Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Specification of Components in a Proposition Solver

B. Lisser, J.J. van Wamel

Software Engineering (SEN)

SEN-R9720 September 30, 1997

Report SEN-R9720
ISSN 1386-369X

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

Specification of Components in a Proposition Solver

Bert Lisser and Jos van Wamel
email: {*bertl,jos*}*@cwi.nl*
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
In this paper we present a specification of components and data types for a proposition solver. The specification language we use is μCRL. The components are specified in such a way that they may serve as building blocks in various applications.

1991 Mathematics Subject Classification: 68Q60 Specification and verification of programs, 68Q65 Abstract data types; Algebraic specification.
1991 Computing Reviews Classification System: D2.1 Requirements/Specifications.
Keywords and Phrases: Abstract Data Types, Algebraic Specification, μCRL, Process Algebra, Proposition Solvers.
Note: Work carried out under project SEN2.1 "Process Specification and Analysis".

1. Introduction

The objective of this paper is to provide a library of specifications of components for generic solver tools for logical propositions. These components may serve as building blocks for solvers that can be tailored for a wide range of applications, such as proof checkers, constraint solvers and generators of transition systems. The style of specification we use is brief and functional, so that the component and data type designs are easy to carry over to any potential programmer. Various implementation details are omitted, and left to the programmer.

A proposition solver solves equations such as $p(\vec{x}) = true$, where $\vec{x}$ stands for the free boolean variables in proposition p. The satisfiability of the proposition p is checked and a solution of the equation is computed, provided that p is satisfiable.

The core system or main component of the solver tools we aim at is the satisfiability checker HeerHugo [Gro97], which was originally developed at Utrecht University and the CWI for checking railway safety systems [GKV95].

It is possible to build totally different applications from the same components. We summarise the advantages of dividing applications in components and specifying them before they are implemented.

- Development time can be saved by first building the components simultaneously and assembling them afterwards.
- In any application, components may easily be replaced by other components with the same functional specifications.
- Components may be reused in different applications.

We use the term *manager* for the process which takes care of scheduling issues and correct transport of the various data structures that flow between the components that are connected to it. A number of components, together with a manager process, define a solver tool. For the components that will be specified here, we require that they only communicate with the manager process, and not with other

components. The advantage of such a process structure is that the interfaces of the components are independent of each other.

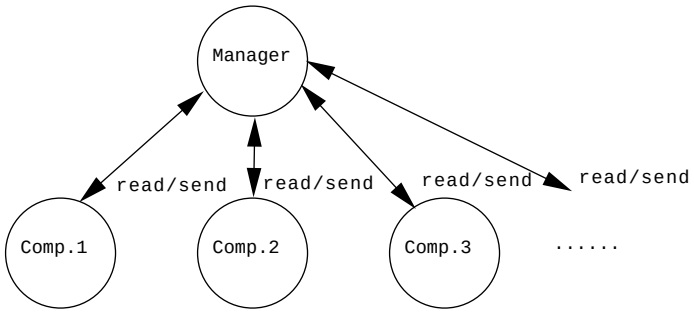

Figure 1: All communication goes via the manager

The implementation of the solver tools we have in mind should be such, that a connection with the *Toolbus* as specified in [BK95] is possible. For any implementation we assume that the Toolbus performs the manager tasks. In such an implementation the various solver components, or tools, are processes that may run concurrently. We do not go into various relevant technical details concerning the Toolbus, such as the current, a-synchronous character of the communication, as this would require the specification of quite some extra implementation details.

A similar project was carried out in [DG95], where a toolset for the manipulation of μCRL specifications was specified and implemented on the Toolbus platform.

Although many useful components may be possible in a solver for propositions, we restrict ourselves in this document to the specification of five components. No implementation details are provided, but we do give an example of a solver tool which uses several components.

The specification language which we use here is μCRL [GP94]. Data types in μCRL are algebraically specified in the standard way, using sorts, functions and axioms, see e.g. [Wam95]. Process specification in μCRL is based on the process algebra ACP (Algebra of Communicating Processes, see e.g. [BK84, BW90]), which is particularly suited for the formal description of concurrent, communicating systems.

Throughout this document we assume that the reader is familiar with proposition logic. Basic texts are for instance [NS93].

Table of Contents

2. Some general notes on proposition solving

2.1 Subtasks of a proposition solver

The easiest way for a programmer to make an application which solves equations of the form $p(\vec{x}) = true$ is to write a program which starts to loop through all 2^n instantiations of the n free variables of p, computes the boolean value of p for each instantiation, and halts with the message *satisfiable* if the proposition p becomes true. If the proposition p does not become true for any instantiation of the free variables, the program returns the message *unsatisfiable*.

This is not very efficient for large n. Here follows a more efficient implementation of this application. The application will be divided in five subtasks.

- Computation of a prenex normal form that is equisatisfiable to a proposition with quantifiers.
- Computation of a conjunctive normal form that is equisatisfiable to a quantifier-free proposition.
- Quantifier elimination in prenex normal forms. The result of this operation should be equisatisfiable to the original.
- Simplification[1] of conjunctive normal forms. The simplified conjunctive normal form should be equisatisfiable to the original one.
- Solution of the equation determined by a conjunctive normal form.

For each of these subtasks we will specify a component. The term "equisatisfiable to" is used in the descriptions of the first four subtasks. In the next subsection the relation "equisatisfiable to" will be defined.

2.2 Equisatisfiability

Let p_1 and p_2 be propositions. Let $V := \{v_1, v_2, \dots\}$ be the set of variables of the proposition logic. A *truth assignment* is a map which assigns a boolean value (*true* or *false*) to each variable v_i in V. $p_1 \sim p_2$ stands for "p_1 *equisatisfiable to* p_2". $p_1 \sim p_2$ means that the set of truth assignments that makes p_1 true is equal to the set of truth assignments that makes p_2 true.

3. System description

3.1 Description of components for proposition solvers

Components are processes that are part of a solver, which must be regarded as a larger, surrounding system. The components perform tasks for what we will call here the manager process. The manager process is also part of the surrounding system. Specific solvers, and therefore manager processes, will not be specified in this document. Our intention is only to provide a generic set-up for the specification of solvers.

In Section 2.1 we defined some natural subtasks for a solver process. The subject of this document is the specification of these components and the data types involved.

PRENEX is the component which translates propositions to prenex normal forms.

CNF is the component that transforms quantifier-free propositions to existential conjunctive normal forms.

UQE, Universal quantifier eliminator, is the component which transforms quantified conjunctive normal forms to existential conjunctive normal forms (prenex normal forms without universal quantifiers).

HH, HeerHugo, is the component that simplifies existential conjunctive normal forms.

[1] The term "simplification" remains undefined in this document except for the requirement that a simplified formula is equisatisfiable to the original one.

SC, Satisfiability Checker, is the component that returns a message about the satisfiability of propositions. If a proposition is satisfiable it returns an instantiation of the free variables which makes this proposition true.

3.2 Description of the data types belonging to the components
The most relevant data types in the specifications of the components are the following:

Prop is the type of the propositions in which the quantifiers $\forall$ and $\exists$ are permitted. From now on, the term proposition denotes an element of type `Prop`.

Let p be a proposition, then

$\forall x.p$ is interpreted as $p[x{:=}\mathit{true}]$ **and** $p[x{:=}\mathit{false}]$ and
$\exists x.p$ is interpreted as $p[x{:=}\mathit{true}]$ **or** $p[x{:=}\mathit{false}]$.

QfProp is the type of the propositions without quantifiers.

QVarList is the type of the list of quantified variables.

QProp is the type of the prenex normal forms. Each member is a pair of a list of quantified variables and a proposition without quantifiers.

Cnf is the type of the conjunctive normal forms. Each member is a set of clauses. A clause is a set of literals.

QCnf is the type of the quantified conjunctive normal forms. Each member is pair of a list of quantified variables and a conjunctive normal form.

ECnf is the type of the existential conjunctive normal forms. Each member is a pair of a set of variables and a conjunctive normal form. The variables of this conjunctive normal which are member of this set are bound by the existential quantifier. The variables of this conjunctive normal form which are not member of this set are the free variables.

VarSet consists of sets of variables.

VarSetSet consists of sets of sets of variables.

BindSet is the type in which an instantiation of free variables of a proposition will be represented. A member is a pair of two sets of variables. One set contains the variables to which the value `true` is assigned, and the other set contains the variables to which the value `false` is assigned.

3.3 Description of a proposition solver
In this section we give an example of a proposition solver as it can be composed from the components described in the previous section. The informal specification of the total system, which we call SOLVER, is as follows. The user has the choice to enter a proposition, to terminate the whole system, to restart the system or to give a time out.

After the user has entered a proposition the system starts processing it. After a while the system writes a message that informs the user about the satisfiability of the proposition and writes an instantiation which makes the proposition true, provided that the proposition is satisfiable.

The manager process in SOLVER has the following tasks.

1. To perform the user interface function.
2. To connect the various components.

3. To split a data element of sort `QProp` into a data element of sort `QfProp` (the quantifier-free body of the prenex formula), a data element of sort `QVarList` (the sequence of quantified variables), and a data element of sort `VarSet` (the set of free variables).

4. To merge a data element of sort `Ecnf` with a data element of sort `QVarList`. The result is of sort `Qcnf`.

5. To supply a read proposition with a unique identification number. This number will be connected to the data elements that belong to that proposition and it will be displayed together with the results.

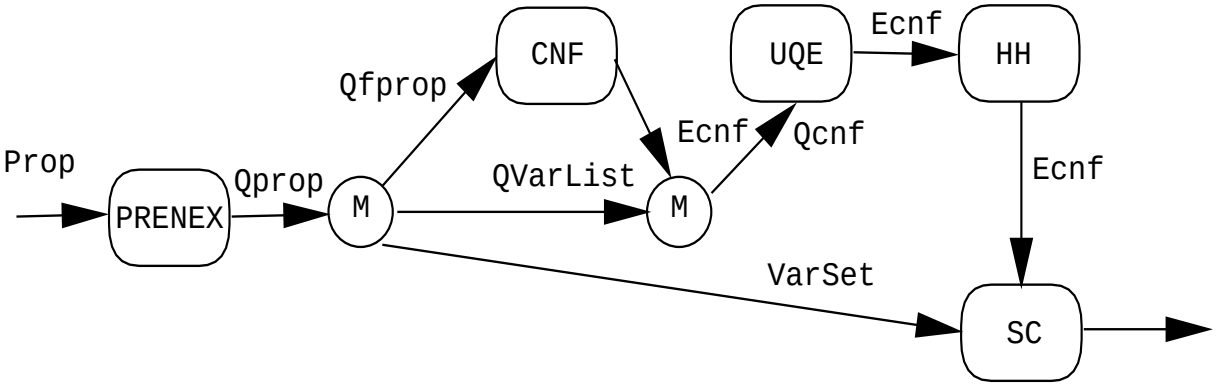

Figure 2: SOLVER – Diagram of data flow between the components

4. Process specifications

4.1 Specification of the manager interface to the components

The main activity of the manager process is to communicate with the components, and to manage the solving process. This process is not specified in this document. Below follows the specification of the interface to the components.

The manager and a component are connected via two channels: a data channel and a control channel. The data – a specific type of logical expressions – flows via the data channel. The control signals `terminate`, `timeout` and `restart` flow via the control channel.

With communication over a data channel the following actions are involved:

`rd` stands for *read data*,

`sd` stands for *send data* and

`cd` stands for *communicate data*.

With communication over a control channel the following actions are involved.

`rc` stands for *read control signal*,

`sc` stands for *send control signal* and

`cc` stands for *communicate control signal*.

The following actions are subscripted by XY, which symbolises that the data or the signals flow from X to Y.

$\texttt{rd}_{\texttt{XY}}$ means component Y reads data which was sent by component X.

$\mathtt{sd_{XY}}$ means component X sends data which will be read by component Y.

$\mathtt{cd_{XY}}$ is the communication action which is the result from the simultaneous occurrence of $\mathtt{rd_{XY}}$ and $\mathtt{sd_{XY}}$.

Analogously to this the notations for $\mathtt{rc_{XY}}, \mathtt{sc_{XY}}$ and $\mathtt{cc_{XY}}$ are defined. The symbols `p,c,u,h,s` and `m` stand for respectively the components `prenex`, `cnf`, `uqe`, `hh`, `sc` and `manager`. The notations introduced in this section will make it possible to understand the following specifications.

act $\mathtt{rd_{cm}}, \mathtt{cd_{cm}}$: ECnf
$\mathtt{sd_{mc}}, \mathtt{cd_{mc}}$: QfProp
$\mathtt{rd_{um}}, \mathtt{cd_{um}}$: ECnf
$\mathtt{sd_{mu}}, \mathtt{cd_{mu}}$: QCnf
$\mathtt{rd_{hm}}, \mathtt{cd_{hm}}$: ECnf
$\mathtt{sd_{mh}}, \mathtt{cd_{mh}}$: ECnf
$\mathtt{rd_{pm}}, \mathtt{cd_{pm}}$: QProp
$\mathtt{sd_{mp}}, \mathtt{cd_{mp}}$: Prop
$\mathtt{rd_{sm}}, \mathtt{cd_{sm}}$: BindSet
$\mathtt{sd_{ms}}, \mathtt{cd_{ms}}$: VarSet
$\mathtt{sd_{ms}}, \mathtt{cd_{ms}}$: ECnf
$\mathtt{rc_{Xm}}, \mathtt{cc_{Xm}}$: Status
$\mathtt{sc_{mX}}, \mathtt{cc_{mX}}$: Command

comm $\mathtt{sd_{Xm}} \mid \mathtt{rd_{Xm}} = \mathtt{cd_{Xm}}$
$\mathtt{sd_{mX}} \mid \mathtt{rd_{mX}} = \mathtt{cd_{mX}}$
$\mathtt{sc_{Xm}} \mid \mathtt{rc_{Xm}} = \mathtt{cc_{Xm}}$
$\mathtt{sc_{mX}} \mid \mathtt{rc_{mX}} = \mathtt{cc_{mX}}$

In the above specification X stands for `c,u,h,p` and `s`.

The set of read and write actions $\{\mathtt{rd_{Xm}}, \mathtt{rc_{Xm}}, \mathtt{sd_{mX}}, \mathtt{sc_{mX}}, \mathtt{rd_{mX}}, \mathtt{rc_{mX}}, \mathtt{sd_{Xm}}, \mathtt{sc_{Xm}}\}$ is defined as H. The set of communications $\{\mathtt{cd_{Xm}}, \mathtt{cc_{Xm}}, \mathtt{cd_{mX}}, \mathtt{cc_{mX}}\}$ is defined as I.

4.2 Specification of the components

Components are processes which may be connected to the manager process. The component specifications are provided in the following subsections.

In the specifications we use *error conditions*, symbolised by $\mathtt{c_1}, \mathtt{c_2}, \mathtt{c_3}, \mathtt{c_4}$ and $\mathtt{c_5}$, which return a boolean **T** (*true*) or **F** (*false*). These error conditions remain underspecified, but they cover all kinds of special situations that may occur in a component when it is active. We think of buffer overflows, memory problems, etc.

Whenever an error situation occurs in some component, the manager process has to be notified about the type of error that has been detected. We assume that in error situations the manager always knows which data is involved. Whenever a `restart` signal is received during a regular computation, a component immediately returns to its initial state without further actions. Termination – via a `terminate` signal – or restarting is possible in any state of the processes.

The notion of equisatisfiabity is defined in 2.2 and the function `equisatisfiable` is specified in 5.4. The expression `equisatisfiable(p1,p2) = T` states that proposition `p1` is equisatisfiable to proposition `p2`.

Let `f` $\in$ {`cnf,uqe,hh,prenex`}, and `Intypetoprop, Outtypetoprop` be conversion functions from respectively the input and output type of *f*, to the sort `Prop` of propositions. The equations

```
equisatisfiable(Intypetoprop(p),Outtypetoprop(f(p))) = T
```

which occur in several places of the process specifications, must be regarded as *requirements* on the specifications of `f` and input types of `f`, although they appear as axioms.

This way of specifying properties of data types may be somewhat unusual; instead of requiring that equisatisfiability should be a derivable property of two expressions, we state it as an axiom. This saves us, however, the often complex task of specifying the functions `f` in detail. Moreover it leaves the programmer freedom to choose between different implementations.

CNF - Transforms to existential conjunctive normal form

CNF is the component that transforms quantifier-free propositions into existential conjunctive normal forms.

func $\mathtt{cnf}: \mathtt{QfProp} \to \mathtt{ECnf}$
$\mathtt{c}_1: \mathtt{QfProp} \to \mathtt{Bool}$
var $\mathtt{qfp}: \mathtt{QfProp}$
rew $\mathtt{equisatisfiable(qfproptoprop(qfp),ecnftoprop(cnf(qfp))) = T}$
act $\mathtt{rd}_{\mathtt{mc}}: \mathtt{QfProp}$
$\mathtt{sd}_{\mathtt{cm}}: \mathtt{ECnf}$
$\mathtt{rc}_{\mathtt{mc}}: \mathtt{Command}$
$\mathtt{sc}_{\mathtt{cm}}: \mathtt{Status}$
proc $\mathtt{CNF}_0 =$
$\sum_{\mathtt{qfp:QfProp}} \mathtt{rd}_{\mathtt{mc}}(\mathtt{qfp}).(\mathtt{CNF}_1(\mathtt{qfp}) \lhd \mathtt{c}_1(\mathtt{qfp}) \rhd \mathtt{CNF}_2) +$
$\mathtt{rc}_{\mathtt{mc}}(\mathtt{restart}).\mathtt{CNF}_0 + \mathtt{rc}_{\mathtt{mc}}(\mathtt{terminate})$
$\mathtt{CNF}_1(\mathtt{qfp} : \mathtt{QfProp}) =$
$\mathtt{sd}_{\mathtt{cm}}(\mathtt{cnf}(\mathtt{qfp})).\mathtt{CNF}_0 + \mathtt{rc}_{\mathtt{mc}}(\mathtt{restart}).\mathtt{CNF}_0 + \mathtt{rc}_{\mathtt{mc}}(\mathtt{terminate})$
$\mathtt{CNF}_2 =$
$\sum_{\mathtt{n:Nat}} \mathtt{sc}_{\mathtt{cm}}(\mathtt{notokmsg}(\mathtt{n})).\mathtt{CNF}_0 + \mathtt{rc}_{\mathtt{mc}}(\mathtt{restart}).\mathtt{CNF}_2 + \mathtt{rc}_{\mathtt{mc}}(\mathtt{terminate})$

UQE - Universal quantifier eliminator

UQE, Universal Quantifier Eliminator, is the component which transforms quantified conjunctive normal forms into existential conjunctive normal forms.

func $\mathtt{uqe}: \mathtt{QCnf} \to \mathtt{ECnf}$
$\mathtt{c}_2: \mathtt{QCnf} \to \mathtt{Bool}$
var $\mathtt{qcn}: \mathtt{QCnf}$
rew $\mathtt{equisatisfiable(qcnftoprop(qcn),ecnftoprop(uqe(qcn))) = T}$
act $\mathtt{rd}_{\mathtt{mu}}: \mathtt{QCnf}$
$\mathtt{sd}_{\mathtt{um}}: \mathtt{ECnf}$
$\mathtt{rc}_{\mathtt{mu}}: \mathtt{Command}$
$\mathtt{sc}_{\mathtt{um}}: \mathtt{Status}$
proc $\mathtt{UQE}_0 =$
$\sum_{\mathtt{qcn:QCnf}} \mathtt{rd}_{\mathtt{mu}}(\mathtt{qcn}).(\mathtt{UQE}_1(\mathtt{qcn}) \lhd \mathtt{c}_2(\mathtt{qcn}) \rhd \mathtt{UQE}_2) +$
$\mathtt{rc}_{\mathtt{mu}}(\mathtt{restart}).\mathtt{UQE}_0 + \mathtt{rc}_{\mathtt{mu}}(\mathtt{terminate})$
$\mathtt{UQE}_1(\mathtt{qcn} : \mathtt{QCnf}) =$
$\mathtt{sd}_{\mathtt{um}}(\mathtt{uqe}(\mathtt{qcn})).\mathtt{UQE}_0 + \mathtt{rc}_{\mathtt{mu}}(\mathtt{restart}).\mathtt{UQE}_0 + \mathtt{rc}_{\mathtt{mu}}(\mathtt{terminate})$
$\mathtt{UQE}_2 =$
$\sum_{\mathtt{n:Nat}} \mathtt{sc}_{\mathtt{um}}(\mathtt{notokmsg}(\mathtt{n})).\mathtt{UQE}_0 + \mathtt{rc}_{\mathtt{mu}}(\mathtt{restart}).\mathtt{UQE}_2 + \mathtt{rc}_{\mathtt{mu}}(\mathtt{terminate})$

HH - Heerhugo

HH, HeerHugo, is the component that simplifies existential conjunctive normal forms. After it has read an existential conjunctive normal form without detecting errors, a simplified existential conjunctive normal form is computed. If a time out signal is received during the computation, then an equisatisfiable existential normal form will be sent as soon as possible. If no time out signal is received, then after the computation the fully simplified, equisatisfiable existential normal form will be sent to the manager process.

func $\mathtt{hh}: \mathtt{ECnf} \to \mathtt{ECnf}$
$\mathtt{c}_3: \mathtt{ECnf} \to \mathtt{Bool}$
$\mathtt{equisatisfiable}: \mathtt{ECnf} \times \mathtt{ECnf} \to \mathtt{Bool}$
var $\mathtt{ecn}, \mathtt{ecn1}, \mathtt{ecn2}: \mathtt{ECnf}$
rew $\mathtt{equisatisfiable}(\mathtt{ecnftoprop}(\mathtt{ecn}),\mathtt{ecnftoprop}(\mathtt{hh}(\mathtt{ecn}))) = \mathbf{T}$
$\mathtt{equisatisfiable}(\mathtt{ecn1},\mathtt{ecn2}) = \mathtt{equisatisfiable}(\mathtt{ecnftoprop}(\mathtt{ecn1}),\mathtt{ecnftoprop}(\mathtt{ecn2}))$
act $\mathtt{rd}_{\mathtt{mh}}: \mathtt{ECnf}$
$\mathtt{sd}_{\mathtt{hm}}: \mathtt{ECnf}$
$\mathtt{rc}_{\mathtt{mh}}: \mathtt{Command}$
$\mathtt{sc}_{\mathtt{hm}}: \mathtt{Status}$
proc $\mathtt{HH}_0 =$
$\sum_{\mathtt{ecn:ECnf}} \mathtt{rd}_{\mathtt{mh}}(\mathtt{ecn}).(\mathtt{HH}_1(\mathtt{ecn}) \lhd \mathtt{c}_3(\mathtt{ecn}) \rhd \mathtt{HH}_2) +$
$\mathtt{rc}_{\mathtt{mh}}(\mathtt{restart}).\mathtt{HH}_0 + \mathtt{rc}_{\mathtt{mh}}(\mathtt{timeout}).\mathtt{HH}_0 + \mathtt{rc}_{\mathtt{mh}}(\mathtt{terminate})$
$\mathtt{HH}_1(\mathtt{ecn} : \mathtt{ECnf}) =$
$\mathtt{sd}_{\mathtt{hm}}(\mathtt{hh}(\mathtt{ecn})).\mathtt{HH}_0 + \mathtt{rc}_{\mathtt{mh}}(\mathtt{timeout}).\mathtt{TIMEOUT}(\mathtt{ecn}) +$
$\mathtt{rc}_{\mathtt{mh}}(\mathtt{restart}).\mathtt{HH}_0 + \mathtt{rc}_{\mathtt{mh}}(\mathtt{terminate})$
$\mathtt{HH}_2 =$
$\sum_{\mathtt{n:Nat}} \mathtt{sc}_{\mathtt{hm}}(\mathtt{notokmsg}(\mathtt{n})).\mathtt{HH}_0 + \mathtt{rc}_{\mathtt{mh}}(\mathtt{restart}).\mathtt{HH}_2 +$
$\mathtt{rc}_{\mathtt{mh}}(\mathtt{timeout}).\mathtt{HH}_2 + \mathtt{rc}_{\mathtt{mh}}(\mathtt{terminate})$
$\mathtt{TIMEOUT}(\mathtt{ecn1} : \mathtt{ECnf}) =$
$\sum_{\mathtt{ecn2:ECnf}} (\mathtt{sd}_{\mathtt{hm}}(\mathtt{ecn2}).\mathtt{HH}_0 \lhd \mathtt{equisatisfiable}(\mathtt{ecn1}, \mathtt{ecn2}) \rhd \delta)$
$+\mathtt{rc}_{\mathtt{mh}}(\mathtt{restart}).\mathtt{HH}_0 + \mathtt{rc}_{\mathtt{mh}}(\mathtt{timeout}).\mathtt{TIMEOUT}(\mathtt{ecn1}) + \mathtt{rc}_{\mathtt{mh}}(\mathtt{terminate})$

PRENEX - Translates propositions to prenex normal forms

PRENEX is the component which translates propositions into prenex normal forms.

func $\mathtt{prenex}: \mathtt{Prop} \to \mathtt{QProp}$
$\mathtt{c}_4: \mathtt{Prop} \to \mathtt{Bool}$
var $\mathtt{p}: \mathtt{Prop}$
rew $\mathtt{equisatisfiable}(\mathtt{p},\mathtt{qproptoprop}(\mathtt{prenex}(\mathtt{p}))) = \mathbf{T}$
act $\mathtt{rd}_{\mathtt{mp}}: \mathtt{Prop}$
$\mathtt{sd}_{\mathtt{pm}}: \mathtt{QProp}$
$\mathtt{rc}_{\mathtt{mp}}: \mathtt{Command}$
$\mathtt{sc}_{\mathtt{pm}}: \mathtt{Status}$
proc $\mathtt{PRENEX}_0 =$
$\sum_{\mathtt{p:Prop}} \mathtt{rd}_{\mathtt{mp}}(\mathtt{p}).(\mathtt{PRENEX}_1(\mathtt{p}) \lhd \mathtt{c}_4(\mathtt{p}) \rhd \mathtt{PRENEX}_2) +$
$\mathtt{rc}_{\mathtt{mp}}(\mathtt{restart}).\mathtt{PRENEX}_0 + \mathtt{rc}_{\mathtt{mp}}(\mathtt{terminate})$
$\mathtt{PRENEX}_1(\mathtt{p} : \mathtt{Prop}) =$

$$\mathtt{sd_{pm}(prenex(p)).PRENEX_0 + rc_{mp}(restart).PRENEX_0 + rc_{mp}(terminate)}$$

$$\mathtt{PRENEX_2} = \sum_{\mathtt{n:Nat}} \mathtt{sc_{pm}(notokmsg(n)).PRENEX_0 + rc_{mp}(restart).PRENEX_2 + rc_{mp}(terminate)}$$

SC - Satisfiability Checker

SC, Satisfiability Checker, is the component that returns for each read proposition p a message about the satisfiability of this proposition. If the proposition is satisfiable it also returns a truth assignment which makes the proposition *true*.

A truth assignment is a map from proposition variables to the booleans *false* and *true*. The user is usually interested in truth assignments restricted to a domain which contains all the free variables of p. This domain is often not equal to the set of free variables of p itself, because the proposition offered to the Satisfiability Checker may be a simplified form of the original one. (Simplification may lead to the elimination of free variables.)

For example, suppose that the user wants to analyze $x \vee \neg x$, which is of course satisfiable. The simplified form is of this expression is *true*. The user may expect a truth assignment like $x = \mathit{false}$. But the Satisfiability Checker has no information that the user is interested in the value of variable x. Therefore, besides that the satisfiability has to be checked for any read proposition p, also the set of variables vs in which the user is interested has to be offered to the Satisfiability Checker.

func
$\mathtt{freevars: ECnf \to VarSet}$
$\mathtt{eval: ECnf \times VarSet \to Bool}$
$\mathtt{satvars: ECnf \to VarSetSet}$
$\mathtt{c_5: ECnf \times VarSet \to Bool}$

var
$\mathtt{vs: VarSet}$
$\mathtt{ecn: ECnf}$

rew
$\mathtt{freevars(ecn) = freevars(ecnftoprop(ecn))}$
$\mathtt{eval(ecn,vs) = eval(ecnftoprop(ecn),vs)}$
$\mathtt{satvars(ecn) = satvars(ecnftoprop(ecn),freevars(ecn))}$

act
$\mathtt{rd_{ms}: ECnf}$
$\mathtt{rd_{ms}: VarSet}$
$\mathtt{sd_{sm}: BindSet}$
$\mathtt{rc_{ms}: Command}$
$\mathtt{sc_{sm}: Status}$
$\mathtt{sc_{sm}: ResultSC}$

proc

$$\mathtt{SC_0} = \mathtt{Inp_{11}} + \mathtt{Inp_{21}} + \mathtt{rc_{ms}(restart).SC_0} + \mathtt{rc_{ms}(terminate)}$$

$$\mathtt{Inp_{11}} = \sum_{\mathtt{ecn:ECnf}} \mathtt{rd_{ms}(ecn).Inp_{12}(ecn)}$$

$$\mathtt{Inp_{12}(ecn : ECnf)} = \sum_{\mathtt{vs:VarSet}} \mathtt{rd_{ms}(vs).}(\mathtt{SC_1(ecn, vs)} \lhd \mathtt{c_5(ecn, vs)} \rhd \mathtt{SC_3}) + \mathtt{rc_{ms}(restart).SC_0} + \mathtt{rc_{ms}(terminate)}$$

$$\mathtt{Inp_{21}} = \sum_{\mathtt{vs:VarSet}} \mathtt{rd_{ms}(vs).Inp_{22}(vs)}$$

$$\mathtt{Inp_{22}(vs : VarSet)} = \sum_{\mathtt{ecn:ECnf}} \mathtt{rd_{ms}(ecn).}(\mathtt{SC_1(ecn, vs)} \lhd \mathtt{c_5(ecn, vs)} \rhd \mathtt{SC_3}) + \mathtt{rc_{ms}(restart).SC_0} + \mathtt{rc_{ms}(terminate)}$$

$$\begin{aligned}
&\mathtt{SC}_1(\mathtt{ecn}:\mathtt{ECnf},\mathtt{vs}:\mathtt{VarSet}) = \\
&\quad \mathtt{sc}_{\mathtt{sm}}(\mathtt{unsatisfiable}).\mathtt{SC}_0 \lhd \mathtt{isempty}(\mathtt{satvars}(\mathtt{ecn})) \rhd \\
&\quad \mathtt{sc}_{\mathtt{sm}}(\mathtt{satisfiable}).\mathtt{SC}_2(\mathtt{ecn},\mathtt{vs}) \\
&\quad + \mathtt{rc}_{\mathtt{ms}}(\mathtt{restart}).\mathtt{SC}_0 + \mathtt{rc}_{\mathtt{ms}}(\mathtt{terminate}) \\
&\mathtt{SC}_2(\mathtt{ecn}:\mathtt{ECnf},\mathtt{vs1}:\mathtt{VarSet}) = \\
&\quad \sum_{\mathtt{vs2}:\mathtt{VarSet}} (\mathtt{sd}_{\mathtt{sm}}(\mathtt{bindset}(\mathtt{vs2},\mathtt{vs1})) \lhd \mathtt{test}(\mathtt{vs2},\mathtt{satvars}(\mathtt{ecn})) \rhd \delta).\mathtt{SC}_0 \\
&\quad + \mathtt{rc}_{\mathtt{ms}}(\mathtt{restart}).\mathtt{SC}_0 + \mathtt{rc}_{\mathtt{ms}}(\mathtt{terminate}) \\
&\mathtt{SC}_3 = \\
&\quad \sum_{\mathtt{n}:\mathtt{Nat}} \mathtt{sc}_{\mathtt{sm}}(\mathtt{notokmsg}(\mathtt{n})).\mathtt{SC}_0 + \mathtt{rc}_{\mathtt{ms}}(\mathtt{restart}).\mathtt{SC}_3 + \mathtt{rc}_{\mathtt{ms}}(\mathtt{terminate})
\end{aligned}$$

4.3 Specification of a solver

The total system SOLVER can now be specified in μCRL as

$$\tau_{\mathtt{I}} \circ \partial_{\mathtt{H}} \left(\mathtt{M} \parallel \mathtt{PRENEX}_0 \parallel \mathtt{CNF}_0 \parallel \mathtt{UQE}_0 \parallel \mathtt{HH}_0 \parallel \mathtt{SC}_0\right)$$

We assume that M specifies a *manager* process, which we do not specify explicitly.

5. Data specifications

5.1 Basic types

μCRL definition of the sort Bool

sort Bool
func **T**,**F**: → Bool
not: Bool → Bool
implies: Bool × Bool → Bool
or: Bool × Bool → Bool
and: Bool × Bool → Bool
eq: Bool × Bool → Bool
var b: Bool
rew **not T** = **F**
not F = **T**
T implies b = b
F implies b = **T**
T and b = b
F and b = **F**
T or b = **T**
F or b = b
eq(**T**,b) = b
eq(**F**,b) = **not** b

For readability the boolean operators **or**, **and** and **implies** will be written in bold as infix operators.

μCRL definition of the sort Nat

sort Nat
func 0: → Nat
S: Nat → Nat
eq: Nat × Nat → Bool
var n,n1,n2: Nat
rew eq(0,0) = **T**

```
eq(0,S(n)) = F
eq(S(n),0) = F
eq(S(n1),S(n2)) = eq(n1,n2)
```

5.2 Variables
Definition of the sort `Var`
A member of the sort `Var` is a variable. With each variable a unique natural number is associated.

μCRL definition of the sort `Var`

```
sort    Var
func    vr:  Nat → Var
        varno:  Var → Nat
        eq:  Var × Var → Bool
var     n:  Nat
        v1,v2:  Var
rew     varno(vr(n)) = n
        eq(v1,v2) = eq(varno(v1),varno(v2))
```

Definition of the sort `QF`
The sort `QF` is the set $\{\forall, \exists\}$.

μCRL definition of the sort `QF`

```
sort    QF
func    exs:  → QF
        all:  → QF
        eq:  QF × QF → Bool
rew     eq(exs,exs) = T
        eq(all,all) = T
        eq(exs,all) = F
        eq(all,exs) = F
```

Definition of the sort `QVar`
A member of the sort `QVar` is a quantified variable, which is a pair that consists of a quantifier and a variable.

μCRL definition of the sort `QVar`

```
sort    QVar
func    qvar:  Var × QF→ QVar
        vr:  QVar → Var
        qf:  QVar → QF
var     v:  Var
        q:  QF
rew     vr(qvar(v,q)) = v
        qf(qvar(v,q)) = q
```

Definition of the sort `QVarList`
A sequence of quantified variables is represented by a list structure. We do not use sets, because the order of the quantified variables is important.

μCRL definition of the sort `QVarList`

```
sort      QVarList
func      ∅_qvl:  → QVarList
          add:  QVar × QVarList → QVarList
```

5.3 Sets
Definition of the sort `VarSet`
The sort `VarSet` contains sets of variables of sort `Var`.

μCRL definition of the sort `VarSet`

```
sort      VarSet
func      ∅_vs:  → VarSet
          add:  Var × VarSet → VarSet
          rem:  Var × VarSet → VarSet
          test:  Var × VarSet → Bool
          diff:  VarSet × VarSet → VarSet
          eq:  VarSet × VarSet → Bool
          union:  VarSet × VarSet → VarSet
          if:  Bool × VarSet × VarSet → VarSet
var       v,v1,v2:  Var
          vs,vs1,vs2:  VarSet
rew       add(v,add(v,vs)) = add(v,vs)
          add(v1,add(v2,vs)) = add(v2,add(v1,vs))
          rem(v,∅_vs) = ∅_vs
          rem(v1,add(v2,vs)) = if(eq(v1,v2),rem(v1,vs),add(v2,rem(v1,vs)))
          test(v1,∅_vs) = F
          test(v1,add(v2,vs)) = eq(v1,v2) or test(v1,vs)
          eq(∅_vs,∅_vs) = T
          eq(∅_vs,add(v,vs)) = F
          eq(add(v,vs1),vs2) = test(v,vs2) and eq(rem(v,vs1),rem(v,vs2))
          union(∅_vs,vs) = vs
          union(add(v,vs1),vs2) = add(v,union(vs1,vs2))
          diff(vs,∅_vs) = vs
          diff(vs1,add(v,vs2)) = rem(v,diff(vs1,vs2))
          if (T,vs1,vs2) = vs1
          if (F,vs1,vs2) = vs2
```

Definition of the sort `VarSetSet`
A member of the sort `VarSetSet` is a set whose members are sets of variables. The function *power* : *VarSet* ↦ *VarSetSet* calculates the powerset of a set of variables.

It uses the auxiliary function *join* : *Var* × *VarSetSet* ↦ *VarSetSet*, which is defined by:

$$join(v, vss) := \{\{v\} \cup vs | vs \in vss\}$$

where *vss* is of sort *VarSetSet* and *v* is of sort *Var*.

The function *power* is defined by:

$$power(\emptyset) := \{\emptyset\}$$

$$power(vs) := power(vs \setminus \{v\}) \cup join(v, power(vs \setminus \{v\}))$$

where vs is of sort *VarSet* and $v \in vs$ of sort *Var*.

μCRL definition of the sort `VarSetSet`

```
sort    VarSetSet
func    ∅vss:  → VarSetSet
        add:  VarSet × VarSetSet → VarSetSet
        rem:  VarSet × VarSetSet → VarSetSet
        test:  VarSet × VarSetSet → Bool
        eq:  VarSetSet × VarSetSet → Bool
        isempty:  VarSetSet → Bool
        union:  VarSetSet × VarSetSet → VarSetSet
        join:  Var × VarSetSet → VarSetSet
        power:  VarSet → VarSetSet
        if:  Bool × VarSetSet × VarSetSet → VarSetSet
var     vs,vs1,vs2:  VarSet
        v:  Var
        vss,vss1,vss2:  VarSetSet
rew     add(vs,add(vs,vss)) = add(vs,vss)
        add(vs1,add(vs2,vss)) = add(vs2,add(vs1,vss))
        rem(vs,∅vss) = ∅vss
        rem(vs1,add(vs2,vss)) = if(eq(vs1,vs2),rem(vs1,vss),add(vs2,rem(vs1,vss)))
        test(vs,∅vss) = F
        test(vs1,add(vs2,vss)) = eq(vs1,vs2) or test(vs1,vss)
        eq(∅vss,∅vss) = T
        eq(∅vss,add(vs,vss)) = F
        eq(add(vs,vss1),vss2) = test(vs,vss2) and eq(rem(vs,vss1),rem(vs,vss2))
        isempty(vss) = eq(vss,∅vss)
        union(∅vss,vss) = vss
        union(add(vs,vss1),vss2) = add(vs,union(vss1,vss2))
        join(v,∅vss) = ∅vss
        join(v,add(vs,vss)) = add(add(v,vs),join(v,rem(vs,vss)))
        power(∅vs) = add(∅vs,∅vss)
        power(add(v,vs)) = union(power(rem(v,vs)),join(v,power(rem(v,vs))))
        if (T,vss1,vss2) = vss1
        if (F,vss1,vss2) = vss2
```

5.4 Propositions

Definition of the sort `Pred`

The sort `Pred` is defined as the set of the 0-ary predicates `true` and `false`.

μCRL definition of the sort `Pred`

```
sort    Pred
func    true:  → Pred
        false:  → Pred
```

Definition of the sort `Atom`
An atom is a propositional variable or a predicate.

μCRL definition of the sort `Atom`

```
sort    Atom
func    pcons:  Pred → Atom
        pvar:  Nat → Atom
        pvarno:  Atom → Nat
        eq:  Atom × Atom → Bool
var     n:  Nat
        a1,a2:  Atom
rew     pvarno(pcons(false)) = 0
        pvarno(pcons(true)) = S(0)
        pvarno(pvar(n)) = S(S(n))
        eq(a1,a2) = eq(pvarno(a1),pvarno(a2))
```

Definition of the sort `Prop`
The sort `Prop` is the set of propositions, which also includes the propositions with quantifiers.

The function $eval : Prop \times VarSet \mapsto Bool$, which has a proposition p and a set of variables vs as arguments, calculates the boolean value of p. For all free occurrences of variables $v \in vs$ of p, $eval(p, vs)$ substitutes *true*. For free variables $v \notin vs$ of p it substitutes *false*.

The function call $equisatisfiable(p_1, p_2)$ tests the following informally formulated statement:

> The solutions of the equation $p_1(\vec{x}) = true$ are the same as the solutions of the equation $p_2(\vec{x}) = true$, where $\vec{x}$ are the joint free variables of p_1 and p_2.

The function call $satvars(p, vs)$ has as arguments a proposition p and a set of variables vs. This function computes the set vss of truth assignments which make the proposition p *true*. For all $vs \in vss$ the set vs contains the free variables of p to which the value *true* is assigned, in order to make p *true*.

μCRL definition of the sort `Prop`

```
sort    Prop
func    prop:  Atom → Prop
        not:  Prop → Prop
        and:  Prop × Prop → Prop
        or:  Prop × Prop → Prop
        implies:  Prop × Prop → Prop
        all:  Var × Prop → Prop
        exs:  Var × Prop → Prop
        sub:  Prop × Var × Atom → Prop
        freevars:  Prop → VarSet
        eval:  Prop × VarSet → Bool
        satvars1:  Prop × VarSetSet × VarSetSet → VarSetSet
        satvars:  Prop × VarSet → VarSetSet
        equisatisfiable:  Prop × Prop → Bool
        if:  Bool × Prop × Prop -> Prop
var     p,p1,p2:  Prop
        a:  Atom
        vss,vss1,vss2:  VarSetSet
```

```
          vs:  VarSet
          v,v1,v2:  Var
          n:  Nat
          pr:  Pred
rew       freevars(prop(pcons(pr))) = ∅vs
          freevars(prop(pvar(n))) = add(vr(n),∅vs)
          freevars(not(p)) = freevars(p)
          freevars(and(p1,p2)) = union(freevars(p1),freevars(p2))
          freevars(or(p1,p2)) = union(freevars(p1),freevars(p2))
          freevars(implies(p1,p2)) = union(freevars(p1),freevars(p2))
          freevars(all(v,p)) = rem(v,freevars(p))
          freevars(exs(v,p)) = rem(v,freevars(p))

          sub(prop(pcons(pr)),v,a) = prop(pcons(pr))
          sub(prop(pvar(n)),v,a) = if (eq(n,varno(v)),prop(a),prop(pvar(n)))
          sub(not(p),v,a) = not(sub(p,v,a))
          sub(and(p1,p2),v,a) = and(sub(p1,v,a),sub(p2,v,a))
          sub(or(p1,p2),v,a) = or(sub(p1,v,a),sub(p2,v,a))
          sub(implies(p1,p2),v,a) = implies(sub(p1,v,a),sub(p2,v,a))
          sub(all(v1,p),v2,a) = if (eq(v1,v2),all(v1,p),all(v1,sub(p,v2,a)))
          sub(exs(v1,p),v2,a) = if (eq(v1,v2),exs(v1,p),exs(v1,sub(p,v2,a)))

          eval(prop(pcons(true)),vs) = T
          eval(prop(pcons(false)),vs) = F
          eval(prop(pvar(n)),vs) = test(vr(n),vs)
          eval(not(p),vs) = not eval(p,vs)
          eval(and(p1,p2),vs) = eval(p1,vs) and eval(p2,vs)
          eval(or(p1,p2),vs) = eval(p1,vs) or eval(p2,vs)
          eval(implies(p1,p2),vs) = eval(p1,vs) implies eval(p2,vs)
          eval(all(v,p),vs) = eval(sub(p,v,pcons(true)),vs) and
               eval(sub(p,v,pcons(false)),vs)
          eval(exs(v,p),vs) = eval(sub(p,v,pcons(true)),vs) or
               eval(sub(p,v,pcons(false)),vs)

          satvars1(p,∅vss,vss) = vss
          satvars1(p,add(vs,vss1),vss2) = if (eval(p,vs),
               satvars1(p,rem(vs,vss1),add(vs,vss2)),satvars1(p,rem(vs,vss1),vss2))

          satvars(p,vs) = satvars1(p,power(freevars(p)),∅vss)

          equisatisfiable(p1,p2) = eq(satvars(p1,union(freevars(p1),freevars(p2))),
               satvars(p2,union(freevars(p1),freevars(p2))))

          if (T,p1,p2) = p1
          if (F,p1,p2) = p2
```

5.5 Prenex normal forms

Definition of the sort `QfProp`

The sort `QfProp` is the set of propositions without quantifiers.

μCRL definition of the sort `QfProp`

```
sort      QfProp
func      qfprop:  Atom → QfProp
          not:  QfProp → QfProp
          and:  QfProp × QfProp → QfProp
          or:  QfProp × QfProp → QfProp
          implies:  QfProp × QfProp → QfProp
          qfproptoprop:  QfProp → Prop
var       qfp,qfp1,qfp2:  QfProp
          a:  Atom
rew       qfproptoprop(qfprop(a)) = prop(a)
          qfproptoprop(not(qfp)) = not(qfproptoprop(qfp))
          qfproptoprop(and(qfp1,qfp2)) = and(qfproptoprop(qfp1),qfproptoprop(qfp2))
          qfproptoprop(or(qfp1,qfp2)) = or(qfproptoprop(qfp1),qfproptoprop(qfp2))
          qfproptoprop(implies(qfp1,qfp2)) = implies(qfproptoprop(qfp1),
               qfproptoprop(qfp2))
```

Definition of the sort `Qprop`
A member of the sort `QProp` is a pair that consists of a list of quantified variables and a quantifier-free proposition. Such a pair will be interpreted as the quantifier-free proposition prefixed with the list. The list of quantified variables will be interpreted from inside to outside. So, for instance, the last added quantified variable in the list will be interpreted as the outermost quantifier-variable pair of the proposition.

μCRL definition of the sort `Qprop`

```
sort      QProp
func      qprop:  QVarList × QfProp → QProp
          qvarlist:  QProp → QVarList
          qfprop:  QProp → QfProp
          qproptoprop:  QProp → Prop
var       qvl:  QVarList
          qfp:  QfProp
          qv:  QVar
rew       qvarlist(qprop(qvl,qfp)) = qvl
          qfprop(qprop(qvl,qfp)) = qfp
          qproptoprop(qprop(∅_qvl,qfp)) = qfproptoprop(qfp)
          qproptoprop(qprop(add(qv,qvl),qfp)) = if(eq(qf(qv),all),
               all(vr(qv),qproptoprop(qprop(qvl,qfp))),
               exs(vr(qv),qproptoprop(qprop(qvl,qfp))))
```

5.6 Conjunctive normal forms
Definition of a literal
A member of the sort `Literal` is an atom or the negation of it. Literals are members of a clause.

μCRL definition of a literal

```
sort      Literal
```

```
func    lit:  Atom → Literal
        not:  Literal → Literal
        atom:  Literal → Atom
        sign:  Literal → Bool
        eq:  Literal × Literal → Bool
var     l1,l2,l:  Literal
        a:  Atom
rew     atom(lit(a)) = a
        atom(not(lit(a))) = a
        not(not(l)) = l
        sign(lit(a)) = T
        sign(not(lit(a))) = F
        eq(l1,l2) = eq(atom(l1),atom(l2)) and eq(sign(l1),sign(l2))
```

Definition of a clause
A member of the sort `Clause` is a set of literals. This will be interpreted as a disjunction of literals. So an empty clause will have to be interpreted as **F**.

μCRL definition of a clause

```
sort    Clause
func    ∅cl:  → Clause
        add:  Literal × Clause → Clause
        rem:  Literal × Clause → Clause
        test:  Literal × Clause → Bool
        eq:  Clause × Clause → Bool
        if:  Bool × Clause × Clause → Clause
var     l,l1,l2:  Literal
        cl,cl1,cl2:  Clause
rew     add(l,add(l,cl)) = add(l,cl)
        add(l1,add(l2,cl)) = add(l2,add(l1,cl))
        rem(l,∅cl) = ∅cl
        rem(l1,add(l2,cl)) = if(eq(l1,l2),rem(l1,cl),add(l2,rem(l1,cl)))
        test(l1,∅cl) = F
        test(l1,add(l2,cl)) = eq(l1,l2) or test(l1,cl)
        eq(∅cl,∅cl) = T
        eq(∅cl,add(l1,cl)) = F
        eq(add(l,cl1),cl2) = test(l,cl2) and eq(rem(l,cl1),rem(l,cl2))
        if (T,cl1,cl2) = cl1
        if (F,cl1,cl2) = cl2
```

Definition of a conjunctive normal form
A member of the sort `cnf` is a set of clauses. This will be interpreted as a conjunction of clauses. So an empty conjunctive normal form will have to be interpreted as **T**.

μCRL definition of a conjunctive normal form

```
sort    Cnf
func    ∅cnf:  → Cnf
```

```
          add:  Clause × Cnf → Cnf
          rem:  Clause × Cnf → Cnf
          if:  Bool × Cnf × Cnf → Cnf
var       cl,cl1,cl2:  Clause
          cn,cn1,cn2:  Cnf
rew       add(cl,add(cl,cn)) = add(cl,cn)
          add(cl1,add(cl2,cn)) = add(cl2,add(cl1,cn))
          rem(cl,∅cnf) = ∅cnf
          rem(cl1,add(cl2,cn)) = if(eq(cl1,cl2),rem(cl1,cn),add(cl2,rem(cl1,cn)))
          if (T,cn1,cn2) = cn1
          if (F,cn1,cn2) = cn2
```

Definition of a quantified conjunctive normal form
A member of the sort `QCnf` consists of a conjunctive normal form and a list of quantified variables. This will be interpreted as a prenex conjunctive normal form.

μCRL definition of a quantified conjunctive normal form

```
sort      QCnf
func      qcnf:  QVarList × Cnf → QCnf
          cnf:  QCnf → Cnf
          qvarlist:  QCnf → QVarList
var       cn:  Cnf
          qvl:  QVarList
rew       cnf(qcnf(qvl,cn)) = cn
          qvarlist(qcnf(qvl,cn)) = qvl
```

Definition of an existential conjunctive normal form
A member of the sort `ECnf` consists of a conjunctive normal form and a set of variables. This will be interpreted as a prenex conjunctive normal form without universal quantifiers.

μCRL definition of an existential conjunctive normal form

```
sort      ECnf
func      ecnf:  VarSet × Cnf → ECnf
          cnf:  ECnf → Cnf
          varset:  ECnf → VarSet
var       cn:  Cnf
          vs:  VarSet
rew       cnf(ecnf(vs,cn)) = cn
          varset(ecnf(vs,cn)) = vs
```

Functions which convert a cnf to a proposition

```
func      littoprop:  Literal → Prop
          clausetoprop:  Clause → Prop
          cnftoprop:  Cnf → Prop
          ecnftoprop:  ECnf → Prop
          qcnftoprop:  QCnf → Prop
```

```
var     v:  Var
        vs:  VarSet
        qvl:  QVarList
        qv:  QVar
        l:  Literal
        cl:  Clause
        cn:  Cnf
rew     littoprop(l) = if (sign(l),prop(atom(l)),not(prop(atom(l))))
        clausetoprop(∅_cl) = prop(pcons(false))
        clausetoprop(add(l,cl)) = or(littoprop(l),clausetoprop(rem(l,cl)))
        cnftoprop(∅_cnf) = prop(pcons(true))
        cnftoprop(add(cl,cn)) = and(clausetoprop(cl),cnftoprop(rem(cl,cn)))
        ecnftoprop(ecnf(∅_vs,cn)) = cnftoprop(cn)
        ecnftoprop(ecnf(add(v,vs),cn)) = exs(v,ecnftoprop(ecnf(rem(v,vs),cn)))
        qcnftoprop(qcnf(∅_qvl,cn)) = cnftoprop(cn)
        qcnftoprop(qcnf(add(qv,qvl),cn)) = if(eq(qf(qv),all),
             all(vr(qv),qcnftoprop(qcnf(qvl,cn))),
             exs(vr(qv),qcnftoprop(qcnf(qvl,cn))))
```

5.7 Auxiliary types

Definition of the sort `BindSet`

The sort `BindSet` is the type in which the truth assignment is represented. A member of the sort `BindSet` generates two sets of variables: *Tvars* and *Fvars*. *Tvars* is interpreted as the set of variables to which the value *true* is assigned and *Fvars* is interpreted as the set of variables to which the value *false* is assigned.

The function $bindset : VarSet \times VarSet \mapsto BindSet$ with as first argument vs_1 and as second argument vs_2 generates the sets *Tvars* and *Fvars* such that $Tvars \cup Fvars = vs_2$, and that for all $v \in vs_2$ it holds that:

if $v \in vs_1$ then $v \in Tvars$ else $v \in Fvars$.

μCRL definition of the sort `BindSet`

```
sort    BindSet
func    bindset:  VarSet × VarSet → BindSet
        Tvars:  BindSet → VarSet
        Fvars:  BindSet → VarSet
var     vs1,vs2:  VarSet
rew     Tvars(bindset(vs1,vs2)) = vs1
        Fvars(bindset(vs1,vs2)) = diff(vs2,vs1)
```

Definition of the sorts `ResultSC`, `Status` *and* `Command`

In the process specifications of the solver components various control signals are used. `ResultSC` is used by the Satisfiability Checker for communicating the result of a check. The signals of sort `Status` are used for notifying the manager process whether something has gone wrong in a component during its calculations. Signals of sort `Command` represent commands from the manager to the components.

μCRL definition of the sorts `ResultSC`, `Status` *and* `Command`

```
sort    ResultSC
sort    Command
```

```
sort      Status
func      satisfiable:  → ResultSC
          unsatisfiable:  → ResultSC
          restart:  → Command
          terminate:  → Command
          timeout:  → Command
          notokmsg:  Nat → Status
```

6. Concluding remarks

The specifications provided in this document were found correct by the μCRL syntax checker from the PSF toolset [MV93, Vel95]. Data types were analysed with the rewriter and the initial algebra calculator tool. Moreover, some analysis of the behaviour of the various processes was done with the PSF simulator.

Our first concern was to give brief and functional specifications of components and tools for proposition solvers. Various implementation details are omitted, and left to the programmer.

Therefore, the specifications of the data types are not complete, in the sense that we did not specify certain functions in detail; we preferred to only specify the requirements of *equisatisfiability* on these functions.

A full specification would require much and complex work, whereas it would not contribute to a better understanding of the individual solver components and their interaction with the manager process. We think, however, that various data types are specified in a clear way, so that the specifications may facilitate the implementation.

For the near future, implementation of a deadlock checker for μCRL specifications is planned. A first application of the methods provided in this document may be the specification of such a tool.

Acknowledgements. The authors would like to thank Jan Friso Groote and Sjouke Mauw for their helpful comments. Also thanks to Sjouke for his help with the PSF toolset.

References

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[BK95] J.A. Bergstra, P. Klint. The discrete time toolbus. Technical Report P9502. Programming Research Group. University of Amsterdam, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[DG95] D. Dams and J.F. Groote. *Specification and Implementation of Components of a μCRL toolbox.* Number 152 in Preprint Series of Logic Group, Utrecht Research Institute for Philosophy, 1995.

[GKV95] J.F. Groote, J.W.C. Koorn and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd (Extended abstract). In *Proceedings 10th Annual Conference on Computer Assurance (COMPASS '95)*, pp 57-68, Gaithersburg, Maryland, 1995.

[GP94] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, editors. *Proceedings of ACP '94*, Utrecht, The Netherlands. Workshops in Computing, Springer-Verlag, pp 26-62, 1994.

[Gro97] J.F. Groote. The propositional formula checker HeerHugo. *Unpublished.* CWI, Amsterdam, 1997.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols.* Cambridge Tracts in Theoretical Computer Science 36, Cambridge University Press, 1993.

[NS93] A. Nerode and R. Shore. *Logic for applications.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Vel95] G.J. Veltink. *Tools for PSF.* Ph.D. thesis, Department of Mathematics and Computer Science, University of Amsterdam, 1995.

[Wam95] J.J. van Wamel. *Verification Techniques for Elementary Data Types and Retransmission Protocols.* Ph.D. thesis, Department of Mathematics and Computer Science, University of Amsterdam, 1995.