are missing. So we add the following rules to the specification:

```
var  v,i,j:nat
     TB:Table
     q: Queue
     b : Bool

rew  get_d(i,upd_d(v,j,TB)) = if(eq(i,j),v,get_d(i,TB))
     get_e(i,upd_e(v,j,TB)) = if(eq(i,j),v,get_e(i,TB))
     get_s(i,upd_s(v,j,TB)) = if(eq(i,j),v,get_s(i,TB))
     get_q(i,upd_q(q,j,TB)) = if(eq(i,j),q,get_q(i,TB))

     get_d(i,upd_e(v,j,TB)) = get_d(i,TB)
     get_d(i,upd_s(v,j,TB)) = get_d(i,TB)
     get_d(i,upd_q(q,j,TB)) = get_d(i,TB)
     get_e(i,upd_d(v,j,TB)) = get_e(i,TB)
     get_e(i,upd_s(v,j,TB)) = get_e(i,TB)
     get_e(i,upd_q(q,j,TB)) = get_e(i,TB)
     get_s(i,upd_d(v,j,TB)) = get_s(i,TB)
     get_s(i,upd_e(v,j,TB)) = get_s(i,TB)
     get_s(i,upd_q(q,j,TB)) = get_s(i,TB)
     get_q(i,upd_d(v,j,TB)) = get_q(i,TB)
     get_q(i,upd_e(v,j,TB)) = get_q(i,TB)
     get_q(i,upd_s(v,j,TB)) = get_q(i,TB)
```

After adding these rules, we try again:

```
> confcheck -counter < DKRleader2.tbf
tau summand  1:Not confluent with summand 1

FALSE SITUATION:
not(eq(upd_q(qin(get_d(j_a1,TB),get_q(j_a1,TB)),j_a1,
        upd_s(S(S(S(S(0)))),j_a1,
```

```
         upd_q(qin(get_d(j_tau1,TB),get_q(j_tau1,TB)),j_tau1,
          upd_s(S(S(S(S(0)))),j_tau1,TB)))),
       upd_q(qin(get_d(j_tau1,TB),get_q(j_tau1,TB)),j_tau1,
        upd_s(S(S(S(S(0)))),j_tau1,
         upd_q(qin(get_d(j_a1,TB),get_q(j_a1,TB)),j_a1,
          upd_s(S(S(S(S(0)))),j_a1,TB))))))
```

This counterexample is also invalid, because the order in which various updates are performed can often be commuted. Note that adding commutative laws easily spoils termination. These considerations give rise to the addition of the following rules, ordering updates on different variables, and deleting overwritten updates:

```
var  d,e,s,v,v',i,j:nat
     TB:Table
     q,q': Queue

rew  upd_d(d,j,upd_s(s,i,TB)) = upd_s(s,i,upd_d(d,j,TB))
     upd_e(e,j,upd_s(s,i,TB)) = upd_s(s,i,upd_e(e,j,TB))
     upd_q(q,i,upd_s(s,j,TB)) = upd_s(s,j,upd_q(q,i,TB))
     upd_d(d,j,upd_q(q,i,TB)) = upd_q(q,i,upd_d(d,j,TB))
     upd_e(e,j,upd_q(q,i,TB)) = upd_q(q,i,upd_e(e,j,TB))
     upd_e(e,i,upd_d(d,j,TB)) = upd_d(d,j,upd_e(e,i,TB))

     upd_d(v,i,upd_d(v',i,TB)) = upd_d(v,i,TB)
     upd_e(v,i,upd_e(v',i,TB)) = upd_e(v,i,TB)
     upd_s(v,i,upd_s(v',i,TB)) = upd_s(v,i,TB)
     upd_q(q,i,upd_q(q',i,TB)) = upd_q(q,i,TB)
```

This doesn't completely solve the problem, because two different updates of the same variable with different indices could still occur. These are exchanged by the following magical rule:

```
var  s,s',i,j:nat
     TB,TB':Table

rew eq(upd_s(s,i,upd_s(s',j,TB)),upd_s(s',j,upd_s(s,i,TB')))
  = if(eq(i,j),
       eq(upd_s(s,i,TB),upd_s(s',j,TB')),
       if(eq(TB,TB'),
          T,
          eq(upd_s(s,i,upd_s(s',j,TB)),
             upd_s(s,i,upd_s(s',j,TB')))))
```

Similar rules are added for the `d,e,q`-variables. Additionally, some other rules are added, also after considering the generated counterexamples.

```
var  d,i,j,k:nat
     q: Queue
     TB: Table

rew  toe(qin(d,q))=if(empty(q),d,toe(q))
     untoe(qin(d,q))= if(empty(q),emq,qin(d,untoe(q)))
     eq(prev(i,k),prev(j,k)) = or(eq(i,j),eq(k,0))
     gt(0,i) = F
```

```
gt(i,i) = F
gt(max(i,j),get_e(k,TB)) = not(geq(get_e(k,TB),max(i,j)))
```

Here the last rule was needed to detect the contradiction between `gt(max(i,j),get_e(k,TB))` and `gt(get_e(k,TB),max(i,j))`.

5.1.2 Results

After adding the equations reported in the previous section, all summands of `DKRleader2` can be proven confluent. This information can be used to generate a marked LPO. Now we can generate a state space from the original LPO with the `instantiator`, or from the marked LPO, with the algorithm by [4]. These are invoked as follows:

```
> confcheck -mark <DKRleader2.tbf >DKRleader2.mark.tbf
> instantiator -i DKRleader2.tbf
> instantiator -i -confluent ctau DKRleader2.mark.tbf
```

We did the same experiment for a network of 3, 5 and 7 nodes, respectively. The results are shown in Table 1. From this table it appears that the state space generated with an increasing number of parties grows quite rapidly. When using confluence, the state space will be constant, viz. one process claims to be the leader. Internally, the instantiator visits some more states, but only a fraction of the complete state space is visited. Note that the confluence formulae must be proved too, but this is only done once and is not influenced by the number of parties. Checking all confluence formulae takes less than one second. However, the user interaction involved by completing the data specification was considerable.

Table 1: Results for the DKR leader election protocol

	original		visited		final	
# components	# st	# tr	# st	# tr	# st	# tr
3	67	124	34	33	2	1
5	864	2687	76	75	2	1
7	18254	77055	222	221	2	1

5.2 Splice Coordination Architecture

In this section, we apply the confluence method to two applications that coordinate via the Splice architecture [5]. In particular, we take the detailed Splice-description in μCRL from [12], see also [11]. The specification provides a detailed description of how Splice-agents communicate via Ethernet.

5.2.1 Invariants

Initially, the confluence formulae appeared to be false. Inspection of the counterexamples showed that a divergence would occur when different applications would start a conversation with the same agent, or when two agents would connect to the same Ethernet-socket. These scenarios are not possible, but in order to exclude this behaviour, we need the following invariant, which was found manually and completed by inspecting the LPO and with the help of the output of `confcheck -counter` and `invcheck -counter`:

```
and(or( eq(agentAddress,address1),
        and(eq(agentAddress,noAddress),
            eq(s125,x2p1(x2p0(x2p0(one0))))))),
and(or( eq(agentAddress0,address2),
        and(eq(agentAddress0,noAddress),
            eq(s126,x2p1(x2p0(x2p0(one0))))))),
and(
and( or( eq(EthernetAddress0,noAddress), eq(EthernetAddress0,address1)),
and( or( eq(node2,noAddress), eq(node2,address1)),
and( or( eq(node3,noAddress), eq(node3,address1)),
and( or( eq(node0,noAddress), eq(node0,address1)),
     or( eq(node1,noAddress), eq(node1,address1))))))),
and( or( eq(EthernetAddress00,noAddress), eq(EthernetAddress00,address2)),
and( or( eq(node20,noAddress), eq(node20,address2)),
and( or( eq(node30,noAddress), eq(node30,address2)),
and( or( eq(node00,noAddress), eq(node00,address2)),
     or( eq(node10,noAddress), eq(node10,address2)))))))))
```

5.2.2 Confluence Checking and Marking

Given this invariant, the following dialogue is reasonable:

```
> mcrl -tbfile -regular2 -binary mysplice
> constelm mysplice.tbf | sumelm > mysplice.cs.tbf
...
Constelm: Number of removed parameter(s):7
Constelm: Number of remaining parameter(s) of output LPO:108
Constelm: Written 57 summands
...
> invcheck -invariant sp_invariant < mysplice.cs.tbf
Init OK ..................................................
Yes! invariant holds on all summands!
> invelm -invariant sp_invariant < mysplice.cs.tbf > mysplice.csi.tbf
..
summand 3 removed.
summand 5 removed.....................................
summand 48 removed.
summand 50 removed.......
> confcheck -mark <mysplice.csi.tbf >mysplice.csim.tbf
```

The output of the last command is displayed in Figure 8. First, the specification is linearized (`mcrl`) and the generated LPO is simplified (using `constelm, sumelm` [17, 28]). This removes 7 parameters. The resulting LPO has 108 parameters and 57 summands. The invariant is checked on the simplified LPO with `invcheck`. It appears to hold. Hence it is safe to use the invariant in order to eliminate the unreachable summands by `invelm`. Four summands are removed by this procedure. The resulting LPO is used as the basis for `confcheck`. Its output can be found in Figure 8. Apparently, three τ-summands are detected and marked as confluent. For this result it was essential to remove the unreachable summands with the means of the invariant of the previous section. Alternatively, instead of using `invelm` first, one can immediately apply `confcheck` as follows:

```
> confcheck -invariant sp_invariant -mark <mysplice.cs.tbf >mysplice.csim.tbf
```

This will find seven confluent summands, four of which owing to the `-invariant` option.

```
tau summand  1:................................................Confluent with all summands!
tau summand  2:.....Not confluent with summand 6
tau summand  3:................................................Confluent with all summands!
tau summand  4:....Not confluent with summand 5
tau summand  5:...Not confluent with summand 4
tau summand 22:....Not confluent with summand 5
tau summand 25:.Not confluent with summand 2
tau summand 44:.Not confluent with summand 2
tau summand 45:....Not confluent with summand 5
tau summand 46:................................................Confluent with all summands!
tau summand 47:....Not confluent with summand 5
tau summand 48:.............................................Not confluent with summand 48
tau summand 49:.............................................Not confluent with summand 48
tau summand 50:..............................................Not confluent with summand 51
tau summand 51:.............................................Not confluent with summand 50
tau summand 52:.Not confluent with summand 2
tau summand 53:.............................................Not confluent with summand 47
```

Figure 8: Output of `confcheck` on `splice.csi.tbf`

5.2.3 Effect on State Space Generation

We can now generate the full state space as a labeled transition system (LTS) in various ways. The `instantiator` [28] generates an LTS from an LPO. Using an algorithm due to Stefan Blom [4], the `instantiator` can use information on confluent τ-summands, as marked by `confcheck`. Alternatively, an LTS in SVC-file-format can be reduced by `svcconf` [19], showing the smallest state space that can be reached by repeatedly using local partial confluence at the level of LTSs. Finally, an LTS in BCG-file-format can be reduced modulo branching equivalence, using a tool from the Caesar/Aldébaran toolset [14]. A typical run goes as follows:

```
> instantiator -i mysplice.csi.tbf
> instantiator -i -confluent ctau mysplice.csim.tbf
> aut2svc mysplice.csi.aut
> svcconf mysplice.csi.svc
> svcinfo mysplice.csi.red.svc
> bcg_io  mysplice.csim.aut mysplice.csim.bcg
> bcg_min -branching -normal mysplice.csim.bcg mysplice.csim_b.bcg
> bcg_min -strong -normal mysplice.csim.bcg mysplice.csim_s.bcg
> aldebaran -info mysplice.csim_b.bcg
> aldebaran -info mysplice.csim_s.bcg
```

The sizes of the state space vary as indicated in Table 2.

Table 2: Sizes of various state graphs of SPLICE

Method	States	Transitions
instantiator (standard)	561661	1533318
instantiator -confluent (visited)	410841	1030516
instantiator -confluent (final)	304426	924101
svcconf	124990	368002
bcg_min -strong	15373	55310
bcg_min -branching	1557	6501

Roughly speaking, using local confluence, the state space can be reduced to 20% of the original half million states (i.e. a reduction of 80%). The drawback of this approach is that the larger space must be generated first. The symbolic approach reduces it only to 50%, but internally the instantiator visits 75% of the states, so it never considers 25% of the states. With branching bisimulation, a reduction to 0.25% is possible. This huge difference can have at least two origins:

- There is a lot of symmetry, in the sense that different states give rise to isomorphic subgraphs.
- Stronger forms of confluence are needed to remove the internal τ-structure

As an indication, we also reduced the state space modulo strong bisimulation, which leaves the internal structure intact, but eliminates a lot of symmetry. This shows that a reduction to 2.5% is possible by removing symmetric parts of the state space, without taking into account the structure of the internal behaviour. It is not clear, however, how to exploit this symmetry at the symbolic level.

6. Conclusions and Directions for Future Work

Although at an experimental level, the prover was successful in checking certain invariance/confluence properties of large applications. Some invariance properties were needed in order to prove the confluence properties. After confluence detection and marking, a smaller state space was generated than without using this knowledge. Experiments showed that the decision procedures of PVS [25] and Isabelle/HOL [26] couldn't deal with these large, automatically generated, formulae. We now indicate a number of possible directions for future work.

User Interaction. Using the `-print` and `-counter` options the user gets some diagnostics on the reason why the prover fails. An incomplete proof can be repaired by adding equations to the specification. This is a quite indirect interaction, compared to theorem provers like those mentioned above. It is still open whether from the BDD more interesting diagnostics can be computed, which could provide a better user interaction.

Another point to be mentioned is that in order to know the parameters of the LPO (needed for the invariants) and the sequence of the summands (needed to interpret the output of `confcheck` and `invcheck`), the user must inspect the LPO, which is in fact not meant to be human readable. Currently, the information in the LPO cannot be easily traced back to the original specification.

For the DKR-protocol user interaction was quite tedious, although straightforward. For the SPLICE-specification user-interaction consisted of providing an invariant, which was not trivial to find by hand.

Completeness of the specification. Algebraic specifications are often meant to be complete for closed terms only. However, the prover applies rewriting on open terms. At present, there is no clear criterion for the specification to be complete. The prover doesn't use induction but only applies equational reasoning. Hence for completeness, it is needed that all open theorems are derivable equationally. This would lead to the notion of ω-completeness [1]. However, many specifications don't have an ω-complete extension, or it is hard to find one. Currently, using the diagnostics of `-counter` can be helpful for incrementally adding equations to the specification.

Induction/Invariants. An alternative to equations for open terms, is adding induction principle to the prover. See [8, 10] for contributions to inductive theorem proving. This could also be helpful when proving invariants. Another obvious step would be to generate simple invariants automatically, along the lines of [3].

Another point is that it is not clear how to use derived equations/invariants. By adding the invariant to each formula to be proved, the formulae become too big to construct BDDs, and the BDDs contain redundant information. Instead of constructing the BDD of $I \wedge \Phi$, it would be more appropriate to construct the smallest BDD for Φ, given that I is known.

Improvements on Algorithm. Some ideas to improve the prover algorithm could be to incorporate the congruence closure algorithm [24] to check satisfiability of paths. Also techniques from constraint programming could be useful to deal with inequalities. Another issue is to investigate completeness for subcases (e.g. the case where the only functions are free constructors). Here the role of the orders, and the role of an ω-complete specification must be clarified. Finally, an extension with quantifiers would make the prover even more useful. The confluence formulae can be weakened with existential quantifiers. Many invariants require universal quantification.

Symmetry and Bisimulation. Ultimately, the goal of confluence analysis is to reduce the size of the generated state space. As one of the examples showed, a promising step is to exhibit bisimulation at the symbolic level. Can simple forms of symmetry/bisimulation be expressed by formulae on LPO-level?

An alternative and seemingly feasible approach is a tool which, given specification and implementation in LPO-format, and a state mapping, checks whether the mapping is a branching bisimulation (cf. focus conditions in [23]).

References

1. J. A. Bergstra and J. Heering. Which data types have omega-complete initial algebra specifications? *Theoretical Computer Science*, 124(1):149–168, 1994.
2. M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings Concur'94*, number 836 in LNCS, pages 401–416. Springer-Verlag, 1994.
3. N. Bjorner, A. Browne, and Z Manna. Automatic generation of invariants and intermediate assertions. In *Int. Conf. on Principles and Practice of Constraint Programming*, volume 976 of *LNCS*, pages 589–623. Springer-Verlag, 1995.
4. S.C.C. Blom. Partial τ-confluence for efficient state space generation. Technical report, CWI, 2001; to appear.
5. M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.
6. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software – Practice & Experience*, 30:259–291, 2000. See also `http://www.cwi.nl/projects/MetaEnv/aterm/`.
7. R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
8. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.
9. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
10. H. Comon. Inductionless induction. In René David, editor, *2nd Int. Conf. in Logic For Computer Science: Automated Deduction. Lecture notes*, Chambéry, 1994. Univ. de Savoie.
11. P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In A. Porto and G.-C. Roman, editors, *Proceedings of the Fourth International Conference on Coordination Models and Languages*, number 1906 in LNCS, pages 335–340, Limassol, Cyprus, 2000. Springer-Verlag.
12. P.F.G. Dechering and I.A. van Langevelde. Towards automated verification of Splice in μCRL. Report SEN-R0015, CWI, Amsterdam, 2000.

13. D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, September 1982.
14. J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proc. 8th Conference on Computer-Aided Verification*, number 1102 in LNCS, pages 437–440. Springer, 1996.
15. L.-Å. Fredlund, J.F. Groote, and H. Korver. Formal verification of a leader election protocol in process algebra. *Theoretical Computer Science*, 177(2):459–486, 1997.
16. J.F. Groote. The syntax and semantics of timed μCRL. Technical report, CWI, 1997.
17. J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Technical report, CWI, Amsterdam, 2001 (to be published).
18. J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Reasoning, LPAR2000*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 161–178. Springer, 2000.
19. J.F. Groote and J.C. van de Pol. State space reduction using partial τ-confluence. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science 2000*, number 1893 in LNCS, pages 383–393. Springer, 2000.
20. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *In Proc. 1st Workshop on the Algebra of Communicating Processes*, Workshops in Computing, pages 26–62. Springer, 1995.
21. J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. Technical Report SEN-R0019, CWI, Amsterdam, The Netherlands, 2000.
22. J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science B (Logic, semantics and theory of programming)*, 170(1–2):47–81, 1996.
23. J.F. Groote and J.S. Springintveld. Focus points and convergent process operators — A proof strategy for protocol verification. Report CS-R9566, CWI, Amsterdam, November 1995.
24. G. Nelson and D.C. Oppen. Fast decision procedures bases on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, 1980.
25. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Proceedings of Computer Aided Verification CAV'96*, LNCS 1102, pages 411–414. Springer-Verlag, 1996.
26. L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.
27. J.C. van de Pol. Just-in-time: on strategy annotations. Technical Report SEN-R0105, CWI, Amsterdam, March 2001.
28. A.G. Wouters. Manual for the μCRL toolset (version 2.07). Technical report, CWI, 2001; to appear. Available at `http://www.cwi.nl/~arnow`.