Specification and Implementation of Components of a μCRL Toolbox

Dennis Dams & Jan Friso Groote

Department of Philosophy, Utrecht University

Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

email: dams@phil.ruu.nl, jfg@phil.ruu.nl

Abstract

We develop a set of tools to translate linear μCRL specifications to finite transition systems. These tools are intended to function as the core of larger toolsets comprising model checkers, weak or branching bisimulation checkers, simulators, etc. The main problem in constructing these larger toolsets is to manage the software development process, especially, when many different people are contributing. Therefore, and this makes our approach unique, we describe specifications and implementations of the tools formally in μCRL. We realise the tools in C and let them communicate via the Toolbus.

1 Introduction

It is a major problem to keep large software development projects manageable. Typically, such projects end up in a situation where there is a lot of code, a lot of not too consistent documentation and many experts, who all have knowledge about parts of the software. In such cases it has become rather hard to make adaptations to the software and rather expensive to train a new expert. As both appear unavoidable, these make maintaining software a costly operation.

For quite a while the formal method community advocates the use of 'formal methods' as a solution. It is claimed that by formally specifying the software better products result, which can moreover be easier maintained and adapted. Within the formal community many projects have been carried out that seem to confirm this finding [7, 10].

However, it is almost impossible to draw firm conclusions and learn from these experiments, for the following reasons. Most of these projects were only targeted at parts of the software development process, for some limited amount of time. Often the actual realisation of the software was not conforming to the specifications; adaptations were made without adapting the formal specifications, new features introduced without formal backup etc. On the other hand it happened that formal specifications were made without sufficiently taking the desired properties of software into account. Understandably, despite the elegance of such specifications on their own, these have had little impact.

Carrying out new experiments with formal methods is of little use if we cannot avoid the problems sketched above. Hence, any new experiment must span the whole life cycle of some software product, including removing bugs and adapting its functionality to 'new wishes'. Unavoidably, this means that this must be a large project, that takes several years.

In order to guarantee that the realisation of the software, i.e., the actual code, conforms the specification, we must have the means to enforce the programmers to adhere to the formal specifications. In order to do so, they must be able to read and understand the specifications.

The software that is built must also be of interest of its own. Otherwise, the importance of the use of formal methods may supersede the value of the end product. Moreover, it is needed to defend the investment of manpower.

For all the reasons mentioned above we decided to start working on a toolset for μCRL, which should be comparable to existing toolsets [8, 16, 5, 6, 15]. The language μCRL is a simple process algebraic language with equationally specified datatypes suitable for both analysis and specification of distributed systems [11, 12]. All specifications of the components of the toolset are made in this language.

But we must realise that our primary contribution is structural formal development, and not fast implementation. This means that we are very willing to borrow from others. For instance the Representant Generator (see Section 2) can very well be built on top of an existing term rewriting system. Existing toolsets may with some adaptations be connected to the toolbus, and henceforth provide extended functionality in a cheap way.

The initial tools that we make translate linear μCRL specifications into explicit (i.e., fully expanded) transition systems. We call this set of tools an *instantiator*. The transition systems that are obtained may serve as input to other tools like a model checker, a (bisimulation-) minimiser, a (bisimulation-) equivalence checker, etc. The assumption that the μCRL specifications are in linear format is expected to allow for simplifications in the design of the instantiator. In order to make the toolset suitable for arbitrary μCRL specifications, a tool developed by Doeko Bosscher and Alban Ponse may be used, see [4]. This tool transforms a large class of specifications to linear format.

We let the tool communicate via the toolbus [1]. The toolbus allows to connect a number of tools that can exchange terms, and some elementary data terms. This is very convenient, because it is very close to the μCRL specification style, where processes communicate terms. Moreover, the communication style of toolbus programmes can be made to match the μCRL communications. There is one disadvantage of the toolbus, and that is that it is relatively slow; currently it can exchange about 10 messages between tools per second. It has been suggested that this will be improved.

In the subsequent sections we proceed by first explaining the general structure of the Instantiator. Then we provide specifications and implementations of the different tools in μCRL. These implementations have all been straightforwardly realised in C (see Appendix C), and the toolset is available for experiments.

2 The general structure

Our toolset consists of four components. The *instantiator* enumerates the reachable state space, given a specification (in linear form) and a start state. In doing so, it relies on the *Stepper* which, when given a state s, returns the outgoing action/state pairs in the form of a list. In order to determine whether two states are equal with respect to the theory specified in the abstract data type belonging to a process, which is needed to find out whether some state to be explored has already been visited, the instantiator has access to a *Representant Generator*, which returns representants for every state that it is given, in such a way that states which are *not* equal are given different representants. Finally, there is an Enumerator which returns valid instantiations of a given boolean term. This component is used by the Stepper, to enumerate all dataterms in a sum that satisfy a conditional. The Enumerator also uses a Representant Generator.

The hierarchical decomposition is given in Figure 1. For completeness a coordinator (COOR) of the tools is included, which uses a statical semantical checker (SSC) such as for instance provided in [14] and a linearisator (Lin), for instance one that is based on [4]. Neither the coordinator nor these tools are described in this document.

The names of the channels will be used in the specifications. Note that channels are unidirectional. For a channel x, the actions s_x and r_x, denoting a send and receive action on this channel respectively, have to synchronise to the action c_x. This convention trivially determines the communication functions on actions, which can be straightforwardly be specified by:

comm $r_{c1}|s_{c1} = c_{c1}$

$r_{c2}|s_{c2} = c_{c2}$
$r_{e1}|s_{e1} = c_{e1}$
$r_{e2}|s_{e2} = c_{e2}$
$r_{r1}|s_{r1} = c_{r1}$
$r_{r2}|s_{r2} = c_{r2}$
$r_{s1}|s_{s1} = c_{s1}$
$r_{s2}|s_{s2} = c_{s2}$

Note that we only give the communication between those components that we have specified.

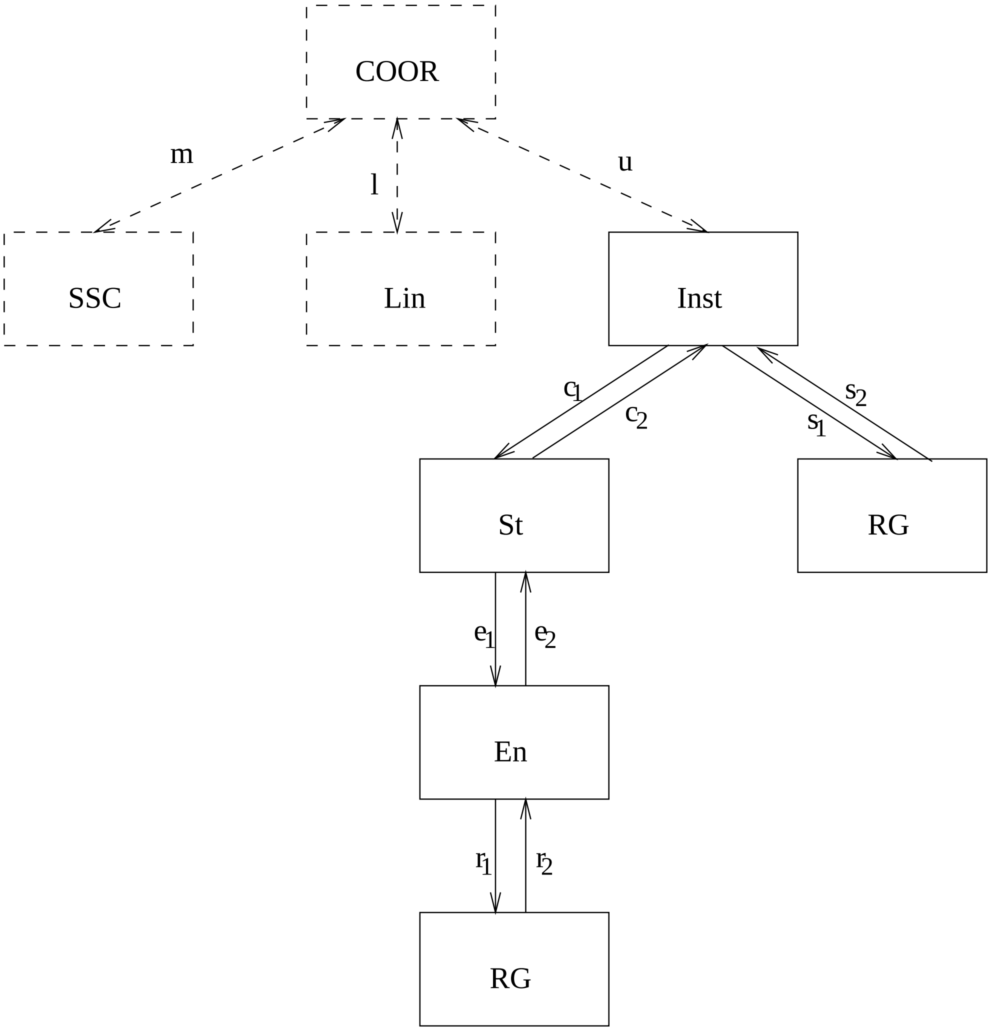

Figure 1: Decomposition of the instantiator.

For all four components we provide a specification of the external behaviour. In order to use a component of the toolset, it is sufficient to have knowledge of this external behaviour only. But as explained above all four tools except the Representant Generator use other tools to realise their functionality. We describe for these tools implementations in μCRL in which it is explained how the specification can be realised (we use here terminology introduced by Blaauw [3]. He calls a *specification* a description of the functionality of a system. An *implementation* is a high level description of the structure of a piece of hardware or software that provides this functionality. A *realisation* is the actual hardware of software providing the functionality). Whereas the specifications are exact prescriptions of the functionality of the tools, the implementations are mere suggestions towards realisations.

For the Enumerator, Stepper and the Instantiator the implementations should provide the functionality prescribed by the specification. This can formally be described by:

- The Enumerator can be constructed using the specification of the Representant Generator and the implementation of the Enumerator. So, the specification of the Enumerator must be equal in the sense of weak or branching bisimulation to the specification of the Representant Generator in parallel with the implementation of the Enumerator, after abstraction from internal actions.

$$EnS_0 = \tau_{\{c_{r1},c_{r2}\}}\partial_{\{r_{r1},r_{r2},s_{r1},s_{r2}\}}(RgS_0 \parallel EnI_0)$$

- The specification of the Enumerator and the implementation of the Stepper form the specification of the Stepper.

$$StS_0 = \tau_{\{c_{e1},c_{e2}\}}\partial_{\{r_{e1},r_{e2},s_{e1},s_{e2}\}}(EnS_0 \parallel StI_0)$$

- Likewise, the specification of the Instantiator must be equal to the implementation of the Instantiator, in combination with the specifications of the Stepper and the Representant Generator.

$$StS_0 = \tau_{\{c_{c1},c_{c2},c_{s1},c_{s2}\}}\partial_{\{r_{c1},r_{c2},s_{c1},s_{c2},r_{s1},r_{s2},s_{s1},s_{s2}\}}(InS_0 \parallel StS_0 \parallel \rho_{\{r1\to s1,r2\to s2\}}(RgS_0))$$

The definitions of all these processes are given in the subsequent text. We have not yet proven these equations, but using the techniques put forward in [13] we think that this should not be too hard.

The components are described separately in the Sections 3 through 6. Each section starts with an informal description of the component's intended behaviour and then gives a specification in μCRL. We have constructed prototypes in C of all implementations that we provide, which actually appeared a very straightforward activity.

Also included, in the appendix, is the definition in μCRL of all datatypes, most of which are accompanied by short informal descriptions. Furthermore, we provide all C code and the toolbus script in Appendix C.

3 The Representant Generator

3.1 Informal description

The Representant Generator receives an equationally specified datatype and a sequence of terms. It returns for each term that it receives a representant that is equal (w.r.t. the datatype — in the sequel we sometimes call this *equivalent* to avoid confusion with the notion of syntactic equality) to the input term. Ideally, the representant should uniquely represent the equivalence class to which the input term belongs. But as calculating such a representant is generally undecidable, we only require that for any term that is offered to the Representant Generator, an equivalent term must be returned, if anything is returned at all. So, a trivial implementation could always decide to return the term itself as its own representant, and neatly fit the specification.

An obvious alternative is to specify that the *normal form* of the term is returned as its representant in case it can be calculated. The reason for not requiring this is that we do not want to restrict the technology to implement a Representant Generator. For instance, one might want to apply Knud-Bendix completion, or redirect equations before applying rewriting, in order to yield better representants or to improve efficiency of this tool. Moreover, we would like the tool to be applicable also to datatypes that are not confluent and/or not terminating.

We actually leave maximal freedom to the implementors to gradually improve the quality of a Representant Generator, and simultaneously, guarantee the users of a Representant Generator a minimal functionality.

3.2 Specification of the Representant Generator

The detailed description of the Representant Generator is below. It first receives a datatype, which is described in appendix A, and checks whether the datatype satisfies a number of statical and semantical correctness conditions. Note that we specify, using c_0, that if the datatype is statically incorrect, it must be rejected, using an error message ($message_0$) of which the contents is not specified. If the datatype is correctly typed, it may still be rejected for reasons dependent on the implementation or realisation; e.g., the datatype may be too large to store in memory, or the equations may not be confluent, etc.

If the datatype is considered acceptable, the Representant Generator returns an *ok* message and proceeds with an event loop in which it can receive either an open or closed term, or a restart or terminate command. For each term that it receives, it again checks certain wellformedness conditions and returns appropriate error messages if necessary ($message_1$, $message_2$). It is specified that an error must be returned if a term is not correctly typed, but again any implementation of a Representant Generator may decide to reject a term on any other ground also (see c_1 and c_2). But if the term is accepted, a representant is returned. The only constraint that we impose is that if the Representant Generator returns a term it must be equivalent to the input data term.

Atomic actions are always send (s) or receive (r) actions. These are subscripted by the name of the channels along which the communication takes place (see Figure 1). Representant Generator receives messages via channel r_1 and sends them via r_2.

func $message_0 : Datatype \to String$
$message_1 : Term \times Datatype \to String$
$message_2 : Openterm \times Datatype \to String$
$c_0 : Datatype \to Bool$
$c_1 : Term \times Datatype \to Bool$
$c_2 : Openterm \times Datatype \to Bool$
$rep : Term \times Datatype \to Term$
$rep : Openterm \times Datatype \to Openterm$

var $d : Datatype$
$t : Term$
$t' : Openterm$

rew $implies(c_0(d), ssc(d)) = \mathsf{t}$
$implies(c_1(t,d), and(closed(t, get_{sig}(d)), equal(t, rep(t,d), d))) = \mathsf{t}$
$implies(c_2(t',d), and(correctopenterm(t', get_{sig}(d)), equal(t', rep(t',d), d))) = \mathsf{t}$

act $r_{r1}, c_{r1} : Datatype$
$r_{r1}, c_{r1} : Command$
$s_{r2} : Status$
$r_{r1}, c_{r1}, s_{r2} : Term$
$r_{r1}, c_{r1}, s_{r2} : Openterm$

proc $RgS_0 =$
$\sum_{d:Datatype} r_{r1}(d) \cdot (s_{r2}(ok) \cdot RgS_1(d) \lhd c_0(d) \rhd s_{r2}(notok(message_0(d))) \cdot RgS_0) +$
$r_{r1}(restart) \cdot RgS_0 +$
$r_{r1}(terminate)$

$$
\begin{array}{l}
RgS_1(d{:}Datatype) = \\
\quad \sum_{t:Term} r_{r1}(t) \cdot \\
\qquad (s_{r2}(rep(t,d)) \cdot RgS_1(d) \triangleleft c_1(t,d) \triangleright s_{r2}(notok(message_1(t,d))) \cdot RgS_1(d)) + \\
\quad \sum_{t:Openterm} r_{r1}(t) \cdot \\
\qquad\quad (s_{r2}(rep(t,d)) \cdot RgS_1(d) \triangleleft c_2(t,d) \triangleright s_{r2}(notok(message_2(t,d))) \cdot RgS_1(d)) + \\
\quad r_{r1}(restart) \cdot RgS_0 + \\
\quad r_{r1}(terminate)
\end{array}
$$

3.3 Realisation of the Representant Generator

In the prototype of the Instantiator, the Representant Generator has been built in a rather simplistic fashion. The equations of the abstract datatype are interpreted as directional rewrite rules. Each term is rewritten by applying to all subterms the textually first applicable rule. This is done a fixed number of times. The C-code is included in Appendix C.

4 The Enumerator

4.1 Informal description

Having received an abstract datatype (which, when correct, is acknowledged by sending an *ok* message), the Enumerator receives open terms of type *Bool* and for each such term sends back either (1) a list of all substitutions that make the term true (only if there is a finite number of such substitutions), or (2) a message indicating that it does not send back any such substitution at all.

Received datatypes are at least checked for statical and semantical correctness (*ssc*). If the check does not succeed, the Enumerator returns an explanatory message and waits for a new datatype. Terms are checked to be correct open terms of type *Bool*.

At any point, the Enumerator may also receive a message *restart* or *terminate*; in this case, it restarts and is ready to receive a new datatype, or terminates.

4.2 Specification of the Enumerator

The specification of the Enumerator is actually very simple, especially compared to a possible implementation which is given below. Basically it receives via channel $e1$ a datatype which it at least checks for static semantic correctness using c_3. In the same way as with the Representant Generator the condition c_3 is underspecified, to allow the Enumerator to reject datatypes that are too large or have other unacceptable properties, depending on constraints put forward by the implementation and realisation. If the datatype is rejected, a *notok* with $message_3$ is sent back via channel $e2$. Otherwise an *ok* is reported and the Enumerator is ready to receive open terms.

After acceptance of the datatype, the Enumerator receives open terms. Using c_4 it checks that these open terms are at least correctly typed and of sort *Bool*. Moreover, c_4 can only be true if there are only a finite number of substitutions that make the open term equal to true. In case there are a finite number of instantiations, these are provided by $satlist(c,d)$, which is sent back via channel $e1$. The function *satlist* yields a *Termlistlist*. Each termlist in *satlist* provides consecutively the terms that must be substituted for the variables in the variablelist $get_{vars}(c)$.

At any time the Enumerator can be asked to terminate, or to restart in its initial state.

$$
\begin{array}{ll}
\textbf{func} & c_3 : Datatype \rightarrow Bool \\
& c_4 : Openterm \times Datatype \rightarrow Bool \\
& message_3 : Datatype \rightarrow String \\
& message_4 : Openterm \times Datatype \rightarrow String
\end{array}
$$

$satlist : Openterm \times Datatype \rightarrow Termlistlist$

var $c : Openterm$
$d : Datatype$

rew $implies(c_3(d), ssc(d)) = \mathsf{t}$
$implies(c_4(c,d), eq(whatsort(c, get_{sig}(d)), Bool)) = \mathsf{t}$
$implies(c_4(c,d), correctopenterm(c, get_{sig}(d))) = \mathsf{t}$

We are not able to specify the essential properties of the function *satlist* in the μCRL style. Therefore, *satlist* is specified in ordinary mathematical terms, as follows. Note that we also add some constraints to c_4.

Let $c{:}Openterm$ and $d{:}Datatype$. If $c_4(c,d)$ is true, then $satlist(c,d)$ is a termlistlist satisfying the following two requirements:

1. Soundness: Each substitution in $satlist(c,d)$ must make the open term c true. In semi μCRL: for each termlist tl in $satlist(c,d)$, we have $equal(subst(tl, get_{vars}(c), get_{term}(c)), T(\mathsf{t}, emt), d)$.
2. Completeness: Each substitution that can make c true, must find an equivalent substitution in *satlist*. In semi μCRL we can formulate this by: for each termlist tl' such that $equal(subst(tl', get_{vars}(c), get_{term}(c)), T(true, emt), d)$, there exists a termlist tl in $satlist(c,d)$ such that $equal(tl, tl', d)$.

Note that if there are an infinite number of different substitutions that make c true, then c_4 must become false, as *satlist* cannot be infinite. The reason for this is that by the semantics of μCRL, the sort *Termlistlist* only contains finite lists.

act $r_{e1}, c_{e1} : Datatype$
$r_{e1}, c_{e1} : Command$
$r_{e1}, c_{e1} : Openterm$
$s_{e2} : Status$
$s_{e2} : Termlistlist$

proc $EnS_0 =$
$\sum_{d:Datatype} r_{e1}(d) \cdot (s_{e2}(ok) \cdot EnS_1(d) \triangleleft c_3(d) \triangleright s_{e2}(notok(message_3(d))) \cdot EnS_0) +$
$r_{e1}(restart) \cdot EnS_0 +$
$r_{e1}(terminate)$

$EnS_1(d{:}Datatype) =$
$r_{e1}(restart) \cdot EnS_0 +$
$r_{e1}(terminate) +$
$\sum_{c:Openterm} r_{e1}(c) \cdot (s_{e2}(satlist(c,d)) \triangleleft c_4(c,d) \triangleright s_{e2}(notok(message_4(c,d)))) \cdot EnS_1(d)$

4.3 Implementation of the Enumerator

We provide a possible implementation of the Enumerator.

The main idea behind our implementation of the Enumerator is to instantiate the input term c of sort *Bool* with terms consisting of constructors that have a limited amount of nesting. The technique seems easiest explained by an example. Consider the natural numbers with constructors zero (0) and the successor (S), and the term $x < S(S(0))$ of type *Bool* [1]. First we try to prove that $x < S(S(0))$ is equal to false, for which we use the Representant Generator. For this example we assume that

[1] For readability we do not adhere to the strict μCRL syntax in these paragraphs.

the Representant Generator is rather powerful, in the sense that it can reduce all terms of type *Bool* below to true or false if they are universally true, or false. In the implementation given below however, the Representant Generator may always "give up", in which case also the Enumerator will return a *notok* message. Suppose that $x < S(S(0))$ can be proven equal to false, then we know that there is no substitution making $x < S(S(0))$ true, and we can reply by sending the empty substitution list. However, as clearly $x < S(S(0))$ is not false under the standard interpretation of $<$, we replace x by all terms consisting of a constructor applied to fresh variables. We call this step an *expansion*. In this case we get the two terms $0 < S(S(0))$ and $S(y) < S(S(0))$. The first instantiation clearly reduces to true, and hence $[x := 0]$ is the first substitution that we have found. In the the second case, it is impossible to prove $S(y) < S(S(0))$ false, and hence we iterate the instantiation of variables. So, we get $S(0) < S(S(0))$ and $S(S(z)) < S(S(0))$. The first is clearly true, yielding $[x := S(0)]$, and the second is generally false, indicating that there are no substitutions of the form $[x := S(S(\ldots))]$. Hence, we are done. In general this search procedure does not terminate. Therefore, we limit this search by a maximal number of rounds, which is defined by the (unspecified) variable *rounds*.

There are a number of ways to perform expansions. We choose for one possibility which we have specified using the function *Expand*. This function receives as input a "current instantiation" t of the original open term c and computes a list of "next instantiations" to be considered. These are obtained by consecutively considering each variable x in t, and adding to the list all instantiations of x consisting of a single constructor function, with fresh variables as arguments (if any). Hence, the instantiations are computed in a breadth-first fashion. *Expand* yields an instantiation list, of sort *itlist*, which is a list of triples. Each triple represents an instantiation t' of t, obtained by instantiating some variable x of t. The first element, of sort *Variablelist* contains the variables of t' that may be further instantiated (in the next "deeper" level of the breadth-first scheme). The second element of the triple is a *Termlist* that contains the terms that are to be substituted for the variables of the original term c (of which t is an instantiation already) in order to obtain the next instantiation t'. The third element denotes the instantiation t'. E.g., for the example above, *Expand* applied to $x < S(S(0))$ yields the list

$$\langle [\langle\rangle, \langle 0\rangle, 0 < S(S(0))], [\langle y{:}Nat\rangle, \langle S(y)\rangle, S(y) < S(S(0))]\rangle$$

i.e., the two instantiations, 0 substituted for x and $S(y))$ substituted for x. Note that we use ad hoc $\langle\ldots\rangle$ notation to denote lists.

The function $Expand(vl_1, vl_2, inst, t, sig, next)$ yields an *itlist* that is an extension of *next*, with instantiations of the *Term* t. The arguments vl_1 and vl_2 keep track of, respectively, which of t's variables have already been considered and which of them have not. The *Termlist* *inst* represents all current instantiations that have already been obtained for the variables of c. The variable *sig* is the current *Signature*.

So, if vl_2 is empty, there is nothing to instantiate, and *Expand* can simply yield *next*. If vl_2 is not empty, i.e., contains at least a variable x of sort s then for each constructor f in the signature with targetsort s a term $f(x_1, \ldots, x_m)$ must be substituted for x, where $x_1, \ldots, x_m$ are fresh variables. This is the task of $Expand_1(s, x, vl_1, fns, vl_2, inst, t, sig, next)$. This function replaces the variable x of sort s in the term t by a single application of a constructor in *fns* and adds this instantiation to the *itlist* next.

In the definition of $Expand_1$ the functions *fresh*, *subst* and mk_{terms} are used, that are all defined in Appendix A. The function $fresh(args, vl, sig)$ generates a list of variables of sorts given in the *Stringlist* *args*, such that there is no clash with variables in the *Variablelist* *vl* and constants in the *Signature* *sig*. The function *subst* replaces a *Term* for a variable in a *Term*/*Termlist*. And finally the function mk_{terms} transforms a list of variables (which are just strings n) to a list of terms (of the form $T(n, emt)$).

The implementation is now rather straightforward, and described as the processes $EnI_0, \ldots, EnI_4$ below. The activities in processes EnI_0, EnI_1 deal with receiving a datatype and reacting adequately to *terminate* messages. We do not explain this.

The behaviour becomes less straightforward when an *Openterm* c is received (in EnI_2). If it is accepted, i.e., if $c_5(c, sig)$ is true, $EnI_3(itl, sig, results, n, nextround)$ is executed. In *results*, satisfying instantiations of c are accumulated. In *itl* partial instantiations of c are stored for which satisfaction must still be checked and that can be further instantiated. In *nextround* partial instantiations of c are stored that only need to be further instantiated. The variable n stores the number of remaining nested instantiations. So if EnI_3 is invoked in EnI_2, the variable c is put in *itl* as satisfaction of c must still be checked, and c can be further instantiated. The variables *nextround* and *results* are empty.

If in EnI_3 *itl* is not empty, its head is sent to the Representant Generator to find out whether it equals true or false. In EnI_4 the answer is collected, and processed. If *itl* is empty, then substitutions of more deeply nested terms are considered, except if $n = 0$ or *nextround* is empty. If nextround is empty there are no satisfying more deeply nested terms, and apparently, *results* contains all substitutions that make c true. Hence, these can be delivered. If $n = 0$ and *nextround* is not empty, further search for more substitutions is stopped, although there could be more possible candidates. In this case $notok(message_6)$ is reported. Otherwise, the elements of nextround are put to be processed and n is decremented.

In EnI_4 the answer provided by the Representant Generator is processed. If it reports failure, via a *notok*, this is forwarded, and processing of c is stopped. If it sends back an *Openterm*, this term can be either true or false, or any other term, not syntactically equal to true or false. The following case distinctions are made, that match those described in EnI_4.

- In case the answer is false, the suggested instantiation is not a possible candidate; hence it can be ignored.
- If the term that was sent to the Representant Generator is closed and the result that is received is true, then a satisfying instantiation has been found which is added to the *Termlistlist results*.
- If the term that was sent to the Representant Generator is closed and if none of the cases above apply (so the answer from the Representant Generator is neither true nor false), the Enumerator cannot decide whether the current instantiation should be ignored or added to the results, nor can it obtain this information by instantiating the variables further. So, it decides to report $notok(message_7(t))$.
- If none of the cases above apply, the suggested instantiation should be made more precise using the function *Expand*. However, it can be that there are no variables for which constructors can be substituted. In this case the Enumerator cannot determine what to do with this instantiation, and an error is reported ($message_8$).
- Now, the function *Expand* can be applied, and the instantiations obtained in this way can be prepared to be considered in a next round.

A realisation of the Enumerator has been constructed according to the implementation that is described below

func $rounds :\rightarrow Nat$
$message_5 : Openterm \times Signature \rightarrow String$
$message_6, message_8 :\rightarrow String$
$message_7 : Openterm \rightarrow String$
$c_5 : Openterm \times Signature \rightarrow Bool$
$Expand : Variablelist \times Variablelist \times Termlist \times Term \times Signature \times itlist \rightarrow itlist$
$Expand_1 : String \times String \times Variablelist \times Functionlist \times$
$\quad Variablelist \times Termlist \times Term \times Signature \times itlist \rightarrow itlist$
var $n, x, s, targ : String$

$args : Stringlist$
$fns : Functionlist$
$sig : Signature$
$vl_1, vl_2 : Variablelist$
$t : Term$
$inst : Termlist$
$next : itlist$
$c : Openterm$

rew $c_5(c, sig) = and(eq(whatsort(c, sig), Bool), correctopenterm(c, sig))$
$Expand(vl_1, emv, inst, t, sig, next) = next$
$Expand(vl_1, ins(x, s, vl_2), inst, t, sig, next) =$
$\quad Expand_1(s, x, vl_1, get_{cons}(sig), vl_2, inst, t, sig, Expand(ins(x, s, vl_1), vl_2, inst, t, sig, next))$
$Expand_1(s, x, vl_1, emf, vl_2, inst, t, sig, next) = next$
$Expand_1(s, x, vl_1, ins(F(n, args, targ), fns), vl_2, inst, t, sig, next) =$
$\quad if(eq(s, targ),$
$\quad\quad Expand_1(s, x, vl_1, fns, vl_2, inst, t, sig,$
$\quad\quad\quad ins(conc(vl_1, conc(vl_2, fresh(args, conc(vl_1, vl_2), sig))),$
$\quad\quad\quad\quad subst(T(n, mk_{terms}(fresh(args, conc(vl_1, vl_2), sig))), x, inst),$
$\quad\quad\quad\quad subst(T(n, mk_{terms}(fresh(args, conc(vl_1, vl_2), sig))), x, t), next)),$
$\quad\quad Expand_1(s, x, vl_1, fns, vl_2, inst, t, sig, next))$

act $s_{r1} : Datatype$
$s_{r1} : Command$
$s_{r1} : Openterm$
$s_{r1} : Term$
$r_{r2}, c_{r2} : Openterm$
$r_{r2}, c_{r2} : Term$
$r_{r2}, c_{r2} : Status$

proc $EnI_0 =$
$\quad \sum_{d:Datatype} r_{e1}(d) \cdot s_{r1}(d) \cdot EnI_1(get_{sig}(d)) +$
$\quad r_{e1}(restart) \cdot EnI_0 +$
$\quad r_{e1}(terminate) \cdot s_{r1}(terminate)$

$EnI_1(sig{:}Signature) =$
$\quad r_{r2}(ok) \cdot s_{e2}(ok) \cdot EnI_2(sig) +$
$\quad \sum_{s:String} r_{r2}(notok(s)) \cdot s_{e2}(notok(s)) \cdot EnI_0$

$EnI_2(sig{:}Signature) =$
$\quad r_{e1}(restart) \cdot s_{r1}(restart) \cdot EnI_0 +$
$\quad r_{e1}(terminate) \cdot s_{r1}(terminate) +$
$\quad \sum_{c:Openterm} r_{e1}(c) \cdot$
$\quad\quad (EnI_3(ins(get_{vars}(c), mk_{terms}(get_{vars}(c)), get_{term}(c), emit), sig, emtll, rounds, emit)$
$\quad \lhd c_5(c, sig) \rhd s_{e2}(notok(message_5(c, sig))) \cdot EnI_2(sig))$

$EnI_3(itl{:}itlist, sig{:}Signature, results{:}Termlistlist, n{:}Nat, nextround{:}itlist) =$
$\quad s_{r1}(O(hd_{term}(itl), hd_{vars}(itl))) \cdot EnI_4(itl, sig, results, n, nextround)$
$\quad \lhd not(empty(itl)) \rhd$
$\quad (s_{e2}(results) \cdot EnI_2(sig)$
$\quad\quad \lhd empty(nextround) \rhd$
$\quad (s_{e2}(notok(message_6)) \cdot EnI_2(sig)$

$$
\begin{array}{l}
\quad\quad \lhd\, eq(n,0) \rhd \\
\quad\quad EnI_3(nextround, sig, results, pred(n), emit)))
\end{array}
$$

$$
\begin{array}{l}
EnI_4(itl{:}itlist, sig{:}Signature, results{:}Termlistlist, n{:}Nat, nextround{:}itlist) = \\
\quad \sum_{s:String} r_{r2}(notok(s)) \,.\, s_{e2}(notok(s)) \,.\, EnI_2(sig) + \\
\quad \sum_{t:Openterm} r_{r2}(t) \,. \\
\quad\quad (EnI_3(tail(itl), sig, results, n, nextround) \\
\quad\quad\; \lhd\, eq(get_{term}(t), T(false, emt)) \rhd \\
\quad\quad\; (EnI_3(tail(itl), sig, ins(hd_{insts}(itl), results), n, nextround) \\
\quad\quad\;\; \lhd\, and(eq(get_{term}(t), T(true, emt)), closed(hd_{insts}(itl), sig)) \rhd \\
\quad\quad\;\; (s_{e2}(notok(message_7(t))) \,.\, EnI_2(sig) \\
\quad\quad\;\;\; \lhd\, closed(hd_{insts}(itl), sig) \rhd \\
\quad\quad\;\;\; (s_{e2}(notok(message_8)) \,.\, EnI_2(sig) \\
\quad\quad\;\;\;\; \lhd\, not(existsconstr(hd_{vars}(itl), get_{cons}(sig))) \rhd \\
\quad\quad\;\;\;\; EnI_3(tail(itl), sig, results, n, \\
\quad\quad\quad\quad\quad Expand(emv, hd_{vars}(itl), hd_{insts}(itl), get_{term}(t), sig, nextround)))))
\end{array}
$$

5 The Stepper

5.1 Informal description

The Stepper receives states from the instantiator. For each state s, it attempts to return all (action, state) pairs denoting the steps that are possible from s. In order to determine the steps that are possible from a given state, the Stepper needs a process specification, which must be provided first. A process specification is a somewhat extended linear process, which is described in extenso in Appendix B.

5.2 Specification of the Stepper

The specification of the Stepper is very simple. Actually, its only complex aspect is the description of *steplist*. It basically receives a specification and checks it using condition c_6. We underspecify condition c_6 by only requiring that if the specification is not statically semantically correct, *notok* must be reported. Again it is up to the implementor to determine under what other circumstances the input is rejected. In case the specification is accepted, the Stepper is prepared to receive a state, in the form of a list of terms and — after some checking, using condition c_7 — responds by sending a complete finite list of outgoing transitions of this state.

At any instance the Stepper can receive a request to terminate execution or to restart with a new specification.

$$
\begin{array}{ll}
\textbf{func} & message_9 : Spec \to String \\
& message_{10} : Termlist \times Spec \to String \\
& c_6 : Spec \to Bool \\
& c_7 : Termlist \times Spec \to Bool \\
& steplist : Termlist \times Spec \to Steplist \\
\textbf{var} & spec{:}Spec \\
& s{:}Termlist \\
\textbf{rew} & implies(c_6(spec), ssc(spec)) = \mathsf{t} \\
& implies(c_7(s, spec), \\
& \quad eq(get_{sorts}(get_{vars}(get_{proc}(spec))), whatsorts(s, emv, get_{sig}(get_{adt}(spec))))) = \mathsf{t} \\
& implies(c_7(s, spec), welltyped(s, emv, get_{sig}(get_{adt}(spec)))) = \mathsf{t}
\end{array}
$$

It is not possible to describe the required properties of *steplist* in μCRL. Therefore, we provide them in the semimathematical notation below.

Let *spec* be in *Spec* and s a state, i.e., s is an element of *Termlist* which matches with the parameter list of the process specification of *spec*. If $c_7(s, spec)$ holds, the following two properties must be true:

1. Soundness: for each triple $\langle a, tl, ntl\rangle$ in $steplist(s, spec)$ with a:*String* (an action name), tl: *Termlist* and ntl:*NTermlist*, there is a summand $SMD(vl, a, tl', ntl', c)$ in $get_{proc}(spec)$ with vl:*Variablelist*, tl':*Termlist*, ntl':*NTermlist* and c:*Term* such that for some *Termlist* tl'' with $whatsorts(tl'', emv, get_{sig}(get_{proc}(spec))) = get_{sorts}(vl)$ we have

$$\begin{array}{l} eq(subst(s, get_{vars}(get_{proc}(spec)), subst(tl'', vl, tl')), tl), \\ eq(subst(s, get_{vars}(get_{proc}(spec)), subst(tl'', vl, ntl')), ntl), \text{ and} \\ equal(subst(s, get_{vars}(get_{proc}(spec)), subst(tl'', vl, c)), T(true, emt)) \end{array}$$

2. Completeness: for each tl'':*Termlist* and each summand $SMD(vl, a, tl', ntl', c)$ with vl:*Variablelist*, a:*String*, tl':*Termlist*, ntl':*NTermlist* and c:*Term* such that

$$\begin{array}{l} equal(subst(s, get_{vars}(get_{proc}(spec)), subst(tl'', vl, c)), T(true, emt)) \text{ and} \\ whatsorts(tl'', emv, get_{sig}(get_{proc}(spec))) = get_{sorts}(vl) \end{array}$$

   there is some triple $\langle a, tl, ntl\rangle$ in $steplist(s, spec)$ such that

$$\begin{array}{l} subst(s, get_{vars}(get_{proc}(spec)), subst(tl'', vl, tl')) = tl \\ subst(s, get_{vars}(get_{proc}(spec)), subst(tl'', vl, ntl')) = ntl \end{array}$$

We are now ready to provide the specification of the Stepper.

$$\begin{array}{ll}
\textbf{act} & r_{c1}, c_{c1} : Spec \\
& r_{c1}, c_{c1} : Command \\
& r_{c1}, c_{c1} : Termlist \\
& s_{c2} : Status \\
& s_{c2} : Steplist \\
\textbf{proc} & StS_0 = \\
& \quad \sum_{spec:Spec}(r_{c1}(spec) \,. \\
& \qquad (s_{c2}(ok) \,.\, StS_1(spec) \lhd c_6(spec) \rhd s_{c2}(notok(message_9(spec))) \,.\, StS_0))+ \\
& \quad r_{c1}(restart) \,.\, StS_0 + \\
& \quad r_{c1}(terminate) \\
& \\
& StS_1(spec{:}Spec) = \\
& \quad r_{c1}(restart) \,.\, StS_0 + \\
& \quad r_{c1}(terminate) + \\
& \quad \sum_{s:Termlist} r_{c1}(s) \,. \\
& \qquad (s_{c2}(steplist(s, spec)) \lhd c_7(s, spec) \rhd s_{c2}(notok(message_{10}(s, spec)))) \,.\, StS_1(spec)
\end{array}$$

5.3 Implementation of the Stepper

Next, we suggest an implementation for the Stepper. Actually, it is the straightforward implementation, assuming the existence of the Enumerator. We provide a tour through the processes $StI_0, \ldots, StI_3$ below. We do neither explain the *terminate* and *restart* messages, nor do we explain the handling of faulty situations.

In state StI_0 the implementation expects a specification *spec*, which consists of a datatype and a linear process. It sends the datatype to the Enumerator, and waits for it to be accepted. If it is not accepted, the Stepper signals a problem. Otherwise, the Stepper checks whether the linear process is statically semantically correct (*ssc*) and reports accordingly.

If the specification *spec* is accepted, the implementation goes to state StI_1. It waits for a list of terms to be received, which represents the state for which outgoing transitions must be generated. We check that the terms in s are of correct sorts and well-typed. If so, and now we are in StI_2, we accumulate all action-state pairs in a list $results$.

For each summand that contains a summation over datatypes, we must know for which values an action can be performed. So, we send its condition to the Enumerator, requesting for all values for which the action can be performed. Note that the substitution of the terms in the "current state" s for variables in the condition can take place without checking whether these variables are not bound by the summation, as this has already been checked by *ssc* in StI_0 (see the function *nooverlap* in the definition of *ssc* in Section B.3). The list of values is received by the action $r_{e2}(tll)$, and the results are transformed to steps, which are, in StI_3, added to the results.

The main function in StI_3 is $add\_transition(smd(v, an, at, st, c), tl_1, vl_1, tl_2, vl_2, results)$. It adds a triple an, at', st' in front of the *Steplist* $results$, where an is a *String* denoting the name of an action, at' is a termlist denoting the arguments of the action, obtained from at where the terms of the *Termlist* tl_i are substituted for the variables in the *Variablelist* vl_i $(i = 1, 2)$. In the same way the *Termlist* st' is obtained from the termlist st which denotes a state that is reached after executing action an with arguments at'.

The implementation of the Stepper is given by

$$
\begin{array}{ll}
\textbf{func} & message_{11} : Spec \to String \\
 & message_{12} : Termlist \times Spec \to String \\
 & c_8 : Termlist \times Spec \to Bool \\
 & add\_transition : Summand \times Termlist \times Variablelist \times \\
 & \qquad\qquad\qquad Termlist \times Variablelist \times Steplist \to Steplist \\
\textbf{var} & sm : Summand \\
 & tl_1, tl_2 : Termlist \\
 & vl_1, vl_2 : Variablelist \\
 & results : Steplist \\
 & spec : Spec \\
\textbf{rew} & c_8(tl_1, spec) = \\
 & \quad and(eq(get_{sorts}(get_{vars}(get_{proc}(spec))), whatsorts(tl_1, emv, get_{sig}(get_{adt}(spec)))), \\
 & \qquad welltyped(tl_1, emv, get_{sig}(get_{adt}(spec)))) \\
 & add\_transition(sm, tl_1, vl_1, tl_2, vl_2, results) = \\
 & \quad ins(ST(get_{actname}(sm), \\
 & \qquad subst(tl_1, vl_1, subst(tl_2, vl_2, get_{actterms}(sm))), \\
 & \qquad subst(tl_1, vl_1, subst(tl_2, vl_2, get_{state}(sm)))), results) \\
\textbf{act} & s_{e1} : Datatype \\
 & s_{e1} : Command \\
 & s_{e1} : Openterm \\
 & r_{e2}, c_{e2} : Status \\
 & r_{e2}, c_{e2} : Termlistlist \\
\textbf{proc} & StI_0 = \\
 & \quad \sum_{spec:Spec} r_{c1}(spec) \cdot s_{e1}(get_{adt}(spec)) \cdot \\
 & \qquad (\sum_{s:String} r_{e2}(notok(s)) \cdot s_{c2}(notok(s)) \cdot StI_0 + \\
 & \qquad\ r_{e2}(ok) \cdot (s_{c2}(ok) \cdot StI_1(spec) \lhd ssc(spec) \rhd s_{c2}(notok(message_{11}(spec))) \cdot StI_0)) + \\
 & \quad r_{c1}(restart) \cdot StI_0 + \\
 & \quad r_{c1}(terminate) \cdot s_{e1}(terminate) \\
 & \\
 & StI_1(spec{:}Spec) =
\end{array}
$$

$$
\begin{aligned}
&r_{c1}(restart) \,.\, s_{e1}(restart) \,.\, StI_0 + \\
&r_{c1}(terminate) \,.\, s_{e1}(terminate) + \\
&\textstyle\sum_{s:\,Termlist} r_{c1}(s) \,. \\
&\quad (StI_2(s, get_{sums}(get_{proc}(spec)), spec, emsl) \lhd c_8(s, spec) \rhd \\
&\qquad\qquad s_{c2}(notok(message_{12}(s, spec))))
\end{aligned}
$$

$$
\begin{aligned}
&StI_2(s{:}\,Termlist, sl{:}\,Sumlist, spec{:}\,Spec, results{:}\,Steplist) = \\
&\quad s_{c2}(results) \,.\, StI_1(spec) \\
&\quad \lhd empty(sl) \rhd \\
&\quad s_{e1}(O(subst(s, get_{vars}(get_{proc}(spec)), get_{cond}(head(sl))), get_{vars}(head(sl)))) \,. \\
&\qquad (\textstyle\sum_{tll:\,Termlistlist} r_{e2}(tll) \,.\, StI_3(s, sl, spec, results, tll) + \\
&\qquad\ \textstyle\sum_{s_1:String} r_{e2}(notok(s_1)) \,.\, s_{c2}(notok(s_1)) \,.\, StI_1(spec))
\end{aligned}
$$

$$
\begin{aligned}
&StI_3(s{:}\,Termlist, sl{:}\,Sumlist, spec{:}\,Spec, results{:}\,Steplist, tll{:}\,Termlistlist) = \\
&\quad StI_2(s, tail(sl), spec, results) \\
&\quad \lhd empty(tll) \rhd \\
&\quad StI_3(s, sl, spec, add\_transition(head(sl), s, get_{vars}(get_{proc}(spec)), head(tll), \\
&\qquad get_{vars}(head(sl)), results), tail(tll))
\end{aligned}
$$

6 The Instantiator

6.1 Informal description

The instantiator first gets a process specification, i.e., a linear process and a datatype. Then it repeatedly receives a *Termlist* denoting an initial state. After the usual checking, it attempts to return a transition system containing all states that are reachable from s. A transition system consists of a triple $TS(d, s, tl)$ where d is the datatype, s the initial state and tl a finite list of transitions. Each transition consists of a source state, in the form of a *Termlist*, an action (*String*) with parameters (of sort *Termlist*) and a target state (*Termlist*).

6.2 Specification of the Instantiator

The specification of the behaviour of the instantiator is rather straightforward, except for the specification of the function *transsys*.

$$
\begin{array}{ll}
\textbf{func} & c_9 : Spec \rightarrow Bool \\
& c_{10} : Termlist \times Spec \rightarrow Bool \\
& message_{12} : Spec \rightarrow String \\
& message_{13} : Termlist \times Spec \rightarrow String \\
& transsys : Termlist \times Spec \rightarrow Transsys \\
\textbf{var} & s : Termlist \\
& spec : Spec \\
\textbf{rew} & implies(c_9(spec), ssc(spec)) = \mathsf{t} \\
& implies(c_{10}(s, spec), \\
& \quad eq(get_{sorts}(get_{vars}(get_{proc}(spec))), whatsorts(s, emv, get_{sig}(get_{adt}(spec))))) = \mathsf{t} \\
& implies(c_{10}(s, spec), welltyped(s, emv, get_{sig}(get_{adt}(spec)))) = \mathsf{t}
\end{array}
$$

For a specification *spec* and a *Termlist* s for which $c_{10}(s, spec)$ holds, we define $transsys(s, spec)$ as the triple $TS(get_{adt}(spec), s, tl)$ where tl is a finite *Translist* containing at least all reachable transitions from s prescribed by *spec* (completeness), and at most all allowed transitions (soundness). Note that there is freedom between this minimum and maximum. Firstly, tl may contain (syntactically) different transitions which indeed denote the same (wrt. the function *equal* defined in Section A.18 in the appendix) transition. Secondly, there may be unreachable transitions in tl.

Letting $sig = get_{sig}(get_{adt}(spec))$ and $al = get_{vars}(get_{proc}(spec))$, tl thus satisfies

1. (Soundness) For every transition $TR(t_1, a, at, t_2)$ in tl it holds that there is a summand $SMD(vl, a, at', t'_2, c)$ and a *Termlist* tl' such that
   - $closed(tl', sig) = \mathsf{t}$
   - $whatsorts(tl', emv, sig) = get_{sorts}(vl)$
   - $equal(subst(t_1, al, subst(tl', vl, c)), T(true, emt)) = \mathsf{t}$
   - $equal(subst(t_1, al, subst(tl', vl, at')), at) = \mathsf{t}$
   - $equal(subst(t_1, al, subst(tl', vl, t'_2)), t_2) = \mathsf{t}$
2. (Completeness) For every *Termlist* t that is either syntactically equal to s or that occurs as the rightmost argument of a *Transition* in tl it holds that for every summand $SMD(vl, a, at', t'_2, c)$ and for every *Termlist* t' such that
   - $welltyped(t', emv, sig) = \mathsf{t}$
   - $whatsort(t', emv, sig) = get_{sorts}(vl)$
   - $equal(subst(t, al, subst(t', vl, c)), T(true, emt)) = \mathsf{t}$

   there is a *Transition* $TR(t, a, at, t_2)$ in tl such that
   - $equal(subst(t, al, subst(t', vl, at')), at) = \mathsf{t}$
   - $equal(subst(t, al, subst(t', vl, t'_2)), t_2) = \mathsf{t}$

   Thus, for every reachable state s', there is a transition $TR(s', a'', at'', s'')$ for every (if any) step from s'. It is not sufficient if there is such a transition $TR(s'_1, a'', at'', s'')$ with s' and s'_1 *equal* (under the equations) but not syntactically equal. We call such a transition system *syntactically continuing*.

act $r_{u1} : Command$
$r_{u1} : Spec$
$r_{u1} : Termlist$
$s_{u2} : Status$
$s_{u2} : Transsys$

proc $InS_0 =$

$$\begin{aligned}
&\textstyle\sum_{spec:Spec} r_{u1}(spec) \,.\, (s_{u2}(ok) \,.\, InS_1(spec) \lhd c_9(spec) \rhd \\
&\qquad s_{u2}(notok(message_{12}(spec))) \,.\, InS_0) + \\
&r_{u1}(restart) \,.\, InS_0 + \\
&r_{u1}(terminate)
\end{aligned}$$

$InS_1(spec{:}Spec) =$

$$\begin{aligned}
&\textstyle\sum_{s:Termlist} r_{u1}(s) \,. \\
&\qquad (s_{u2}(transsys(s, spec)) \,.\, InS_0 \lhd c_{10}(s, spec) \rhd \\
&\qquad\qquad s_{u2}(notok(message_{13}(s, spec))) \,.\, InS_1) + \\
&r_{u1}(restart) \,.\, InS_0 + \\
&r_{u1}(terminate)
\end{aligned}$$

6.3 Implementation of the Instantiator

We provide a rather straightforward candidate for the implementation of the Instantiator. It uses a Stepper via actions s_{c1} and r_{c2} and a Representant Generator via actions s_{s1} and r_{s2}. After receiving, forwarding and checking a specification (in InI_0 and InI_1) the instantiator receives an initial state s of sort *Termlist* in InI_2. It then invokes InI_3 whose third argument tl is the list of transitions that must be delivered. The *Termlistlist*'s g and b contain the states for which the outgoing transitions are to be generated, and the states that have already been considered, respectively. Both the initial state s and the current datatype d have to be added to the transition system that must be delivered. The process InI_3 sends a *Termlist* taken from g to the Stepper to obtain all transitions leaving this state.

In InI_4 all elements of this list are prefixed to the transition list tl. There are two cases to be considered. If the target state of a step is *terminated*, i.e., the predefined terminated state, then the transition can easily be added. If, however, the target state is some termlist t', the terms in t' should be uniquely represented by replacing each of them by their representant. For this purpose, all terms in t' are sent to the Representant Generator (in InI_5). After t' is 'normalised' it is checked whether it already occurs in g or b, as this means that this state is already being considered. Otherwise it must be added to g.

We have also built in a maximum size of the transition system, which is the unspecified natural number $max\_transition\_system\_size$. If the transition system that is generated gets more transitions than this constant, further generation of the transition system is halted and $message_{15}$ is flagged.

func $c_{11} : \mathit{Termlist} \times \mathit{Spec} \rightarrow \mathit{Bool}$
$message_{14} : \mathit{Termlist} \times \mathit{Spec} \rightarrow \mathit{String}$
$message_{15} :\rightarrow \mathit{String}$
$max\_transition\_system\_size :\rightarrow Nat$
var $s : \mathit{Termlist}$
$spec : \mathit{Spec}$
rew $c_{11}(s, spec) = and(welltyped(s, emv, get_{sig}(get_{adt}(spec))),$
$eq(get_{sorts}(get_{vars}(get_{proc}(spec))), whatsorts(s, emv, get_{sig}(get_{adt}(spec)))))$

act $r_{s2}, s_{s2}, c_{s2} : \mathit{Status}$
$r_{s2}, s_{s2}, c_{s2} : \mathit{Term}$
$r_{s2}, s_{s2}, c_{s2} : \mathit{Openterm}$
$r_{s1}, s_{s1}, c_{s1} : \mathit{Command}$
$r_{s1}, s_{s1}, c_{s1} : \mathit{Datatype}$
$r_{s1}, s_{s1}, c_{s1} : \mathit{Term}$
$r_{s1}, s_{s1}, c_{s1} : \mathit{Openterm}$
$r_{c2}, c_{c2} : \mathit{Status}$
$r_{c2}, c_{c2} : \mathit{Steplist}$
$s_{c1} : \mathit{Command}$
$s_{c1} : \mathit{Termlist}$
$s_{c1} : \mathit{Spec}$

proc $InI_0 =$
$\sum_{spec:Spec} r_{u1}(spec) \cdot s_{c1}(spec) \cdot s_{s1}(get_{adt}(spec)) \cdot InI_1(spec)+$
$r_{u1}(restart) \cdot InI_0+$
$r_{u1}(terminate) \cdot s_{c1}(terminate) \cdot s_{s1}(terminate)$

$InI_1(spec{:}Spec) =$
$r_{c2}(ok) \cdot$
$(r_{s2}(ok) \cdot s_{u2}(ok) \cdot InI_2(spec)+$
$\sum_{s:String} r_{s2}(notok(s)) \cdot s_{u2}(notok(s)) \cdot s_{c1}(restart) \cdot InI_0)+$

$\sum_{s:String} r_{c2}(notok(s))\ .$
$\quad (r_{s2}(ok)\ .\ s_{s1}(restart)\ .\ s_{u2}(notok(s)) +$
$\quad\quad \sum_{ss:String} r_{s2}(notok(ss))\ .\ s_{u2}(notok(compose(s, ss))))\ .\ InI_0$

$InI_2(spec{:}Spec) =$
$\quad \sum_{s:Termlist} r_{u1}(s)\ .\ (InI_3(ins(s, emtll), emtll, emts, s, get_{adt}(spec))$
$\quad\quad \lhd c_{11}(s, spec) \rhd s_{u2}(notok(message_{14}(s, spec)))\ .\ InI_2) +$
$\quad r_{u1}(restart)\ .\ s_{c1}(restart)\ .\ s_{s1}(restart)\ .\ InI_0 +$
$\quad r_{u1}(terminate)\ .\ s_{c1}(terminate)\ .\ s_{s1}(terminate)$

$InI_3(g{:}Termlistlist, b{:}Termlistlist, tl{:}Translist, s{:}Termlist, d{:}Datatype) =$
$\quad s_{u2}(TS(d, s, tl))\ .\ s_{c1}(restart)\ .\ s_{s1}(restart)\ .\ InI_0$
$\quad \lhd empty(g) \rhd$
$\quad (s_{u2}(notok(message_{15}))\ .\ s_{c1}(restart)\ .\ s_{s1}(restart)\ .\ InI_0$
$\quad \lhd geq(len(tl), max\_transition\_system\_size) \rhd$
$\quad s_{c1}(head(g))\ .$
$\quad\quad (\sum_{ss:String} r_{c2}(notok(ss))\ .\ s_{u2}(notok(ss))\ .\ s_{c1}(restart)\ .\ s_{s1}(restart)\ .\ InI_0 +$
$\quad\quad\quad \sum_{stpl:Steplist} r_{c2}(stpl)\ .\ InI_4(tail(g), ins(head(g), b), stpl, tl, s, d)))$

$InI_4(g{:}Termlistlist, b{:}Termlistlist, stpl{:}Steplist, tl{:}Translist, s{:}Termlist, d{:}Datatype) =$
$\quad InI_3(g, b, tl, s, d)$
$\quad \lhd empty(stpl) \rhd$
$\quad (InI_4(g, b, tail(stpl), ins(TR(head(g), get_{act}(head(stpl)),$
$\quad\quad get_{actterms}(head(stpl)), terminated), tl), s, d)$
$\quad \lhd eq(get_{state}(head(stpl)), terminated) \rhd$
$\quad InI_5(g, b, head(g), r(get_{state}(head(stpl))), emt, stpl, tl, s, d))$

$InI_5(g{:}Termlistlist, b{:}Termlistlist, t{:}Termlist,$
$\quad\quad t_1{:}Termlist, t_2{:}Termlist, stpl{:}Steplist, tl{:}Translist, s{:}Termlist, d{:}Datatype) =$
$\quad InI_4(g, if(or(test(t_2, g), test(t_2, b)), b, ins(t_2, b)), tail(stpl),$
$\quad\quad ins(TR(t, get_{act}(head(stpl)), get_{actterms}(head(stpl)), i(t_2)), tl), s, d)$
$\quad \lhd empty(t_1) \rhd$
$\quad s_{s1}(toe(t_1))\ .$
$\quad\quad (\sum_{ss:String} r_{s2}(notok(ss))\ .\ s_{u2}(notok(ss))\ .\ s_{c1}(restart)\ .\ s_{s1}(restart)\ .\ InI_0 +$
$\quad\quad\quad \sum_{u:Term} r_{s2}(u)\ .\ InI_5(g, b, t, untoe(t_1), ins(u, t_2), stpl, tl, s, d))$

7 Realisation of the Toolset

7.1 Communication via the Toolbus

We let the tools communicate via the Toolbus [1], which has recently been developed as a standardised way to let a number of tools that operate in parallel communicate data with each other. The communication between tools is depicted in Figure 2. For each tool that is connected to the toolbus, there is a corresponding 'toolbus process' that handles all communication. E.g., if a tool wants an action to take place, it sends a request to the toolbus process, that tries to let the action take place. If the action successfully happens, the toolbus process reports by sending a message back to the tool.

We explain below in detail how tools communicate with their toolbus processes in our realisations. We also explain how we let the toolbus processes communicate with each other for our application.

First a tool receives a `rec-execute` from the toolbus. A tool responds by sending a `snd-connect` to the toolbus. Then, a tool can decide to either send or receive data. If the tool sends data it only

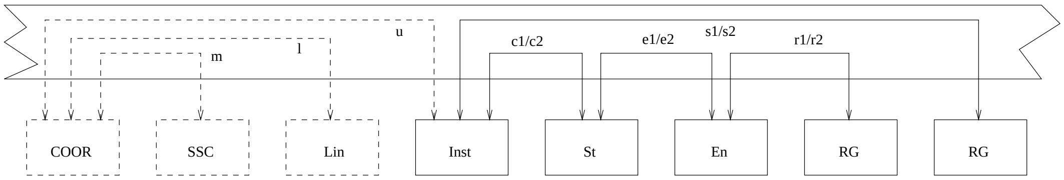

Figure 2: Communication via the toolbus

wants to take part in one single particular action, whereas if it wants to receive data, it can take part in a series of actions. This allows for an efficient implementation of synchronous actions using asynchronous message passing [9].

- If a tool wants to send data via channel a, with a term t, it sends a `snd-event`($send$,$a(t)$) to the toolbus process. It assumes that the toolbus process will take care that action $a(t)$ properly synchronises, and expects afterwards a `rec-ack-event`($send$) back.
- If a tool can receive data via channels $a_1 \ldots a_n$ it sends `snd-event`(receive$a_1 a_2 \ldots a_n$,""), i.e., the string receive followed by the names of the actions, which are assumed to consist of two characters each, in alphabetical order. If the toolbus process can establish a communication on one of these actions, say a_i, then the tool receives the data via a `rec-do`($a_i(t)$). Then, in order to acknowledge the event, the tool also receives a `rec-ack-event`(S,receive$a_1 a_2 \ldots a_n$). This looks inefficient, but is due to the restrictions enforced to tool/toolbus process communications.

If a tool decides to terminate, and henceforth disconnect from the toolbus the tool simply sends a `snd-disconnect` to the toolbus.

A typical example of a toolbus script for the Instantiator is given in Appendix C.1.

7.2 Realisation in C

Prototypes of all tools have been constructed in C. The code can be found in Appendix C. It appeared that, given the implementations and the tool/toolbus interaction scheme it is quite straightforward to make such an initial implementation. It is more involved to make an efficient implementation. In particular, an efficient implementation of all required function on datatypes is not always a straightforward job.

A General datatypes related to data

We list all the general datatypes that have been used in the specifications and implementations. The main purpose is to define the sorts *Term* and *Datatype* as the tools exchange mainly terms of these sorts. Elements of *Datatype* are the standard equationally specified datatypes, with as extra feature that each sort can contain both constructors and functions. The semantic interpretation is that if a sort contains constructors then each element of that sort can be written using constructors only. Each datatype is preceded by an informal description of the sort, in case that is not obvious, and outlining the functions that operate on that sort.

Almost every function is completely defined in the usual style using equations, with a few exceptions. In some cases the specified function is left undetermined for certain values in its domain. E.g. the predecessor of zero, $pred(0)$, is left undetermined, as are the tails of various empty lists.

For strings, we have not specified how these look like, but assume that there is a large domain of strings, on which an equality predicate eq has been defined, which we also have not specified.

There are a few functions on terms that we could not specify in μCRL. These are the equality function *equal* on terms modulo a datatype, *satlist* providing a list of closed terms satisfying a predicate, *steplist* generating the outgoing transitions of a particular state, and *transsys* generating a transition system.

All these are defined in a 'common mathematical style' in the section where they are used. Only the function *equal* is used outside its own section.

A.1 Bool

The sort *Bool*, its constants f (false) and t (true), and its operations are standard and require no explanation. It is important to realise that in the semantics of μCRL the sort *Bool* contains exactly two different elements represented by f and t.

$$
\begin{array}{ll}
\textbf{sort} & Bool \\
\textbf{func} & \mathsf{t}, \mathsf{f} :\rightarrow Bool \\
 & not : Bool \rightarrow Bool \\
 & and, or, implies : Bool \times Bool \rightarrow Bool \\
 & if : Bool \times Bool \times Bool \rightarrow Bool \\
 & \\
\textbf{var} & x, y : Bool \\
\textbf{rew} & not(\mathsf{t}) = \mathsf{f} \\
 & not(\mathsf{f}) = \mathsf{t} \\
 & and(\mathsf{f}, x) = \mathsf{f} \\
 & and(\mathsf{t}, x) = x \\
 & or(\mathsf{f}, x) = x \\
 & or(\mathsf{t}, x) = \mathsf{t} \\
 & implies(x, \mathsf{t}) = \mathsf{t} \\
 & implies(\mathsf{f}, x) = \mathsf{t} \\
 & implies(\mathsf{t}, \mathsf{f}) = \mathsf{f} \\
 & if(\mathsf{t}, x, y) = x \\
 & if(\mathsf{f}, x, y) = y
\end{array}
$$

A.2 Natural numbers

The natural numbers require no explanation. Note that we have not defined the effect of the predecessor function *pred* on zero.

$$
\begin{array}{ll}
\textbf{sort} & Nat \\
\textbf{func} & S : Nat \rightarrow Nat \\
 & 0 :\rightarrow Nat \\
 & eq : Nat \times Nat \rightarrow Bool \\
 & pred : Nat \rightarrow Nat \\
 & geq : Nat \times Nat \rightarrow Bool \\
\textbf{var} & x, y : Nat \\
\textbf{rew} & eq(0, 0) = \mathsf{t} \\
 & eq(S(x), 0) = \mathsf{f} \\
 & eq(0, S(x)) = \mathsf{f} \\
 & eq(S(x), S(y)) = eq(x, y) \\
 & pred(S(x)) = x \\
 & geq(x, 0) = \mathsf{t} \\
 & geq(0, S(x)) = \mathsf{f}
\end{array}
$$

$geq(S(x), S(y)) = geq(x, y)$

A.3 String

Elements of sort *String* are used to represent sort, function, action and variable names that appear in a datatype, as well as the contents of *notok* messages, that are communicated between tools. We leave the contents of the sort *String* undetermined, but it seems reasonable that this sort is taken to be as large as possible in realisations. We also do not define equality on strings, but it should be understood that eq on strings s_1 and s_2 should yield t exactly when the strings s_1 and s_2 are of the same length, and consist of the same sequences of symbols.

There are three standard strings, that are referred to by $true$, $false$ and *Bool*. According to the definition of μCRL these refer to the concrete strings '*T*', '*F*' and '*Bool*'. Note that the string *Bool* should not be confused with the sort *Bool*. The first one refers to the sort *Bool* in the datatypes that we are specifying, whereas the second refers to the sort *Bool* in the specification of datatypes.

The function *string* is defined to yield a unique string for every natural number. It is used in the definition of *fresh*, which is a function yielding fresh variables.

The function *compose* turns two strings into one.

sort $String$
func $\ldots :\rightarrow String$
$true, false, Bool :\rightarrow String$
$eq : String \times String \rightarrow Bool$
$if : Bool \times String \times String \rightarrow String$
$string : Nat \rightarrow String$
$compose : String \times String \rightarrow String$

var $x, y : String$
$n_1, n_2 : Nat$
rew $if(\mathsf{t}, x, y) = x$
$if(\mathsf{f}, x, y) = y$
$eq(n_1, n_2) = eq(string(n_1), string(n_2))$

A.4 Stringlist

The sort *Stringlist* contains lists of strings. The function ems represents the empty list of strings. The function $ins(x, s)$ returns the string list obtained by putting string x to the front of string list s. The predicate $test(x, s)$ checks whether string x occurs in string list s and the predicate $eq(s_1, s_2)$ determines whether the stringlists s_1 and s_2 are equal. Furthermore, the function $conc(s_1, s_2)$ yields the string list obtained by concatenating the string lists s_1 and s_2 and $subsetof(s_1, s_2)$ checks whether the strings in string list s_1 are all contained in string list s_2.

sort $Stringlist$
func $ems :\rightarrow Stringlist$
$ins : String \times Stringlist \rightarrow Stringlist$
$test : String \times Stringlist \rightarrow Bool$
$eq : Stringlist \times Stringlist \rightarrow Bool$
$subsetof : Stringlist \times Stringlist \rightarrow Bool$
$conc : Stringlist \times Stringlist \rightarrow Stringlist$
var $x, y : String$
$s, s_1, s_2 : Stringlist$
rew $test(x, ems) = \mathsf{f}$
$test(x, ins(y, s)) = or(eq(x, y), test(x, s))$

$eq(ems, ems) = \mathsf{t}$
$eq(ems, ins(x, s)) = \mathsf{f}$
$eq(ins(x, s), ems) = \mathsf{f}$
$eq(ins(x, s), ins(y, s_1)) = and(eq(x, y), eq(s, s_1))$
$subsetof(ems, s_2) = \mathsf{t}$
$subsetof(ins(x, s_1), s_2) = and(test(x, s_2), subsetof(s_1, s_2))$
$conc(ems, s) = s$
$conc(ins(x, s), s_1) = ins(x, conc(s, s_1))$

A.5 Status

Elements of sort *Status* are used to communicate the results of requested operations between tools. The element *ok* generally means that operation has been successfully executed. The element *notok* generally indicates that the operation was not successful. It has an argument of sort *String* that can be used to indicate the reason of failure. In all descriptions of both specifications and implementations of tools we have left the contents of this string undetermined. The functions $eq_{\dots}$ can be used to determine the nature of a *Status* message. Moreover, they imply that *ok* and *notok* messages can not be taken the same in any model of this datatype, as this would cause the constants t and f to become equal.

sort $Status$
func $ok : \to Status$
$notok : String \to Status$
$eq_{ok}, eq_{notok} : Status \to Bool$

var $s : String$
rew $eq_{ok}(ok) = \mathsf{t}$
$eq_{ok}(notok(s)) = \mathsf{f}$
$eq_{notok}(ok) = \mathsf{f}$
$eq_{notok}(notok(s)) = \mathsf{t}$

A.6 Command

The *Commands* *restart* en *terminate* are respectively used to instruct a tool to go back to its initial state, and to instruct a tool to terminate its execution.

sort $Command$
func $restart, terminate : \to Command$
$eq_{restart}, eq_{terminate} : Command \to Bool$

rew $eq_{restart}(restart) = \mathsf{t}$
$eq_{restart}(terminate) = \mathsf{f}$
$eq_{terminate}(restart) = \mathsf{f}$
$eq_{terminate}(terminate) = \mathsf{t}$

A.7 Function

The sort *Function* contains terms of the form $F(n, L, n_1)$, which represents the function declaration of a function with name n, argument sorts L, and target sort n_1. The function $sorts(f)$ returns a list of all the sorts that occur in the function declaration f.

sort $Function$

func $F : String \times Stringlist \times String \rightarrow Function$
$get_{name} : Function \rightarrow String$
$get_{args} : Function \rightarrow Stringlist$
$get_{targ} : Function \rightarrow String$
$sorts : Function \rightarrow Stringlist$
var $x, y : String$
$s : Stringlist$
rew $get_{name}(F(x, s, y)) = x$
$get_{args}(F(x, s, y)) = s$
$get_{targ}(F(x, s, y)) = y$
$sorts(F(x, s, y)) = ins(y, s)$

A.8 Functionlist

The sort *Functionlist* contains lists of functions. As such a functionlist represents all functions or constructors that belong to a certain datatype. $exists(n, L, n_1, F_1)$ checks whether there is a function $F(n, L, n_1)$ in function list F_1. $exists(n, L, F_1)$ checks whether, for some string n_1, there is a function $F(n, L, n_1)$ in function list F_1. $uniquetarget(F)$ checks whether each function in function list F, as determined by its name and list of argument sorts, does not occur more than once in F. $sorts(F)$ returns a list of all sorts that occur in the function declarations in function list F. $existsconstr(V, F)$ checks whether there is a variable n in variable list V for which there exists a function in function list F whose target sort is the same as the sort of n. $existsconstr(n, F)$ is similar.

sort $Functionlist$
func $emf : \rightarrow Functionlist$
$ins : Function \times Functionlist \rightarrow Functionlist$
$conc : Functionlist \times Functionlist \rightarrow Functionlist$
$exists : String \times Stringlist \times String \times Functionlist \rightarrow Bool$
$exists : String \times Stringlist \times Functionlist \rightarrow Bool$
$uniquetarget : Functionlist \rightarrow Bool$
$sorts : Functionlist \rightarrow Stringlist$
$existsconstr : Variablelist \times Functionlist \rightarrow Bool$
$existsconstr : String \times Functionlist \rightarrow Bool$

var $x, n, n_1, n_2, n_3 : String$
$L, L_1 : Stringlist$
$F_0, F_1 : Functionlist$
$f : Function$
$vl_1 : Variablelist$
rew $conc(emf, F_0) = F_0$
$conc(ins(f, F_0), F_1) = ins(f, conc(F_0, F_1))$
$exists(n, L, n_1, emf) = \mathsf{f}$
$exists(n, L, n_1, ins(F(n_2, L_1, n_3), F_0)) =$
$\quad or(and(eq(n, n_2), and(eq(L, L_1), eq(n_1, n_3))), exists(n, L, n_1, F_0))$
$exists(n, L, emf) = \mathsf{f}$
$exists(n, L, ins(F(n_2, L_1, n_3), F_0)) = or(and(eq(n, n_2), eq(L, L_1)), exists(n, L, F_0))$
$uniquetarget(emf) = \mathsf{t}$
$uniquetarget(ins(F(n, L, n_1), F_0)) = and(not(exists(n, L, F_0)), uniquetarget(F_0))$
$sorts(emf) = ems$
$sorts(ins(f, F_0)) = conc(sorts(f), sorts(F_0))$

$existsconstr(emv, F_0) = \mathsf{f}$
$existsconstr(ins(x, n, vl_1), F_0) = or(existsconstr(n, F_0), existsconstr(vl_1, F_0))$
$existsconstr(n, emf) = \mathsf{f}$
$existsconstr(n, ins(F(x, L, n_1), F_0)) = or(eq(n, n_1), existsconstr(n, F_0))$

A.9 Signature

The sort *Signature* contains signatures of datatypes, i.e., elements of the form $S(L, F_1, F_2)$ where L is a list of strings representing the sort names that occur in the datatype, F_1 is a list of functions representing the constructors in the datatype, and F_2 is a list of functions representing the function symbols that can occur in the datatype.

There are two predicates that verify whether signatures satisfy basic 'soundness' properties. The predicate $boolsexist(S)$ checks whether there are constant functions named $true$ and $false$, both with result type $Bool$, among the constructors of signature S, and $sortsexist(S)$ checks whether all the sorts that occur in the constructor and function declarations of signature S are contained in its list of sorts.

sort $Signature$
func $S : Stringlist \times Functionlist \times Functionlist \rightarrow Signature$
$get_{sorts} : Signature \rightarrow Stringlist$
$get_{cons}, get_{func} : Signature \rightarrow Functionlist$
$boolsexist, sortsexist : Signature \rightarrow Bool$

var $L_1 : Stringlist$
$F_1, F_2 : Functionlist$
$S : Signature$
rew $get_{sorts}(S(L_1, F_1, F_2)) = L_1$
$get_{cons}(S(L_1, F_1, F_2)) = F_1$
$get_{func}(S(L_1, F_1, F_2)) = F_2$
$boolsexist(S(L_1, F_1, F_2)) = and(exists(true, ems, Bool, F_1), exists(false, ems, Bool, F_1))$
$sortsexist(S) = subsetof(sorts(conc(get_{cons}(S), get_{func}(S))), get_{sorts}(S))$

A.10 Variablelist

The sort *Variablelist* contains lists with variables. Each variable is represented as a pair of strings of which the first one is the name of the variable, and the second one the sort of the variable. Variablelists are generally used to declare the variables with their sorts in open terms, equations, process expressions, etc. Variablelists are primary constructed using the constant emv, the empty variable list, and the function $ins(n, n_1, V)$ which prefixes a variable with name n and sort n_1 to the front of variablelist V.

The following functions have been defined on *Variablelists*. The function $test_1(n, V)$ checks whether *String* n occurs as the name of a variable in V and $test_2(n, V)$ tests whether string n occurs among the *sorts* in variable list V.

For a non-empty variable list V, $get_{sort}(n, V)$ returns the sort of variable n in V. For empty V, this function is unspecified. The function $get_{sorts}(V)$ yields a list containing the sorts of V.

The function $targetsort(n, L, V, S)$ yields the sort of the function or variable n applied to arguments that have sorts prescribed by L with respect to a list of variables defined by V and a signature provided by S. For instance if L is an empty stringlist and n occurs as a variable in V, its sort is provided. Note that if n occurs more than once with arguments of sort L in V or S only the sort of the first matching occurrence is provided. If n does not occur, with matching arguments, the result of the function targetsort is not specified.

The function $noclash(V, S)$ checks whether the variables from variable list V do not occur as constant functions or constructors in signature S. Note that the sorts of variables, occurring in V, are ignored for this purpose.

The function $sortsexist(V, F)$ checks whether all the sorts of variables, occurring in variable list V, are included in the string list F and the function $nooverlap(V, V_1)$ checks whether there are no variables with the same name in V and V_1.

The function $fresh(L, V, S)$ yields a variable list, of which the variables do not clash with S or overlap with V and that have sorts as prescribed in L. $fresh(V, S, n)$ is auxiliary.

The function $conc(V, V_1)$ yields the concatenation of variable lists V and V_1 and $eq(V_1, V_2)$ checks whether V_1 and V_2 are equal.

sort $Variablelist$

func $emv :\rightarrow Variablelist$
$ins : String \times String \times Variablelist \rightarrow Variablelist$
$test_1 : String \times Variablelist \rightarrow Bool$
$test_2 : String \times Variablelist \rightarrow Bool$
$get_{sort} : String \times Variablelist \rightarrow String$
$get_{sorts} : Variablelist \rightarrow Stringlist$
$targetsort : String \times Stringlist \times Variablelist \times Signature \rightarrow String$
$targetsort : String \times Stringlist \times Variablelist \times Functionlist \rightarrow String$
$noclash : Variablelist \times Signature \rightarrow Bool$
$sortsexist : Variablelist \times Stringlist \rightarrow Bool$
$nooverlap : Variablelist \times Variablelist \rightarrow Bool$
$fresh : Stringlist \times Variablelist \times Signature \rightarrow Variablelist$
$fresh : Variablelist \times Signature \times Nat \rightarrow String$
$conc : Variablelist \times Variablelist \rightarrow Variablelist$
$eq : Variablelist \times Variablelist \rightarrow Bool$

var $n, n_1, n_2, n_3 : String$
$V, V_1 : Variablelist$
$S : Signature$
$L_1, L_2 : Stringlist$
$F_1, F_2 : Functionlist$
$m : Nat$

rew $test_1(n, emv) = \mathsf{f}$
$test_1(n, ins(n_1, n_2, V)) = or(eq(n, n_1), test_1(n, V))$
$test_2(n, emv) = \mathsf{f}$
$test_2(n, ins(n_1, n_2, V)) = or(eq(n, n_2), test_1(n, V))$
$get_{sort}(n, ins(n_1, n_2, V)) = if(eq(n, n_1), n_2, get_{sort}(n, V))$
$get_{sorts}(emv) = ems$
$get_{sorts}(ins(n, n_1, V)) = ins(n_1, get_{sorts}(V))$
$targetsort(n, L_1, emv, S(L_2, F_1, F_2)) = targetsort(n, L_1, emv, conc(F_1, F_2))$
$targetsort(n, L_1, ins(n_1, n_2, V), S) =$
$\quad if(and(eq(n, n_1), eq(L_1, ems)), n_2, targetsort(n, L_1, V, S))$
$targetsort(n, L_1, V, ins(F(n_1, L_2, n_2), F_1)) =$
$\quad if(and(eq(n, n_1), eq(L_1, L_2)), n_2, targetsort(n, L_1, V, F_1))$
$noclash(emv, S) = \mathsf{t}$
$noclash(ins(n, n_1, V), S) =$
$\quad and(not(exists(n, ems, conc(get_{func}(S), get_{cons}(S)))), noclash(V, S))$
$sortsexist(emv, L_1) = \mathsf{t}$
$sortsexist(ins(n, n_1, V), L_1) = and(test(n_1, L_1), sortsexist(V, L_1))$

$nooverlap(emv, V) = \mathsf{t}$
$nooverlap(ins(n, n_1, V), V_1) = and(not(test_1(n, V)), nooverlap(V, V_1))$
$fresh(ems, V, S) = emv$
$fresh(ins(n, L_1), V, S) = ins(fresh(V, S, 0), n, fresh(L_1, ins(fresh(V, S, 0), n, V), S))$
$fresh(V, S, m) = if(or(exists(string(m), ems, conc(get_{cons}(S), get_{func}(S))),$
$\quad test_1(string(m), V)), fresh(V, S, S(m)), string(m))$
$conc(emv, V) = V$
$conc(ins(n_1, n_2, V), V_1) = ins(n_1, n_2, conc(V, V_1))$
$eq(emv, emv) = \mathsf{t}$
$eq(emv, ins(n, n_1, V)) = \mathsf{f}$
$eq(ins(n, n_1, V), emv) = \mathsf{f}$
$eq(ins(n, n_1, V), ins(n_2, n_3, V_1)) = and(eq(n, n_2), and(eq(n_1, n_3), eq(V, V_1)))$

A.11 Term and Termlist

We define terms as the application of a function symbol, represented by a *String*, to a number of arguments, represented by a *Termlist*. We use the function T for this. For instance the constant '0' is denoted as $T(0, emt)$ where 0 is a *String* and emt is the empty termlist. The term $S(0)$, i.e., the successor applied to 0, is denoted by $T(S, ins(T(0, emt), emt))$, i.e., the application of the *String* S to the *Termlist* of length 1 containing the *Term* representing 0. Note that the sorts *Term* and *Termlist* are defined in terms of each other.

The definition of terms may look somewhat far-fetched on first sight. However, the current definition is one of the more natural ways to define terms as an abstract datatype.

Using the explanation above it should now be easy to understand the standard functions on terms that we have defined. The function $T(n, L)$ combines the (function) name n with arguments provided by the *Termlist* L to yield a new *Term*. The function emt represents the empty term list and $ins(t, L)$ returns the term list obtained by inserting the term t into term list L.

The function $empty(L)$ checks whether the termlist L is empty, $toe(L)$ returns the last term of term list L and $untoe(L)$ returns the term list L without its last term.

For a term t, $welltyped(t, V, S)$ checks whether: (1) the functor of t with the sorts of its arguments occurs among the constructors or functions of signature S, or (2) the functor of t occurs among the variables in variable list V and the list of arguments of t is empty. In addition, it recursively checks the arguments of t. Similarly, $welltyped(T, V, S)$ performs this check for all the terms in term list T. The function $closed(L, S)$ checks whether *Termlist* L is well typed with respect to S. Provided L is well typed wrt. some variablelist, this exactly yields true if the terms in L do not contain variables.

The function $whatsort(t, V, S)$ returns the sort of term t as determined by variable list V and signature S (where V is searched before S, as in the case of the function $targetsort$ above). Similarly, $whatsorts(L, V, S)$ returns the list of sorts of the terms in the term list L.

The function $subst(L, V, t)$ returns the term that is obtained by simultaneously substituting, in t, those variables that occur in variable list V by the corresponding terms in term list L (so L and V should have the same length). Similarly, $subst(L, V, L_1)$ iteratively substitutes terms in L for the respective variables in V in the termlist L_1. $subst(t, x, t_1)$ yields t_1 where t has been substituted for x, and similarly, $subst(t, x, L)$ yields the *Termlist* L where t has been substituted for x in all terms of L.

If string n occurs in variable list V, $repl(L, V, n)$ returns the term occurring in the corresponding position in term list L. Otherwise, it returns the constant $T(n, emt)$.

The function $mk_{terms}(V)$ transforms the *Variablelist* V into a *Termlist* where each variable is transformed into its corresponding term.

$constructorterm(t, S)$ is true if there are no constructors (in signature S) for the sort of term t, or if the top-level function symbol of t is a constructor in S and its arguments are, recursively, constructorterms again. $constructorterm(T, S)$ has a similar meaning for a list T of terms.

sort *Term Termlist*

func
$T : String \times Termlist \to Term$
$emt :\to Termlist$
$ins : Term \times Termlist \to Termlist$
$empty : Termlist \to Bool$
$toe : Termlist \to Term$
$untoe : Termlist \to Termlist$
$eq : Term \times Term \to Bool$
$eq : Termlist \times Termlist \to Bool$
$welltyped : Term \times Variablelist \times Signature \to Bool$
$welltyped : Termlist \times Variablelist \times Signature \to Bool$
$closed : Term \times Signature \to Bool$
$closed : Termlist \times Signature \to Bool$
$whatsort : Term \times Variablelist \times Signature \to String$
$whatsorts : Termlist \times Variablelist \times Signature \to Stringlist$
$subst : Termlist \times Variablelist \times Term \to Term$
$subst : Termlist \times Variablelist \times Termlist \to Termlist$
$subst : Term \times String \times Term \to Term$
$subst : Term \times String \times Termlist \to Termlist$
$repl : Termlist \times Variablelist \times String \to Term$
$mk_{terms} : Variablelist \to Termlist$
$if : Bool \times Term \times Term \to Term$
$if : Bool \times Termlist \times Termlist \to Termlist$
$constructorterm : Term \times Signature \to Bool$
$constructorterm : Termlist \times Signature \to Bool$

var
$n, n_1, n_2, s : String$
$t, t_1 : Term$
$L, L_1, M : Termlist$
$S : Signature$
$V, V_1 : Variablelist$

rew
$empty(emt) = \mathsf{t}$
$empty(ins(t, L)) = \mathsf{f}$
$toe(ins(t, L)) = if(eq(L, emt), t, toe(L))$
$untoe(ins(t, L)) = if(eq(L, emt), emt, ins(t, untoe(L)))$
$eq(T(n, L), T(n_1, L_1)) = and(eq(n, n_1), eq(L, L_1))$
$eq(emt, emt) = \mathsf{t}$
$eq(ins(t, L), emt) = \mathsf{f}$
$eq(emt, ins(t, L)) = \mathsf{f}$
$eq(ins(t, L), ins(t_1, L_1)) = and(eq(t, t_1), eq(L, L_1))$
$welltyped(T(n, L), V, S) =$
$\quad and(or(exists(n, whatsorts(L, V, S), conc(get_{cons}(S), get_{func}(S))),$
$\quad and(test_1(n, V), eq(L, emt))), welltyped(L, V, S))$
$welltyped(emt, V, S) = \mathsf{t}$
$welltyped(ins(t, L), V, S) = and(welltyped(t, V, S), welltyped(L, V, S))$
$closed(t, S) = welltyped(t, emv, S)$
$closed(L, S) = welltyped(L, emv, S)$
$whatsort(T(n, L), V, S) = targetsort(n, whatsorts(L, V, S), V, S)$
$whatsorts(emt, V, S) = ems$
$whatsorts(ins(t, L), V, S) = ins(whatsort(t, V, S), whatsorts(L, V, S))$
$subst(L, V, T(n, emt)) = repl(L, V, n)$

$subst(L, V, T(n, ins(t_1, L_1))) = T(n, subst(L, V, ins(t_1, L_1)))$
$subst(L, V, emt) = emt$
$subst(L, V, ins(t_1, L_1)) = ins(subst(L, V, t_1), subst(L, V, L_1))$
$subst(t, n, T(n_1, emt)) = if(eq(n, n_1), t, T(n_1, emt))$
$subst(t, n, T(n_1, ins(t_1, L_1))) = T(n_1, subst(t, n, ins(t_1, L_1)))$
$subst(t, n, emt) = emt$
$subst(t, n, ins(t_1, L_1)) = ins(subst(t, n, t_1), subst(t, n, L_1))$
$repl(emt, emv, n) = T(n, emt)$
$repl(ins(t, L), ins(n_1, n_2, V), n) = if(eq(n, n_1), t, repl(L, V, n))$
$mk_{terms}(emv) = emt$
$mk_{terms}(ins(n, s, V)) = ins(T(n, emt), mk_{terms}(V))$
$if(\mathsf{t}, t, t_1) = t$
$if(\mathsf{f}, t, t_1) = t_1$
$if(\mathsf{t}, L, L_1) = L$
$if(\mathsf{f}, L, L_1) = L_1$
$constructorterm(T(n, L), S) =$
$\quad if(existsconstr(whatsort(T(n, L), emv, S), get_{cons}(S)),$
$\quad\quad and(exists(n, whatsorts(L, emv, S), get_{cons}(S)), constructorterm(L, S)), \mathsf{t})$
$constructorterm(emt, S) = \mathsf{t}$
$constructorterm(ins(t, L), S) = and(constructorterm(t, S), constructorterm(L, S))$

A.12 Termlistlist

The sort *Termlistlist* contains lists of lists of terms. It is very straightforwardly defined. The function *emtll* is the empty termlistlist and $ins(L, LL)$ returns the list of term lists obtained by inserting the term list L into list of term lists LL.

The predicate $empty(LL)$ checks whether the termlistlist LL is empty, $head(LL)$ returns the termlist which is the "head" of termlistlist LL, and $tail(LL)$ returns the termlistlist which is its "tail". The function $test$ checks whether a particular termlist occurs in a termlistlist and the function rem removes a given termlist from a termlistlist.

sort *Termlistlist*
func $emtll : \to Termlistlist$
$ins : Termlist \times Termlistlist \to Termlistlist$
$empty : Termlistlist \to Bool$
$head : Termlistlist \to Termlist$
$tail : Termlistlist \to Termlistlist$
$test : Termlist \times Termlistlist \to Bool$
$rem : Termlist \times Termlistlist \to Termlistlist$
$if : Bool \times Termlistlist \times Termlistlist \to Termlistlist$

var $L, L_1 : Termlist$
$LL, LL_1 : Termlistlist$
rew $empty(emtll) = \mathsf{t}$
$empty(ins(L, LL)) = \mathsf{f}$
$head(ins(L, LL)) = L$
$tail(ins(L, LL)) = LL$
$test(L, emtll) = \mathsf{f}$
$test(L, ins(L_1, LL)) = if(eq(L, L_1), \mathsf{t}, test(L, LL))$
$rem(L, emtll) = emtll$
$rem(L, ins(L_1, LL)) = if(eq(L, L_1), rem(L, LL), ins(L_1, rem(L, LL)))$

$if(\mathsf{t}, LL, LL_1) = LL$
$if(\mathsf{f}, LL, LL_1) = LL_1$

A.13 itlist

The sort *itlist* is an auxiliary datatype that is solely used in the implementation of the Enumerator. The purpose of *itlist* is explained in Section 4.3. The functions that are defined here are all elementary list handling functions.

sort $itlist$
func $emit :\rightarrow itlist$
$ins : Variablelist \times Termlist \times Term \times itlist \rightarrow itlist$
$hd_{vars} : itlist \rightarrow Variablelist$
$hd_{insts} : itlist \rightarrow Termlist$
$hd_{term} : itlist \rightarrow Term$
$tail : itlist \rightarrow itlist$
$empty : itlist \rightarrow Bool$
$if : Bool \times itlist \times itlist \rightarrow itlist$

var $vl_1 : Variablelist$
$tl : Termlist$
$t : Term$
$itl, itl_1 : itlist$
rew $hd_{vars}(ins(vl_1, tl, t, itl)) = vl_1$
$hd_{insts}(ins(vl_1, tl, t, itl)) = tl$
$hd_{term}(ins(vl_1, tl, t, itl)) = t$
$tail(ins(vl_1, tl, t, itl)) = itl$
$empty(emit) = \mathsf{t}$
$empty(ins(vl_1, tl, t, itl)) = \mathsf{f}$
$if(\mathsf{t}, itl, itl_1) = itl$
$if(\mathsf{f}, itl, itl_1) = itl_1$

A.14 Openterm

An open term is a pair of a term and a variablelist, indicating which strings in the term are to be considered as variables. The function $correctopenterm(t, S)$ checks (1) the "noclash" condition, in signature S, of those variables that occur in the variable list of the open term t, (2) the "sortsexist" condition for the variable list of t, and (3) the "welltyped" condition of the term of t with respect to its variable list and S.

The function $whatsort$ yields the sort of an open term.

sort $Openterm$
func $O : Term \times Variablelist \rightarrow Openterm$
$get_{term} : Openterm \rightarrow Term$
$get_{vars} : Openterm \rightarrow Variablelist$
$correctopenterm : Openterm \times Signature \rightarrow Bool$
$whatsort : Openterm \times Signature \rightarrow String$

var $t : Term$
$V : Variablelist$
$S : Signature$
rew $get_{term}(O(t, V)) = t$

$get_{vars}(O(t,V)) = V$
$correctopenterm(O(t,V),S) =$
$\quad and(noclash(V,S), and(sortsexist(V, get_{sorts}(S)), welltyped(t,V,S)))$
$whatsort(O(t,V),S) = whatsort(t,V,S)$

A.15 Equation

The expression $E(V,t_1,t_2)$ represents the equation with term t_1 in its left hand side, t_2 in its right hand, and V a variable list for the variables occurring in these terms.

sort *Equation*
func $E : Variablelist \times Term \times Term \rightarrow Equation$
$get_{vars} : Equation \rightarrow Variablelist$
$get_{left} : Equation \rightarrow Term$
$get_{right} : Equation \rightarrow Term$

var $v : Variablelist$
$t_1, t_2 : Term$
rew $get_{vars}(E(v,t_1,t_2)) = v$
$get_{left}(E(v,t_1,t_2)) = t_1$
$get_{right}(E(v,t_1,t_2)) = t_2$

A.16 Equationlist

Equations are grouped together in elements of sort *Equationlist*. The major function on equationlists is $correcteqs(S,L)$ that checks whether for each equation $E(V,t_1,t_2)$ in equation list L, the sorts in variable list V are declared in signature S, the variables in V do not clash with the signature S and all terms are well typed wrt. S.

sort *Equationlist*
func $eme : \rightarrow Equationlist$
$ins : Equation \times Equationlist \rightarrow Equationlist$
$correcteqs : Signature \times Equationlist \rightarrow Bool$

var $S : Signature$
$V : Variablelist$
$t_1, t_2 : Term$
$L : Equationlist$
rew $correcteqs(S, eme) = \mathsf{t}$
$correcteqs(S, ins(E(V,t_1,t_2),L)) =$
$\quad and(and(and(sortsexist(V, get_{sorts}(S)), noclash(V,S)),$
$\quad and(welltyped(t_1,V,S), welltyped(t_2,V,S))), correcteqs(S,L))$

A.17 Datatype

A pair of a signature and a list of equations forms a datatype. The function $ssc(d)$ checks whether all constructors and functions in the signature of datatype d have a unique target sort. Moreover, it verifies whether all sorts are properly declared, whether all equations are well-typed and whether the constructors $true$ and $false$ exists.

The function *equal* yields true exactly if its two first arguments are equal with respect to all models of the data type, which is the third argument. This function on both open and closed terms are defined in the next section, as it is inconvenient to define them in the datatype. The function *equal* on term lists can be defined straightforwardly on the basis of the definition of *equal* on terms. Note the difference between the functions *eq* and *equal*. The first one always refers to equality with respect to the equations of the datatype in this text, whereas the second refers to equality with respect to the equations in the datatype provided as third argument.

sort $Datatype$
func $D : Signature \times Equationlist \to Datatype$
$get_{sig} : Datatype \to Signature$
$get_{eqs} : Datatype \to Equationlist$
$ssc : Datatype \to Bool$
$equal : Openterm \times Openterm \times Datatype \to Bool$
$equal : Term \times Term \times Datatype \to Bool$
$equal : Termlist \times Termlist \times Datatype \to Bool$
var $S : Signature$
$EL : Equationlist$
$t_1, t_2 : Term$
$L, L_1 : Termlist$
$d : Datatype$
rew $get_{sig}(D(S, EL)) = S$
$get_{eqs}(D(S, EL)) = EL$
$ssc(D(S, EL)) =$
$and(uniquetarget(conc(get_{func}(S), get_{cons}(S))), and(sortsexist(S),$
$and(correcteqs(S, EL), boolsexist(S))))$
$equal(emt, emt, d) = \mathsf{t}$
$equal(ins(t_1, L), emt, d) = \mathsf{f}$
$equal(emt, ins(t_1, L), d) = \mathsf{f}$
$equal(ins(t_1, L), ins(t_2, L_1), d) = and(equal(t_1, t_2, d), equal(L, L_1, d))$

A.18 Interpretation of datatypes

In this section we define properties of the function *equal* on open and closed terms. Basically, these are that terms are equal if equality can be proven using axioms and the inference rules for equality. Terms of a sort that contain a constructor must be equal to a so called constructorterm. Moreover, the predicate equal may not relate *true* and *false*, or terms of a different sort. So, we find that

- $equal(T(true, emt), T(false, emt), d) = \mathsf{f}$ for every *Datatype* d.
- If $equal(t_1, t_2, d) = \mathsf{t}$ then $closed(t_1, get_{sig}(d))$ and $closed(t_2, get_{sig}(d))$.
- If t_1 and t_2 are terms such that $welltyped(t_1, emv, get_{sig}(d))$, $welltyped(t_2, emv, get_{sig}(d))$, $eq(whatsort(t_1, emv, get_{sig}(d)), whatsort(t_1, emv, get_{sig}(d))) = \mathsf{f}$, then $equal(t_1, t_2, d) = \mathsf{t}$.
- $equal(t, t, d) = \mathsf{t}$ for all *Term*'s t and *Datatype*'s d.
- If $equal(t_1, t_2, d) = \mathsf{t}$, then $equal(t_2, t_1, d) = \mathsf{t}$ for all *Term*'s t_1, t_2 and every *Datatype* d.
- If $equal(t_1, t_2, d) = \mathsf{t}$ and $equal(t_2, t_3, d) = \mathsf{t}$, then $equal(t_1, t_3, d) = \mathsf{t}$ for all *Term*'s t_1, t_2, t_3 and every *Datatype* d.
- If $equal(t_1, t_2, d) = \mathsf{t}$ then for every *Term* t and variable (*String*) x such that $welltyped(t, ins(x, srt, emv), get_{sig}(d)) = \mathsf{t}$ (where $srt = whatsort(t_1, emv, get_{sig}(d))$), then $equal(subst(t_1, x, t), subst(t_2, x, t), d)$.

- If $E(V, t_1, t_2)$ is an equation in $get_{eqs}(d)$, then for every *Termlist* tl such that $closed(tl, get_{sig}(d))$ and $whatsort(tl, emv, get_{sig}(d)) = get_{sorts}(V)$, we have

$$equal(subst(tl, V, t_1), subst(tl, V, t_2), d) = \mathsf{t}$$

- If t is a *Term* such that $closed(t, get_{sig}(d))$ and $existsconstr(whatsort(t, emv, get_{sig}(d)), get_{cons}(get_{sig}(d)))$, then there is a *Term* t' such that $constructorterm(t', get_{sig}(d)) = \mathsf{t}$ and $equal(t, t', d) = \mathsf{t}$.

Open terms are *equal* iff they are *equal* under all instantiations which make them closed terms:

- $equal(t_1, t_2, d)$ iff for all termlists tl_1, tl_2 (of appropriate sorts) such that $closed(ins(subst(tl_1, get_{vars}(t_1), t_1), emt), get_{sig}(d)) = \mathsf{t}$ and $closed(ins(subst(tl_2, get_{vars}(t_2), t_2), emt), get_{sig}(d)) = \mathsf{t}$, we have $equal(subst(tl_1, get_{vars}(t_1), t_1), subst(tl_2, get_{vars}(t_2), t_2), d)$.

B Datatypes describing processes

In this section we specify how terms denoting processes look like. Actually, the definitions can be divided in four sections. The sort *NTermlist* is a general sort denoting states of processes. The sorts *Summand*, *Sumlist*, *ProcSpec* and *Spec* together describe the input of the instantiator. Actually a term of sort *ProcSpec* denotes a linear process as described in [2].

The sorts *Transition*, *Translist* and *Transsys* together denote transition systems, which is the output of the instantiator. The two remaining sorts *Step* and *Steplist* are lists of steps, i.e., a pair of an action and a state, that are provided in response to a request to the Stepper to provide all outgoing transitions of a state.

All the sorts in this section are specified in a very straightforward way. Each sort is again accompanied with a short explanation, to indicate the purpose of such a sort.

B.1 NTermlist

Terms of the sort *NTermlist* denote states of processes. Basically, a state is just a *Termlist* indicating the values of the respective variables of the process. There is an embedding function i that translates terms of sort *Termlist* into terms of sort *NTermlist*. However, a process can also be in the terminated state. We have introduced a special element *terminated* in the sort *NTermlist* to denote this state.

We define two obvious functions on *NTermlist*, being *eq*, for equality, and *subst* that substitutes terms for variables in an *NTermlist*. Moreover, we define $matchsort(vl, ntl, vl_1, sig)$ that checks whether the sorts of the terms in ntl do match the sorts of the variables of vl, and $welltyped(ntl, vl, sig)$ checking whether the termlist ntl is well typed wrt. the signature sig.

sort $NTermlist$
func $i : Termlist \rightarrow NTermlist$
$r : NTermlist \rightarrow Termlist$
$terminated :\rightarrow NTermlist$
$eq : NTermlist \times NTermlist \rightarrow Bool$
$subst : Termlist \times Variablelist \times NTermlist \rightarrow NTermlist$
$matchsort : Variablelist \times NTermlist \times Variablelist \times Signature \rightarrow Bool$
$welltyped : NTermlist \times Variablelist \times Signature \rightarrow Bool$

var $tl, tl_1 : Termlist$
$vl, vl_1 : Variablelist$
$sig : Signature$

rew $r(i(tl)) = tl$
$eq(terminated, i(tl)) = \mathsf{f}$
$eq(terminated, terminated) = \mathsf{t}$
$eq(i(tl), terminated) = \mathsf{f}$
$eq(i(tl), i(tl_1)) = eq(tl, tl_1)$
$subst(tl, vl, terminated) = terminated$
$subst(tl, vl, i(tl_1)) = i(subst(tl, vl, tl_1))$
$matchsort(vl, terminated, vl_1, sig) = \mathsf{t}$
$matchsort(vl, i(tl), vl_1, sig) = eq(get_{sorts}(vl), whatsorts(tl, conc(vl, vl_1), sig))$
$welltyped(terminated, vl, sig) = \mathsf{t}$
$welltyped(i(tl), vl, sig) = welltyped(tl, vl, sig)$

B.2 Summand

We now start defining a linear process. The general definition of a linear process equation, based on the linear process of [2] is the following:

$$\begin{aligned} X(\vec{d}{:}\vec{D}) = \; & \textstyle\sum_{i\in I}\sum_{\vec{e_i}:\vec{E_i}} a_i(f_i(\vec{d},\vec{e_i})) \,.\, X(g_i(\vec{d},\vec{e_i})) \triangleleft c_i(\vec{d},\vec{e_i}) \triangleright \delta + \\ & \textstyle\sum_{j\in J}\sum_{\vec{e'}_j:\vec{E'}_j} a'_j(f'_j(\vec{d},\vec{e'}_j)) \triangleleft c'_j(\vec{d},\vec{e'}_j) \triangleright \delta \end{aligned}$$

It says that a process that starts in a state represented by the variables in $\vec{d}$ can perform actions a_i and a'_j, with appropriate parameters, resulting in some new state, represented by adapting the vector $\vec{d}$.

More precisely, for each $i \in I$, and each vector $\vec{e_i}{:}\vec{E}$, an action $a_i(f_i(\vec{d},\vec{e_i}))$ can be performed, provided condition $c_i(\vec{d},\vec{e_i})$ holds, and the process results in state $X(g_i(\vec{d},\vec{e_i}))$. The functions, f_i, g_i and c_i can be arbitrarily chosen.

Moreover, for each $j \in J$ and each vector $\vec{e}'_i{:}\vec{E}'$ an action $a'_j(f'_j(\vec{d},\vec{e}'_j))$ can be performed, provided condition $c'_i(\vec{d},\vec{e}'_i)$ holds, after which the process terminates instantly.

The datastructures below capture this notion of a linear process. First we define what a summand is, then we define what lists of summands are, and finally, we give a datastructure for complete linear processes. (A summand represents one of the alternatives of a linear process.)

A summand is a six-tuple $SMD(vl, a, tl, ntl, t)$ where vl is a *Variablelist* corresponding with $\vec{e_i}{:}\vec{E_i}$ above. The *String* a is the name of the action a_i, the *Termlist* tl represents the arguments of the action a, and as such represents the function f_i. The variables $\vec{d}$ and $\vec{e_i}$ may appear free in tl. The *Term* t represents the condition c_i. Note that t should have sort *Bool*. Finally, the *NTermlist* ntl denotes the nextstate, or if $ntl = terminated$, it denotes that the process is terminated. All functions that have been defined on the sort *Summand* are straightforward.

sort $Summand$
func $SMD : Variablelist \times String \times Termlist \times NTermlist \times Term \to Summand$
$get_{vars} : Summand \to Variablelist$
$get_{actname} : Summand \to String$
$get_{actterms} : Summand \to Termlist$
$get_{state} : Summand \to NTermlist$
$get_{cond} : Summand \to Term$
$eq : Summand \times Summand \to Bool$

var $v, v_1 : Variablelist$
$an, an_1 : String$
$at, at_1 : Termlist$
$st, st_1 : NTermlist$

$c, c_1 : Term$

rew $get_{vars}(SMD(v, an, at, st, c)) = v$
$get_{actname}(SMD(v, an, at, st, c)) = an$
$get_{actterms}(SMD(v, an, at, st, c)) = at$
$get_{state}(SMD(v, an, at, st, c)) = st$
$get_{cond}(SMD(v, an, at, st, c)) = c$
$eq(SMD(v, an, at, st, c), SMD(v_1, an_1, at_1, st_1, c_1)) =$
$\quad and(eq(v, v_1), and(eq(an, an_1), and(eq(at, at_1), and(eq(st, st_1), eq(c, c_1)))))$

B.3 Sumlist

The summands of the previous section are in a standard way grouped together in the form of a list. The only non standard function is $ssc(sl, vl, sig)$. It checks whether all summands in sl are statically semantically correct. This means that for each summand $SMD(vl_1, s, tl, ntl, t)$ the sorts of the terms ntl must match those of vl, as vl represents here the parameters $\vec{d}{:}\vec{D}$ of the process. It checks that the condition t is well typed and of sort $Bool$. It checks that the argument list tl of action s and the new state ntl are well typed. Finally, it checks whether the variablelists vl and vl_1 do not overlap with constants in the signature sig and with each other.

sort $Sumlist$

func $eml : \rightarrow Sumlist$
$ins : Summand \times Sumlist \rightarrow Sumlist$
$eq : Sumlist \times Sumlist \rightarrow Bool$
$empty : Sumlist \rightarrow Bool$
$head : Sumlist \rightarrow Summand$
$tail : Sumlist \rightarrow Sumlist$
$ssc : Sumlist \times Variablelist \times Signature \rightarrow Bool$

var $vl, vl_1 : Variablelist$
$sl, sl_1 : Sumlist$
$sm, sm_1 : Summand$
$adt : Datatype$
$s : String$
$tl : Termlist$
$ntl : NTermlist$
$t : Term$
$sig : Signature$
$eql : Equationlist$

rew $eq(eml, eml) = \mathsf{t}$
$eq(ins(sm, sl), eml) = \mathsf{f}$
$eq(sl, ins(sm, sl)) = \mathsf{f}$
$eq(ins(sm, sl), ins(sm_1, sl_1)) = and(eq(sm, sm_1), eq(sl, sl_1))$
$empty(eml) = \mathsf{t}$
$empty(ins(sm, sl)) = \mathsf{f}$
$head(ins(sm, sl)) = sm$
$tail(ins(sm, sl)) = sl$
$ssc(eml, vl, sig) = \mathsf{t}$
$ssc(ins(SMD(vl_1, s, tl, ntl, t), sl), vl, sig) =$
$\quad and(matchsort(vl, ntl, vl_1, sig),$
$\quad and(eq(whatsort(t, conc(vl, vl_1), sig), Bool),$
$\quad and(welltyped(t, conc(vl, vl_1), sig),$

$and(welltyped(tl, conc(vl, vl_1), sig),$
$and(noclash(vl, sig),$
$and(noclash(vl_1, sig),$
$and(nooverlap(vl, vl_1),$
$and(welltyped(ntl, conc(vl, vl_1), sig), ssc(sl, vl, sig)))))))))$

B.4 ProcSpec

A process specification consists of a pair $PROC(vl, sl)$ where the variablelist vl represents the parameters $\vec{d}{:}\vec{D}$ of a process, and the sumlist sl represents its summands. The functions defined on this sort are all fully standard.

sort $ProcSpec$
func $PROC : Variablelist \times Sumlist \to ProcSpec$
$get_{vars} : ProcSpec \to Variablelist$
$get_{sums} : ProcSpec \to Sumlist$

var $v : Variablelist$
$s : Sumlist$
rew $get_{vars}(PROC(v, s)) = v$
$get_{sums}(PROC(v, s)) = s$

B.5 Spec

In the sort spec a process specification is combined with a datatype to provide a complete specification. Such a specification forms the input of the Instantiator. Note that the function ssc only checks whether the process part of the specification satisfies a number of correctness requirements.

sort $Spec$
func $SPEC : Datatype \times ProcSpec \to Spec$
$get_{adt} : Spec \to Datatype$
$get_{proc} : Spec \to ProcSpec$
$ssc : Spec \to Bool$

var $a : Datatype$
$p : ProcSpec$
$sig : Signature$
$eql : Equationlist$
$vl : Variablelist$
$sl : Sumlist$
rew $get_{adt}(SPEC(a, p)) = a$
$get_{proc}(SPEC(a, p)) = p$
$ssc(SPEC(D(sig, eql), PROC(vl, sl))) = ssc(sl, vl, sig)$

B.6 Step

The output of the Stepper consists of a number of actions and a target state. A single such step is represented as $Step(a, tl, ntl)$, i.e., an action a with a termlist tl to indicate is parameters, and a targetstate ntl of sort $NTermlist$.

sort $Step$
func $ST : String \times Termlist \times NTermlist \to Step$

	$get_{act} : Step \rightarrow String$
	$get_{actterms} : Step \rightarrow Termlist$
	$get_{state} : Step \rightarrow NTermlist$
var	$a : String$
	$at : Termlist$
	$st : NTermlist$
rew	$get_{act}(ST(a, at, st)) = a$
	$get_{actterms}(ST(a, at, st)) = at$
	$get_{state}(ST(a, at, st)) = st$

B.7 Steplist

Steps, as provided in the previous section, are grouped as elements of the sort *Steplist*.

sort	*Steplist*
func	$emsl :\rightarrow Steplist$
	$ins : Step \times Steplist \rightarrow Steplist$
	$empty : Steplist \rightarrow Bool$
	$head : Steplist \rightarrow Step$
	$tail : Steplist \rightarrow Steplist$
var	$st : Step$
	$sl : Steplist$
rew	$empty(emsl) = \mathsf{t}$
	$empty(ins(st, sl)) = \mathsf{f}$
	$head(ins(st, sl)) = st$
	$tail(ins(st, sl)) = sl$

B.8 Transition

The results of the Instantiator consist of a transitionsystem. The major component of a transitionsystem is a list of transitions. Single transitions constitute the sort *Transition*. Each transition has the form $TR(tl_1, a, tl_2, ntl)$ where *Termlist* tl_1 is the source state, *String* a is the name of an action, tl_2 are the parameters of the action and *NTermlist* ntl is the targetstate. The operations are standard projection operators.

sort	*Transition*
func	$TR : Termlist \times String \times Termlist \times NTermlist \rightarrow Transition$
	$sourcestateof : Transition \rightarrow Termlist$
	$actof : Transition \rightarrow String$
	$acttermsof : Transition \rightarrow Termlist$
	$targetstateof : Transition \rightarrow NTermlist$
	$eq : Transition \times Transition \rightarrow Bool$
var	$a, a_1 : String$
	$st, at, st_1, at_1 : Termlist$
	$tt, tt_1 : NTermlist$
rew	$sourcestateof(TR(st, a, at, tt)) = st$
	$actof(TR(st, a, at, tt)) = a$
	$acttermsof(TR(st, a, at, tt)) = at$
	$targetstateof(TR(st, a, at, tt)) = tt$
	$eq(TR(st, a, at, tt), TR(st_1, a_1, at_1, tt_1)) =$

$and(eq(st, st_1), and(eq(a, a_1), and(eq(at, at_1), eq(tt, tt_1))))$

B.9 Transitionlist

Transitions are in a straightforward way combined to transitionlists. All functions are standard.

sort *Translist*
func $emts :\rightarrow Translist$
$ins : Transition \times Translist \rightarrow Translist$
$test : Transition \times Translist \rightarrow Bool$
$rem : Transition \times Translist \rightarrow Translist$
$len : Translist \rightarrow Nat$
$if : Bool \times Translist \times Translist \rightarrow Translist$

var $tr, tr_1 : Transition$
$tl, tl_1 : Translist$
rew $test(tr, emts) = \mathsf{t}$
$test(tr, ins(tr_1, tl)) = or(eq(tr, tr_1), test(tr, tl))$
$rem(tr, emts) = emts$
$rem(tr, ins(tr_1, tl)) = if(eq(tr, tr_1), rem(tr, tl), ins(tr_1, rem(tr, tl)))$
$len(emts) = 0$
$len(ins(tr, tl)) = S(len(tl))$
$if(\mathsf{t}, tl, tl_1) = tl$
$if(\mathsf{f}, tl, tl_1) = tl_1$

B.10 Transitionsystem

The combination of a *Datatype*, an initial state in the form of a *Termlist* and a list of transitions is a transitions system. These transition systems are the result of the Instantiator.

sort *Transsys*
func $TS : Datatype \times Termlist \times Translist \rightarrow Transsys$

C Realisation

In this section we provide the full toolbus code of a simple combination of the tools and the C code of all programmes that we have written. Note that the purpose of the C code as we have provided it, is only meant to show feasibility of realisation of our implementations and is neither optimised for execution speed or even readability. We provide the code without explanation.

C.1 A toolbus script

```
#define Specification \
  spec( \
   d(s(ins("Bool",ins("nat",ems)),\
    ins(f("T",ems,"Bool"),\
     ins(f("F",ems,"Bool"),emf)),\
      ins(f("inv",ins("Bool",ems),"Bool"),ins(f("S",ins("nat",ems),"nat"),\
     ins(f("0",ems,"nat"),emf)))),\
         ins(e(emv,t("inv",ins(t("T",emt),emt)),t("F",emt)),\
         ins(e(emv,t("inv",ins(t("F",emt),emt)),t("T",emt)),eme))),\
     proc(ins("x","nat",emv),\
```

```
       ins(smd(ins("y","Bool",emv),"a",ins(t("y",emt),emt),\
       i(ins(t("x",emt),emt)),t("inv",ins(t("y",emt),emt))),\
 ins(smd(emv,"b",emt,i(ins(t("S",ins(t("0",emt),emt)),emt)),t("T",emt)),\
       ins(smd(emv,"c",emt,i(ins(t("0",emt),emt)),t("T",emt)),eml)))))

#define Initial \
       ins(t("S",ins(t("S",ins(t("0",emt),emt)),emt)),emt)

process INSTANTIATOR is
let C : instantiator,
    T : term
in
   execute(instantiator, C?) .
   (rec-event(C,send,u2(T?)) . printf("sendINSTu2 %t\n",T) .
                    snd-msg(u2(T)) . snd-ack-event(C,send) +
    rec-event(C,send,c1(T?)) . printf("sendINSTc1 %t\n",T) .
                    snd-msg(c1(T)) . snd-ack-event(C,send) +
    rec-event(C,send,s1(T?)) . printf("sendINSTs1 %t\n",T) .
                    snd-msg(s1(T)) . snd-ack-event(C,send) +
    rec-event(C,receivec2,"") . rec-msg(c2(T?)) . snd-do(C,c2(T)) .
        printf("receiveINSc2 %t\n",T) . snd-ack-event(C,receivec2) +
    rec-event(C,receives2,"") . rec-msg(s2(T?)) .
             printf("receiveINSs2 %t\n",T) . snd-do(C,s2(T)) .
        snd-ack-event(C,receives2) +
    rec-event(C,receiveu1,"") . rec-msg(u1(T?)) .
          printf("receiveINSu1 %t\n",T) . snd-do(C,u1(T)) .
        snd-ack-event(C,receiveu1)) *
    rec-disconnect(C)
endlet

process STEPPER is
let S : stepper,
    T : term
in
   execute(stepper, S?) .
   (rec-event(S,send,e1(T?)) . snd-msg(e1(T)) .
           printf("sendSTEPe1 %t\n",T) . snd-ack-event(S,send) +
    rec-event(S,send,c2(T?)) . snd-msg(c2(T)) .
           printf("sendSTEPc2 %t\n",T) . snd-ack-event(S,send) +
    rec-event(S,receivee2,"") . rec-msg(e2(T?)) .
           printf("receiveSTEPe2 %t\n",T) . snd-do(S,e2(T)).
         snd-ack-event(S,receivee2) +
    rec-event(S,receivec1,"") . rec-msg(c1(T?)) .
          printf("receiveSTEPc1 %t\n",T) . snd-do(S,c1(T)) .
         snd-ack-event(S,receivec1)) *
    rec-disconnect(S)
endlet

process ENUMERATOR is
let E : enumerator,
    T : term
in
   execute(enumerator, E?) .
   (rec-event(E,send,e2(T?)) .
         printf("sendENUMe2 %t\n",T) . snd-msg(e2(T)) . snd-ack-event(E,send) +
```

```
    rec-event(E,send,r1(T?)) . printf("sendENUMr1 %t\n",T) .
              snd-msg(r1(T)) . snd-ack-event(E,send) +
    rec-event(E,receiver2,"") . rec-msg(r2(T?)) .
              printf("receiveENUMr2 %t\n",T) . snd-do(E,r2(T)) .
        snd-ack-event(E,receiver2) +
    rec-event(E,receivee1,"") . rec-msg(e1(T?)) .
        printf("receiveENUMe1 %t\n",T) . snd-do(E,e1(T)) .
        snd-ack-event(E,receivee1)) *
    rec-disconnect(E)
endlet

process REPRESENTANT1 is
let R1 : representant,
    T : term
in
   execute(representant, R1?) .
   (rec-event(R1,send,r2(T?)) . printf("sendR1 %t \n",T) .
               snd-msg(r2(T)) . snd-ack-event(R1,send) +
    rec-event(R1,receiver1,"") .  rec-msg(r1(T?)) . printf("receiveR1 %t \n",T)
                   . snd-do(R1,r1(T)) .
        snd-ack-event(R1,receiver1)) *
    rec-disconnect(R1)
endlet

process REPRESENTANT2 is
let R2 : representant,
    T : term
in
   execute(representant, R2?) .
   (rec-event(R2,send,r2(T?)) . printf("sendR2 %t \n",T) .
                snd-msg(s2(T)) . snd-ack-event(R2,send) +
    rec-event(R2,receiver1,"") .  rec-msg(s1(T?)) .
          printf("receiveR2 %t \n",T) . snd-do(R2,r1(T)) .
        snd-ack-event(R2,receiver1)) *
    rec-disconnect(R2)
endlet

process TESTER is
let T1 : term,
    T2 : term,
    T3 : term,
    S : str
in
  snd-msg(u1(Specification)) .
  (rec-msg(u2(notok(S?))) . printf("PROBLEM %s\n",S) .
                       snd-msg(u1(terminate)) +
   rec-msg(u2(ok)) .
   snd-msg(u1(Initial)) .
      ( rec-msg(u2(ts(T1?,T2?,T3?))) .
          printf("RESULT:\nts(%t,%t,%t)\n",T1,T2,T3) +
        rec-msg(u2(notok(S?))) . printf("NOTOK %s\n",S)) .
                   snd-msg(u1(terminate)))
endlet

tool representant is {command = "representant"}
```

```
tool enumerator is {command = "enumerator"}
tool stepper is {command = "stepper"}
tool instantiator is {command = "instantiator"}

toolbus(STEPPER,ENUMERATOR,REPRESENTANT1,REPRESENTANT2,TESTER,INSTANTIATOR)
```

C.2 datatypes.h

```
#ifndef __DATATYPE_H__
#define __DATATYPE_H__

#include <TB.h>

void error(char *s, term *t);

int correct(term *termlist, term *proc, term *adt, char **reason);
int welltyped(term *t, term *d);

int ssc(term *d, char **errormessage);
int ssc_spec(term *d, char **errormessage);
int correctopenterm(term *t, term *d);

term *lnsubstituteterm(term *terms, term *tlist, term *vars);
term *substituteterm(term *terms, term *singleterm, term *vars);
term *lsubstituteterm(term *terms, term *tlist, term *vars);
term *repl(char *name,term *subs, term *t);
term *lrepl(char *name, term *subs, term *termlist);
term *conc_var(term *l1, term *l2);
int closedl(term *t,term *sig);
int closed(term *t,term *sig);

int isemptytl(term *l);
term *toe(term *l);
term *untoe(term *l);
int iseltll(term *t, term *l);
int isemptytll(term *l);
term *remall(term *t, term *l);

#endif /* __DATATYPE_H__ */
```

C.3 datatypes.c

```
#include "datatypes.h"

term *lsubstituteterm(term *terms, term *tlist, term *vars);
term *repl(char *name,term *subs, term *t);
static int noclash(term *vars, term *funcs);
int eqterml(term *l1, term *l2);
int nodoubles(term *t, term *vars);
int notin(char *n, term *t, term *t1);
term *conc_tl(term *t, term *t1);

void error(char *s, term *t)
```

```
{ fprintf(stderr,"Error: %s: \n",s);
  TBprintf(stderr,"Problematic term \"%t\" \n", t);
  fprintf(stderr,"\n\n\n\n");
  exit(0);}

/* WELLTYPED */


static int sortsequal(term *t1, term *t2)
{ char *hd1, *hd2;
  term *tl1, *tl2;
  if (TBmatch(t1,"ems"))
    {if (TBmatch(t2,"ems"))  return(TBtrue);
     else return(TBfalse);}
  if (!TBmatch(t1,"ins(%s,%t)",&hd1,&tl1)) error("sortsequal expects term
        of sort stringlist",t1);
  if (!TBmatch(t2,"ins(%s,%t)",&hd2,&tl2)) error("sortsequal expects term
        of sort stringlist (2)",t2);
  if (strcmp(hd1,hd2)!=0) return(TBfalse);
  return(sortsequal(tl1,tl2));
}


static int exists1(char *str, term *sorts, term *list, char **sort)
{ term *slist, *flist;
  char *s1, *s2;
  while (!TBmatch(list,"emf")){
  if (!TBmatch(list,"ins(f(%s,%t,%s),%t)",&s1,&slist,&s2,&flist))
          error("exists1 receives term that is not of sort Functionlist",list);
  if ((strcmp(s1,str)==0) && sortsequal(sorts,slist))
     {*sort=s2;
      return(TBtrue);}
  list=flist;
  }
  return(TBfalse);

}

int existvar(char *str, term *vars, char **sort)
{ char *s1,*s2;
  term *vars1;
  while (!TBmatch(vars,"emv"))
  { if (!TBmatch(vars,"ins(%s,%s,%t)",&s1,&s2,&vars1))
               error("existvar expects variablelist",vars);
    if (strcmp(s1,str)==0) {*sort=s2; return(TBtrue);}
    vars=vars1;
  }
  return(TBfalse);
}


int exists(char *str, term *sorts, term *sig, char **sort)
{ term *sorts1, *functions, *constr;
  if (!TBmatch(sig,"s(%t,%t,%t)",&sorts1,&constr,&functions))
       error("Wrong term of sort signature in exists",sig);
```

```
  if (exists1(str, sorts,functions,sort)) return(TBtrue);
  if (exists1(str, sorts,constr,sort)) return(TBtrue);
  return(TBfalse);
}

static int welltyped2(term *t, term *sig, term *vars, term **sorts);

static int welltyped1(term *t, term *sig, term *vars, char **sort)
{  /* t refers to a closed term of type Term,
              and sig is a term of type Signature */
   char *str;
   term *tlist,*sorts;

   if (TBmatch(t,"t(%s,%t)",&str,&tlist))
      { if (!welltyped2(tlist,sig,vars,&sorts)) return(TBfalse);
        if (exists(str,sorts,sig,sort)) return(TBtrue);
        if (TBmatch(tlist,"emt") && existvar(str,vars,sort)) return(TBtrue);
        return(TBfalse);
      }
   else { error("Welltyped1 receives a term that is not of type Term",t);}
   return(TBfalse);
}

static int welltyped2(term *t, term *sig, term *vars, term **sorts)
{  /* t refers to a closed term of type Termlist; In case t is well typed,
      the function yields true and sorts refers to a term of type stringlist,
      giving the type of t */

   term *tm, *tlist, **tempsorts;
   char *sort;

   if (TBmatch(t,"emt"))
      { *sorts=TBmake("ems"); return(TBtrue); }
   else if (TBmatch(t,"ins(%t,%t)",&tm,&tlist))
      {  if (!welltyped1(tm,sig,vars,&sort)) return(TBfalse);
         if (!welltyped2(tlist, sig,vars,sorts)) return(TBfalse);
         tempsorts=sorts;
         *sorts=TBmake("ins(%s,%t)",sort,*sorts);
         return(TBtrue);
      }
   else error("Welltyped1 receives a term that is not of type Termlist",t);
   return(TBfalse);
}

int welltyped(term *t, term *sig)
{  /* t refers to a closed term of sort Term,
              and sig is a term of sort signature */
  char *tmp;
  term *vars;
  vars=TBmake("emv");
  return(welltyped1(t,sig,vars,&tmp));
}

/*   Statically Semantically Correct --- SCC   */
```

```
static int sortexist(char *sort, term *sortlist)
{ term *tl;
  char *hd;
  if (TBmatch(sortlist,"ems")) return(TBfalse);
  if (!TBmatch(sortlist,"ins(%s,%t)",&hd,&tl))
            error("sortexist expects term of sort stringlist",sortlist);
  if (strcmp(sort,hd)==0) return(TBtrue);
  return(sortexist(sort,tl));
}

static int sortlistexist(term *sortlist, term *sorts)
{ char *name;
  term *tl;
  if (TBmatch(sortlist,"ems")) return(TBtrue);
  if (!TBmatch(sortlist,"ins(%s,%t)",&name,&tl))
       error("sortlistexist expects term of sort stringlist",sortlist);
  if (!sortexist(name,sorts)) return(TBfalse);
  return(sortlistexist(tl,sorts));
}

static int sortsexist(term *sorts,term *functionlist)
{ term *args, *tl;
  char *name, *target;
  if (TBmatch(functionlist,"emf")) return(TBtrue);
  if (!TBmatch(functionlist,"ins(f(%s,%t,%s),%t)",&name,&args,&target,&tl))
         error("sortsexist expects term of sort functionlist",functionlist);
  if (!sortexist(target,sorts)) return(TBfalse);
  if (!sortlistexist(args,sorts)) return(TBfalse);
  return(TBtrue);
}

static int boolsexist(term *constr,int truedetected, int falsedetected)
{ term *tl;
  if (TBmatch(constr,"emf")) return(TBfalse);
  if (TBmatch(constr,"ins(f(\"T\",ems,\"Bool\"),%t)",&tl))
      {if (falsedetected)
          return(TBtrue);
       else return(boolsexist(tl,TBtrue,TBfalse));}
  if (TBmatch(constr,"ins(f(\"F\",ems,\"Bool\"),%t)",&tl))
      {if (truedetected)
          return(TBtrue);
       else return(boolsexist(tl,TBfalse,TBtrue));}
  if (!TBmatch(constr,"ins(f(<str>,<term>,<str>),%t)",&tl))
          error("boolsexist expects term of sort functionlist",constr);
  return(boolsexist(tl,truedetected,falsedetected));
}

static int checkunique(char *name, term *args, char *target, term *conslist)
{ char *cname, *ctarget;
  term *cargs, *tl;
  if (TBmatch(conslist,"emf")) return(TBtrue);
  if (!TBmatch(conslist,"ins(f(%s,%t,%s),%t)",&cname,&cargs,&ctarget,&tl))
     error("checkunique expects term of sort functionlist",conslist);
  if ((strcmp(name,cname)==0) &&
        sortsequal(args,cargs) &&
```

```
        strcmp(target,ctarget)!=0) return(TBfalse);
  return(checkunique(name,args,target,tl));
}

static int uniquetarget1(term *conslist)
{ char *name, *target;
  term *args, *tl;
  if (TBmatch(conslist,"emf")) return(TBtrue);
  if (!TBmatch(conslist,"ins(f(%s,%t,%s),%t)",&name,&args,&target,&tl))
     error("uniquetarget1 expects term of sort functionlist",conslist);
  if (!checkunique(name,args,target,tl)) return(TBfalse);
  return(uniquetarget1(tl));
}

static int uniquetarget(term *funclist, term *conslist)
{ char *name, *target;
  term *args, *tl;
  if (TBmatch(funclist,"emf")) return(uniquetarget1(conslist));
  if (!TBmatch(funclist,"ins(f(%s,%t,%s),%t)",&name,&args,&target,&tl))
     error("uniquetarget expects term of sort functionlist",conslist);
  if (!checkunique(name,args,target,tl)) return(TBfalse);
  if (!checkunique(name,args,target,conslist)) return(TBfalse);
  return(uniquetarget(tl,conslist));
}

static int checksig(term *sig, char **errormessage)
{ /* uniquetarget, sortsexist, boolsexist */
  term *sorts, *func, *constr;
  if (!TBmatch(sig,"s(%t,%t,%t)",&sorts,&constr,&func))
     error("checksig expects signature",sig);
  if (!sortsexist(sorts,func))
        {*errormessage="Unknown sort in functions";
         return(TBfalse);}
  if (!sortsexist(sorts,constr))
        {*errormessage="Unknown sort in constructors";
         return(TBfalse);}
  if (!boolsexist(constr,0,0))
        {*errormessage="Sort Bool with constructors T and F does not exist";
         return(TBfalse);}
  if (!uniquetarget(func,constr))
         {*errormessage="A function does not have a unique target sort";
         return(TBfalse);}
  if (!uniquetarget1(constr))
         {*errormessage="A constructor does not have a unique target sort";
         return(TBfalse);}
  return(TBtrue);
}

static int correcteqs(term *eqs, term *sig, char **errormessage)
{  term *vars, *t1, *t2, *tail;
   char *sort1,*sort2;

   if (TBmatch(eqs,"eme")) return(TBtrue);
   if (!TBmatch(eqs,"ins(e(%t,%t,%t),%t)",&vars,&t1,&t2,&tail))
          error("correcteqs expects term of sort equation",eqs);
```

```
   if (!welltyped1(t1,sig,vars,&sort1))
      {*errormessage="Lefthandside of an equation is badly typed";
       return(TBfalse);}
   if (!welltyped1(t2,sig,vars,&sort2))
      {*errormessage="Righthandside of an equation is badly typed";
       return(TBfalse);}
   if (strcmp(sort1,sort2)!=0)
      {*errormessage="Equation with different sorts left and right";
       return(TBfalse);}
   if (!nodoubles(t1,vars))
      {*errormessage="Equation with left-hand side containing double occurrence of variable";
       return(TBfalse);}
    return(correcteqs(tail,sig,errormessage));
}

int nodoubles(term *t, term *vars)   /* checks whether no variable from
                                        variable list vars occurs doubly
                                        in term or term list t */
{term *args, *tail;
 char *n, *srt;

 if (TBmatch(t,"emt")) return(TBtrue);
 if (TBmatch(t,"t(%s,%t)",&n,&args)){
   if (existvar(n,vars,&srt) && !notin(n,args,TBmake("emt"))) return(TBfalse);
   return(nodoubles(args,vars));
 };
 if (TBmatch(t,"ins(t(%s,%t),%t)",&n,&args,&tail)){
   if (existvar(n,vars,&srt) && !notin(n,args,tail)) return(TBfalse);
   return(nodoubles(conc_tl(args,tail),vars));
 };
 error("nodoubles expects term or term list in first argument",t);
 return(TBfalse);
}

int notin(char *n, term *t, term *t1)   /* checks whether string n does not
                                           occur in term lists t and t1 */
{term *args, *tail;
 char *n1;

 if (TBmatch(t,"emt") && TBmatch(t1,"emt")) return(TBtrue);
 if (TBmatch(t,"emt")) return(notin(n,t1,TBmake("emt")));
 if (TBmatch(t,"ins(t(%s,%t),%t)",&n1,&args,&tail)){
   if (strcmp(n,n1)==0) return(TBfalse);
   return(notin(n,args,tail));
 };
 error("notin expects term list in first argument",t);
 return(TBfalse);
}

term *conc_tl(term *t, term *t1)   /* returns the termlist obtained by
                                      concatenating t and t1 */
{term *head,*tail;
 if (TBmatch(t,"emt"))
   return(t1);
 if (TBmatch(t,"ins(%t,%t)",&head,&tail))
```

```
     return(TBmake("ins(%t,%t)",head,conc_tl(tail,t1)));
 error("conc_tl expects term list in first argument",t);
 return(TBfalse);
}

int ssc(term *d, char **errormessage)
{   /* uniquetarget
       sortsexist
       correcteqs (done)
       boolsexist  */
  term *sig, *eqs;
  if (!TBmatch(d,"d(%t,%t)",&sig,&eqs))
         error("ssc expects term of type datatype",d);
  if (!checksig(sig,errormessage)) return(TBfalse);
  if (!correcteqs(eqs,sig,errormessage)) return(TBfalse);
  return(TBtrue);
}

static int matchsort1(term *vl, term *sorts)
{ term *tl1, *tl2;
  char *s1, *s2;
  if (TBmatch(vl,"emv"))
     { if (TBmatch(sorts,"ems"))
          return(TBtrue);
          return(TBfalse);}
  if (TBmatch(vl,"ins(<str>,%s,%t)",&s1,&tl1))
     { if (TBmatch(sorts,"ins(%s,%t)",&s2,&tl2))
          {if (strcmp(s1,s2)==0)
              return(matchsort1(tl1,tl2));
              return(TBfalse);}
       return(TBfalse);}
  error("Matchsort1 expects term of sort variablelist",vl);
  return(TBfalse);
}

static int matchsort(term *vl, term *sorts, term *ntl)
{ term *tl;
  if (TBmatch(ntl,"none"))
     return(TBtrue);
  if (TBmatch(ntl,"i(%t)",&tl))
     return(matchsort1(vl,sorts));
  error("Matchsort expects term of sort NTermlist",ntl);
  return(TBfalse);
}

static int welltyped3(term *ntl, term *sig, term *vl, term **sorts)
{ term *tl;
  if (TBmatch(ntl,"none"))
     return(TBtrue);
  if (TBmatch(ntl,"i(%t)",&tl))
     return(welltyped2(tl,sig,vl,sorts));
  error("Welltyped3 expects term of sort NTermlist",vl);
  return(TBfalse);
}
```

```
static int nooverlap(term *l1, term *l2)
{ term *tl;
  char *s,*s1;
  if (TBmatch(l1,"emv"))
     return(TBtrue);
  if (TBmatch(l1,"ins(%s,<str>,%t)",&s,&tl))
     { if (existvar(s,l2,&s1))
          return(TBfalse);
          return(nooverlap(tl,l2));}
  error("Nooverlap expects variablelist",l1);
  return(TBfalse);
}

term *conc_var(term *l1, term *l2)
{ term *tl;
  char *s1, *s2;
  if (TBmatch(l1,"emv"))
     return(l2);
  if (TBmatch(l1,"ins(%s,%s,%t)",&s1,&s2,&tl))
     return(TBmake("ins(%s,%s,%t)",s1,s2,conc_var(tl,l2)));
  error("Conc expects variablelist",l1);
  return(TBfalse);
}

static int ssc_triple(term *sl, term *vl, term *sig, char **errormessage)
{ term *vl1, *tl, *ntl, *t, *sl1, *sorts, *cons, *funcs;
  char *s, *sort;
  if (TBmatch(sl,"eml"))
     return(TBtrue);
  if (TBmatch(sl,"ins(smd(%t,%s,%t,%t,%t),%t)",&vl1,&s,&tl,&ntl,&t,&sl1))
     {
       if (!welltyped1(t,sig,conc_var(vl1,vl),&sort))
          {*errormessage="Conditions are not all well typed";
           return(TBfalse);}
       if (strcmp(sort,"Bool")!=0)
          {*errormessage="Condition are not all of sort Bool";
           return(TBfalse);}
       if (!welltyped2(tl,sig,conc_var(vl1,vl),&sorts))
          {*errormessage="Arguments of an action are not well typed";
           return(TBfalse);}
       if (!TBmatch(sig,"s(<term>,%t,%t)",&cons,&funcs))
          error("Ssc_triple expects signature",sig);
       if (!noclash(vl,cons))
          {*errormessage="Vars in proc params clash with constructor in sig";
           return(TBfalse);}
       if (!noclash(vl,funcs))
          {*errormessage="Vars in proc params clash with function in sig";
           return(TBfalse);}
       if (!noclash(vl1,cons))
          {*errormessage="Vars in sum clash with constructor constant in sig";
           return(TBfalse);}
       if (!noclash(vl1,funcs))
          {*errormessage="Vars in sum clash with function constant in sig";
           return(TBfalse);}
       if (!nooverlap(vl,vl1))
```

```
          {*errormessage="Vars in process params and vars in sum clash";
           return(TBfalse);}
       if (!welltyped3(ntl,sig,conc_var(vl1,vl),&sorts))
          {*errormessage="Process args are not well typed";
           return(TBfalse);}
       if (!matchsort(vl,sorts,ntl))
          {*errormessage="Process args do not match sorts of process params";
           return(TBfalse);}
       return(ssc_triple(sl1,vl,sig,errormessage));
     }
  error("Ssc_triple expects sumlist",sl);
  return(TBfalse);
}

int ssc_spec(term *d, char **errormessage)
{ term *sl, *vl, *sig;
  if (TBmatch(d,"spec(d(%t,<term>),proc(%t,%t))",&sig,&vl,&sl))
     return(ssc_triple(sl,vl,sig,errormessage));
  error("Ssc_spec expects term of sort ProcSpec",d);
  return(TBfalse);
}

/* ---------------  CORRECT OPENTERM ------------------ */
static int sortsexist1(term *vars, term *sorts)
{ term *tl;
  char *sort;
  if (TBmatch(vars,"emv")) return(TBtrue);
  if (!TBmatch(vars,"ins(<str>,%s,%t)",&sort,&tl))
       error("sortsexist1 expects term of sort variablelist",vars);
  if (!sortexist(sort,sorts)) return(TBfalse);
  return(sortsexist1(tl,sorts));
}

static int noclash1(char *name, term *funcs)
{ term *tl;
  char *lname;
  if (TBmatch(funcs,"emf")) return(TBtrue);
  if (TBmatch(funcs,"ins(f(<str>,ins(<str>,<term>),<str>),%t)",&tl))
       return(noclash1(name,tl));
  if (TBmatch(funcs,"ins(f(%s,ems,<str>),%t)",&lname,&tl))
     {if (strcmp(name,lname)==0)
         return(TBfalse);
      else return(noclash1(name,tl));}
  error("noclash1 expects term of sort functionlist",funcs);
  return(TBfalse);
}

static int noclash(term *vars, term *funcs)
{ term *tl;
  char *name;
  if (TBmatch(vars,"emv")) return(TBtrue);
  if (!TBmatch(vars,"ins(%s,<str>,%t)",&name,&tl))
       error("noclash expects term of sort variablelist",vars);
  if (!noclash1(name,funcs)) return(TBfalse);
  return(noclash(tl,funcs));
```

```
}

int correctopenterm(term *t, term *sig)
{
  term *tm, *vars, *sorts, *funcs, *constr;
  char *dummysort;
  if (!TBmatch(t,"o(%t,%t)",&tm,&vars))
        error("correctopenterm expects term of sort openterm",t);
  if (!TBmatch(sig,"s(%t,%t,%t)",&sorts,&constr,&funcs))
        error("correctopenterm expects term of sort signature",sig);
  if (!sortsexist1(vars,sorts)) return(TBfalse);
  if (!noclash(vars, funcs)) return(TBfalse);
  if (!noclash(vars, constr)) return(TBfalse);
  return(welltyped1(tm,sig,vars,&dummysort));
}

/* ---------------  SUBSTITUTIONS  lsubstitute, substitute, repl, lrepl   ---------------*/
term *lnsubstituteterm(term *terms, term *tlist, term *vars)
{ term *tl;
  if (TBmatch(tlist,"none"))
     return(TBmake("none"));
  if (TBmatch(tlist,"i(%t)",&tl))
     return(TBmake("i(%t)",lsubstituteterm(terms,tl,vars)));
  error("lnsubstituterm expects NTermlist",tlist);
  return(TBfalse);
}

term *lsubstituteterm(term *terms, term *tlist, term *vars)
{ term *hd, *tl;
  if (TBmatch(tlist,"emt")) return(TBmake("emt"));
  if (TBmatch(tlist,"ins(%t,%t)",&hd,&tl))
     return(TBmake("ins(%t,%t)",substituteterm(terms,hd,vars),
                               lsubstituteterm(terms,tl,vars)));
  error("lsubstituteterm expects a termlist",tlist);
  return(TBfalse);
}

term *substituteterm(term *terms, term *singleterm, term *vars)
{ term *hd, *tl, *vartl;
  char *var;
  if (TBmatch(terms,"emt"))
     { if (TBmatch(vars,"emv"))
            return(singleterm);
       error("term and variablelist should have the same length in substituteterm",vars);
     }
  if (TBmatch(terms,"ins(%t,%t)",&hd,&tl))
     { if (TBmatch(vars,"ins(%s,<str>,%t)",&var,&vartl))
          { return(substituteterm(tl,repl(var,hd,singleterm),vartl));
          }
       error("Substituteterm expects non empty variablelist",vars);
     }
  error("substituteterm expects a termlist",terms);
  return(TBfalse);
}
```

```
term *repl(char *name,term *subs, term *t)
{ char *name1;
  term *args;
  if (TBmatch(t,"t(%s,emt)",&name1))
     { if (strcmp(name,name1)==0)
           return(subs);
       return(t);}
  if (TBmatch(t,"t(%s,%t)",&name1,&args))
     return(TBmake("t(%s,%t)",name1,lrepl(name,subs,args)));
  error("repl expects term",t);
  return(TBfalse);
}

term *lrepl(char *name, term *subs, term *termlist)
{ term *hd, *tl;
  if (TBmatch(termlist,"emt"))
     return(termlist);
  if (TBmatch(termlist,"ins(%t,%t)",&hd,&tl))
     return(TBmake("ins(%t,%t)",repl(name,subs,hd),lrepl(name,subs,tl)));
  error("lrepl expects termlist",termlist);
  return(TBfalse);
}

int correct(term *termlist, term *proc, term *adt, char **reason)
{ term *sig, *vl, *sorts;
  if (!TBmatch(adt,"d(%t,<term>)",&sig))
     error("Correct expects datatype",adt);
  if (!TBmatch(proc,"proc(%t,<term>)", &vl))
     error("Correct expects ProcSpec",proc);
  if (!welltyped2(termlist,sig,TBmake("emv"),&sorts))
     {*reason="Terms are not well typed";
      return(TBfalse);}
  if (!matchsort1(vl,sorts))
     {*reason="Sorts of terms do not match those of process pars";
      return(TBfalse);}
  return(TBtrue);
}

int isemptytl(term *l)   /* checks whether termlist l is empty */
{if (TBmatch(l,"emt"))
   return(TBtrue);
 if (!TBmatch(l,"ins(<term>,<term>)"))
   error("isemptytl expects emt or ins(..., ...)",l);
 return(TBfalse);
}

term *toe(term *l)   /* returns the "last" term of termlist l */
{term *hd,*tl;
 if (TBmatch(l,"emt"))
   error("toe got an empty termlist",l);
 if (!TBmatch(l,"ins(%t,%t)",&hd,&tl))
   error("toe expects ins(..., ...)",l);
 if (TBmatch(tl,"emt"))
   return(hd);
 return(toe(tl));
```

```
}

term *untoe(term *l)   /* returns the termlist obtained by removing
                          from l its "last" element */
{term *hd,*tl;
 if (TBmatch(l,"emt"))
   error("untoe got an empty termlist",l);
 if (!TBmatch(l,"ins(%t,%t)",&hd,&tl))
   error("untoe expects ins(..., ...)",l);
 if (TBmatch(tl,"emt"))
   return(tl);
 return(TBmake("ins(%t,%t)",hd,untoe(tl)));
}

int iseltll(term *t, term *l)   /* checks whether termlist t is
                                   element of termlistlist l */
{term *hd,*tl;
 if (TBmatch(l,"emtll"))
   return(TBfalse);
 if (!TBmatch(l,"ins(%t,%t)",&hd,&tl))
   error("iseltll expects emtll or ins(..., ...) in second argument",l);
 if (eqterml(t,hd))
   return(TBtrue);
 return(iseltll(t,tl));
}

int eqterm(term *t1, term *t2)   /* checks whether terms t1 and t2 are
                                    equal */
{char *n1,*n2;
 term *l1,*l2;
 if (!TBmatch(t1,"t(%s,%t)",&n1,&l1))
   error("eqterm expects t(..., ...) in first argument",t1);
 if (!TBmatch(t2,"t(%s,%t)",&n2,&l2))
   error("eqterm expects t(..., ...) in second argument",t2);
 if (strcmp(n1,n2)!=0)
   return(TBfalse);
 return(eqterml(l1,l2));
}

int eqterml(term *l1, term *l2)   /* checks whether termlists l1 and
                                     l2 are equal */
{term *hd1,*hd2,*tl1,*tl2;
 if (TBmatch(l1,"emt")){
   if (TBmatch(l2,"emt"))
     return(TBtrue);
   else
     return(TBfalse);
 }
 if (!TBmatch(l1,"ins(%t,%t)",&hd1,&tl1))
   error("eqterml expects ins(..., ...) in first argument",l1);
 if (!TBmatch(l2,"ins(%t,%t)",&hd2,&tl2))
   error("eqterml expects ins(..., ...) in second argument",l2);
 if (!eqterm(hd1,hd2))
   return(TBfalse);
 return(eqterml(tl1,tl2));
```

```
}

int isemptytll(term *l)   /* checks whether termlistlist l is empty */
{if (TBmatch(l,"emtll"))
   return(TBtrue);
 if (!TBmatch(l,"ins(<term>,<term>)"))
   error("isemptytll expects emtll or ins(..., ...)",l);
 return(TBfalse);
}


term *remall(term *t, term *l)   /* returns the termlistlist obtained
                                    by removing all occurrences of
                                    termlist t in termlistlist l */
{term *hd,*tl;
 if (TBmatch(l,"emtll"))
   return(l);
 if (!TBmatch(l,"ins(%t,%t)",&hd,&tl))
   error("remall expects emtll or ins(..., ...)",l);
 if (eqterml(t,hd))
   return(remall(t,tl));
 return(TBmake("ins(%t,%t)",hd,remall(t,tl)));
}

int correctopenboolterm(term *t, term *sig)
{ term *tm,*vars =NULL;
  char *sort="";
  if (!correctopenterm(t,sig))
     return(TBfalse);
  if (!TBmatch(t,"o(%t,%t)",&tm,&vars))
     error("correctopenboolterm expects openterm",t);
  if (!welltyped1(tm,sig,vars,&sort))
     return(TBfalse);
  if (strcmp(sort,"Bool")==0)
     return(TBtrue);
  return(TBfalse);
}

int closedl(term *tl, term *sig)
{ term *hd, *tail;
  if (TBmatch(tl,"emt"))
     return(TBtrue);
  if (!TBmatch(tl,"ins(%t,%t)",&hd,&tail))
     error("closedl expects termlist",tl);
  if (!closed(hd,sig))
     return(TBfalse);
  return(closedl(tail,sig));
}

int closed(term *t,term *sig)
{ term *args ;
  char *fn, *dummy;
  if (!TBmatch(t,"t(%s,%t)",&fn,&args))
     error("closed expects term",t);
  if (TBmatch(args,"emt"))
```

```
     { if (exists(fn,TBmake("ems"),sig,&dummy))
       {return(TBtrue);}
        return(TBfalse);}
  return(closedl(args,sig));
}
```

C.4 rewriter.c

```
#include <stdio.h>
#include <string.h>
#include "TB.h"

term *subst1(term *s, term *t);
term *lsubst1(term *s, term *t);
term *repl1(char *n, char *ss, term *s);
term *sconcat(term *s1, term *s2);
int length(term *l);
term *match(term *t1, term *t2);
term *lmatch(term *l1, term *l2);
term *toprewrite(term *t, term *e);
term *lrewrite(term *l, term *e);
term *rewrite(term *t, term *e);
term *lunprep1(term *l);
term *concat(term *v1, term *v2);
term *lprep(term *l, term *v);
term *test(char *n, term *v);

/***************** CONVERSION TO AND FROM COMPACT TERMS *********************/

term *prep(term *t)   /* returns the compact term for open or
                         closed term t */
{char *n,*s,*ss;
 term *v,*t1,*l1,*tt;
 if (TBmatch(t,"t(%s,%t)",&ss,&tt))   /* TBmatch(t,"t(<str>,<term>)") */
   return(t);
 if (TBmatch(t,"o(t(%s,emt),%t)",&n,&v)){
   if (TBmatch(test(n,v),"no"))
     return(TBmake("t(%s,emt)",n));
   if (TBmatch(test(n,v),"sort(%s)",&s))
     return(TBmake("t(var(%s,%s),emt)",n,s));
   error("prep expects no or sort(...) from test",test(n,v));
 }
 if (TBmatch(t,"o(t(%s,ins(%t,%t)),%t)",&n,&t1,&l1,&v))
   return(TBmake("t(%s,%t)",n,lprep(TBmake("ins(%t,%t)",t1,l1),v)));
 error("prep expects a closed or open term",t);
 return(TBmake("zzprep"));
}

term *lprep(term *l, term *v)   /* returns the list of terms obtained
                                   by applying prep to each of the
                                   terms of termlist l w.r.t. the
                                   variables in variable list v */
{term *t1,*l1;
 if (TBmatch(l,"emt"))
   return(l);
```

```
 if (TBmatch(l,"ins(%t,%t)",&t1,&l1))
   return(TBmake("ins(%t,%t)",prep(TBmake("o(%t,%t)",t1,v)),lprep(l1,v)));
 error("lprep expects a termlist as first argument",l);
 return(TBmake("zzlprep"));
}

term *test(char *n, term *v)   /* returns "sort(s)" if string n occurs
                                  as variable name with sort s in
                                  variablelist v, and "no" otherwise */
{char *n1,*s1;
 term *v1;
 if (TBmatch(v,"emv"))
   return(TBmake("no"));
 if (TBmatch(v,"ins(%s,%s,%t)",&n1,&s1,&v1)){
   if (strcmp(n,n1)==0)
     return(TBmake("sort(%s)",s1));
   else
     return(test(n,v1));
 }
 error("test expects a variable list as second argument",v);
 return(TBmake("zztest"));
}

term *unprep1(term *t)   /* returns the open term for compact
                            term t */
{char *n,*s;
 term *t1,*l1,*trms,*vrs;
 if (TBmatch(t,"t(var(%s,%s),emt)",&n,&s))
   return(TBmake("o(t(%s,emt),ins(%s,%s,emv))",n,n,s));
 if (TBmatch(t,"t(%s,emt)",&n))
   return(TBmake("o(t(%s,emt),emv)",n));
 if (TBmatch(t,"t(%s,ins(%t,%t))",&n,&t1,&l1)){
   TBmatch(lunprep1(TBmake("ins(%t,%t)",t1,l1)),"o(%t,%t)",&trms,&vrs);
   return(TBmake("o(t(%s,%t),%t)",n,trms,vrs));
 }
 error("unprep1 expects a term",t);
 return(TBmake("zzunprep1"));
}

term *lunprep1(term *l)   /* returns o(l1,v1) where l1 is the list of
                             terms obtained by applying unprep1 to the
                             terms of l, but the variables are
                             collected in v1 */
{term *t1,*l1,*trm,*vrs,*ltrms,*lvrs;
 if (TBmatch(l,"emt"))
   return(TBmake("o(emt,emv)"));
 if (TBmatch(l,"ins(%t,%t)",&t1,&l1)){
   TBmatch(unprep1(t1),"o(%t,%t)",&trm,&vrs);
   TBmatch(lunprep1(l1),"o(%t,%t)",&ltrms,&lvrs);
   return(TBmake("o(ins(%t,%t),%t)",trm,ltrms,concat(vrs,lvrs)));
 }
 error("lunprep1 expects a termlist",l);
 return(TBmake("zzlunprep1"));
}
```

```
term *concat(term *v1, term *v2)   /* returns the variablelist
                                      obtained by concatenating
                                      variablelists v1 and v2 */
{char *n,*s;
 term *w1;
 if (TBmatch(v1,"emv"))
   return(v2);
 if (TBmatch(v1,"ins(%s,%s,%t)",&n,&s,&w1))
   return(TBmake("ins(%s,%s,%t)",n,s,concat(w1,v2)));
 error("concat expects a variablelist as first argument",v1);
 return(TBmake("zzconcat"));
}


/******************************** REWRITING ********************************/

term *rewrite(term *t, term *e)   /* rewrites the compact open term t
                                     by using the first equation
                                     in equation list e that matches
                                     with t's top-level functor; but
                                     first all arguments of t are
                                     recursively rewritten in the
                                     same fashion */
{char *n;
 term *l,*v;
 if (TBmatch(t,"t(%s,%t)",&n,&l))
   return(toprewrite(TBmake("t(%s,%t)",n,lrewrite(l,e)),e));
 if (TBmatch(t,"t(%t,%t)",&v,&l))
   return(toprewrite(TBmake("t(%t,%t)",v,lrewrite(l,e)),e));
 error("rewrite expects a term as first argument",t);
 return(TBmake("zzrewrite"));
}

term *lrewrite(term *l, term *e)   /* as rewrite, but used for lists */
{term *t1,*l1;
 if (TBmatch(l,"emt"))
   return(l);
 if (TBmatch(l,"ins(%t,%t)",&t1,&l1))
   return(TBmake("ins(%t,%t)",rewrite(t1,e),lrewrite(l1,e)));
 error("lrewrite expects a termlist as first argument",l);
 return(TBmake("zzlrewrite"));
}

term *toprewrite(term *t, term *e)   /* returns the term obtained by
                                        rewriting the compact open
                                        term t using the first
                                        applicable equation from
                                        equationlist e */
{term *v,*t1,*t2,*e1,*pt1,*pt2,*m;
 if (TBmatch(e,"eme"))
   return(t);
 if (TBmatch(e,"ins(e(%t,%t,%t),%t)",&v,&t1,&t2,&e1)){
   pt1=prep(TBmake("o(%t,%t)",t1,v));
   m=match(t,pt1);
   if (TBmatch(m,"fail"))
```

```
    return(toprewrite(t,e1));
  pt2=prep(TBmake("o(%t,%t)",t2,v));
  return(subst1(m,pt2));
 }
 error("toprewrite expects an equationlist as second argument",e);
 return(TBmake("zztoprewrite"));
}


/********************************* MATCHING *********************************/

term *match(term *t1, term *t2)   /* if (compact open) term t1 is an
                                     instantiation of (compact open)
                                     term t2, then match(t1,t2)
                                     returns the corresponding
                                     substitution, and "fail"
                                     otherwise (t2 is assumed to have
                                     no duplicated variables) */
{char *vn1,*sn1,*vn2,*sn2,*fn1,*fn2;
 term *l1,*l2;
 if (TBmatch(t2,"t(var(%s,%s),%t)",&vn2,&sn2,&l2))
   return(TBmake("ins(p(%s,%t),emp)",vn2,t1));
 else{
   if (TBmatch(t1,"t(var(%s,%s),%t)",&vn1,&sn1,&l1))   /* TBmatch(n1,"var(
                                                          <str>,<str>)") */
     return(TBmake("fail"));
   TBmatch(t1,"t(%s,%t)",&fn1,&l1);
   TBmatch(t2,"t(%s,%t)",&fn2,&l2);
   if ((strcmp(fn1,fn2)==0) && (length(l1)==length(l2)))
     return(lmatch(l1,l2));
   else
     return(TBmake("fail"));
 }
}

term *lmatch(term *l1, term *l2)
{term *t1,*t2,*m1,*m2;
 if (TBmatch(l1,"emt")){
   if (TBmatch(l2,"emt"))
     return(TBmake("emp"));
   else
     error("lmatch expects termlists of equal lengths",l1);
 }
 if (TBmatch(l1,"ins(%t,%t)",&t1,&m1)){
   if (TBmatch(l2,"ins(%t,%t)",&t2,&m2))
     return(sconcat(match(t1,t2),lmatch(m1,m2)));
   else
     error("lmatch expects a termlist as second argument",l2);
 }
 error("lmatch expects a termlist as first argument",l1);
 return(TBmake("zzlmatch"));
}

int length(term *l)   /* returns the length of termlist l */
{term *l0,*l1;
```

```
 if (TBmatch(l,"emt"))
   return(0);
 if (TBmatch(l,"ins(%t,%t)",&l0,&l1)) /* TBmatch(l,"ins(<term>,%t)",&l1)
                                          does not work !? */
   return(length(l1)+1);
 return(-1);
}


/******************************* SUBSTITUTION *******************************/

term *sconcat(term *s1, term *s2)   /* if s1 and s2 are both lists of
                                       replacements, sconcat(s1,s2)
                                       returns their concatenation; if
                                       one of them is "fail", it
                                       returns "fail" */
{term *r,*t1;
 if ((TBmatch(s1,"fail")) || (TBmatch(s2,"fail")))
   return(TBmake("fail"));
 if (TBmatch(s1,"emp"))
   return(s2);
 if (TBmatch(s1,"ins(%t,%t)",&r,&t1))
   return(TBmake("ins(%t,%t)",r,sconcat(t1,s2)));
 error("sconcat expects fail or replacementlists",s1);
 return(TBmake("zzsconcat"));
}


term *subst1(term *s, term *t)   /* performs substitution s in compact term t */
{char *vn,*fn,*ss;
 term *t1,*l1;
 if (TBmatch(t,"t(var(%s,%s),emt)",&vn,&ss))
   return(repl1(vn,ss,s));
 if (TBmatch(t,"t(%s,emt)",&ss))
   return(t);
 if (TBmatch(t,"t(%s,ins(%t,%t))",&fn,&t1,&l1))
   return(TBmake("t(%s,%t)",fn,lsubst1(s,TBmake("ins(%t,%t)",t1,l1))));
 error("subst1 expects a term as second argument",t);
 return(TBmake("zzsubst1"));
}


term *lsubst1(term *s, term *t)   /* performs substitution s in termlist t */
{term *t1,*l1;
 if (TBmatch(t,"emt"))
   return(t);
 if (TBmatch(t,"ins(%t,%t)",&t1,&l1))
   return(TBmake("ins(%t,%t)",subst1(s,t1),lsubst1(s,l1)));
 error("lsubst1 expects a termlist as second argument",t);
 return(TBmake("zzlsubst1"));
}


term *repl1(char *n, char *ss, term *s)   /* if variable n occurs in the
                                             replacement list s, repl1(n,ss,s)
```

```
                                             returns the corresponding
                                             substituted term, otherwise
                                             t(var(n,ss),emt) is returned */
{char *n1;
 term *t,*s1;
 if (TBmatch(s,"emp"))
   return(TBmake("t(var(%s,%s),emt)",n,ss));
 if (TBmatch(s,"ins(p(%s,%t),%t)",&n1,&t,&s1)){
   if (strcmp(n,n1)==0)
     return(t);
   else
     return(repl1(n,ss,s1));
 }
 error("repl1 expects a replacementlist as third argument",s);
 return(TBmake("zzrepl1"));
}
```

C.5 communicate.c

```
#define verbose 0

term *in=NULL; /* take care to protect in */
int finished=TBfalse;

static term *handler(term *t)
{ if (TBmatch(t,"rec-do(%t)",&in))
            return(NULL);
  if (TBmatch(t,"rec-ack-event(<term>)"))
            return(NULL);
  error("A tool receives an unexpected message",t);
  return(NULL);
}

/* static void receive(void)
{
  TBsend(TBmake("snd-event(receive,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives: %t\n",who,in);
  TBreceive();
} */

static void receive_e1(void)
{
  TBsend(TBmake("snd-event(receivee1,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via e1: %t\n",who,in);
  TBreceive();
}

static void receive_e2(void)
{
  TBsend(TBmake("snd-event(receivee2,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via e2: %t\n",who,in);
```

```
  TBreceive();
}

static void receive_r1(void)
{ TBsend(TBmake("snd-event(receiver1,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via r1: %t\n",who,in);
  TBreceive();
}

static void receive_r2(void)
{
  TBsend(TBmake("snd-event(receiver2,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via r2: %t\n",who,in);
  TBreceive();
}

static void receive_c2(void)
{
  TBsend(TBmake("snd-event(receivec2,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via c2: %t\n",who,in);
  TBreceive();
}

static void receive_c1(void)
{
  TBsend(TBmake("snd-event(receivec1,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via c1: %t\n",who,in);
  TBreceive();
}

static void receive_s1(void)
{
  TBsend(TBmake("snd-event(receives1,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via s1: %t\n",who,in);
  TBreceive();
}

static void receive_s2(void)
{
  TBsend(TBmake("snd-event(receives2,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via s2: %t\n",who,in);
  TBreceive();
}

static void receive_u1(void)
{
  TBsend(TBmake("snd-event(receiveu1,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via u1: %t\n",who,in);
```

```
  TBreceive();
}

static void receive_u2(void)
{
  TBsend(TBmake("snd-event(receiveu2,\"\")"));
  TBreceive();
  if (verbose) TBprintf(stderr,"%s receives via u2: %t\n",who,in);
  TBreceive();
}

static void send(term *t)
{
  if (verbose) TBprintf(stderr,"%s sends: %t\n",who,t);
  TBsend(TBmake("snd-event(send,%t)",t));
  TBreceive();
}

static void terminate(void)
{
  TBsend(TBmake("snd-disconnect"));
  if (verbose) TBprintf(stderr,"%s terminates\n",who);
}
```

C.6 representant.c

```
#include "TB.h"
#include "datatypes.h"
#define who "Representant"
#include "communicate.c"
#define NO_REWRITES 20

extern term *rewrite(term *t, term *e);
extern term *unprep1(term *t);

int c0(term *d,char **errormessage)
{ return(ssc(d,errormessage));
}

int c1(term *t, term *sig)
{ return(welltyped(t,sig));
}

int c2(term *t, term *sig)
{ return(correctopenterm(t,sig));
}

term *rewrite_open(term *t, term *eqn)
{ term *result;
  int i;
  result=prep(t);
  for(i=0 ; (i<=NO_REWRITES) ; i++)
    { result=rewrite(result,eqn); }
```

```
  return(unprep1(result));
}

term *rewrite_closed(term *t, term *eqn)

{ term *result1,*result2,*vars;
  result1=rewrite_open(t,eqn);
  if (!TBmatch(result1,"o(%t,%t)",&result2,&vars))
   error("rewrite_closed expects back an openterm from rewrite_open",result1);
  if (!TBmatch(vars,"emv"))
    error("rewriting a closed term resulted in open term",result1);
  return(result2);
}

char *s="This is not an error message, everything should be ok     ";

void main(int argc, char *argv[])

/* main program of representant generator */
{ term *t1=NULL,*d=NULL,*sig=NULL,*eqn=NULL,
    *auxt=NULL;
  char *auxs="This is not an error message, everything should be ok     ";

  TBprotect(&in);
  TBprotect(&d);
  TBprotect(&sig);
  TBprotect(&t1);
  TBprotect(&eqn);
  TBprotect(&auxt);

  TBinit("representant",
         argc, argv,
         handler, NULL);

  RgS:
     receive_r1();
     if (TBmatch(in,"r1(terminate)"))
        goto Finito;
     if (TBmatch(in,"r1(restart)"))
        goto RgS;
     if (TBmatch(in,"r1(d(%t,%t))",&sig,&eqn))
        {TBmatch(in,"r1(%t)",&d) ;
         if (c0(d,&s))
            { send(TBmake("r2(ok)"));
              goto RgS_d; }
            { send(TBmake("r2(notok(%s))",s));
              goto RgS; }}
     error("representant expects terminate, restart or datatype via r",in);
  RgS_d:
     receive_r1();
     if (TBmatch(in,"r1(terminate)"))
        goto Finito;
     if (TBmatch(in,"r1(restart)"))
        goto RgS;
     if (TBmatch(in,"r1(o(<term>,<term>))"))
```

```
        {TBmatch(in,"r1(%t)",&t1);
         if (c2(t1,sig))
            { send(TBmake("r2(%t)",rewrite_open(t1,eqn)));
              goto RgS_d;}
            { send(TBmake("r2(notok(\"Erroneous open term\"))"));
              goto RgS_d;}}
     if (TBmatch(in,"r1(t(%s,%t))",&auxs,&auxt))
        {TBmatch(in,"r1(%t)",&t1);
         if (c1(t1,sig))
            { send(TBmake("r2(%t)",rewrite_closed(t1,eqn)));
      goto RgS_d;}
            { send(TBmake("r2(notok(\"Term is not well typed\"))"));
              goto RgS_d;}}
     error("Representant expects restart, terminate or term via r1",in);
  Finito: terminate() ;
          TBunprotect(&t1);
          TBunprotect(&in);
          TBunprotect(&d);
          TBunprotect(&sig);
          TBunprotect(&eqn);
}
```

C.7 enumerator.c

```
#include "TB.h"
#include "datatypes.h"
#define who "Enumerator"
#include "communicate.c"
#define no_rounds 1   /* maximal instantiation depth */

int *correctopenboolterm(term *t, term *sig);

int existsconsvar(char *n, term *conslist)
{ term *tl;
  char *n1;
  if (TBmatch(conslist,"emf"))
     return(TBfalse);
  if (!TBmatch(conslist,"ins(f(<term>,<term>,%s),%t)",&n1,&tl))
     error("existsconsvar expects functionlist",conslist);
  if (strcmp(n,n1)==0)
     return(TBtrue);
  return(existsconsvar(n,tl));
}

int existscons(term *varlist, term *conslist)
{ term *tl;
  char *n;
  if (TBmatch(varlist,"emv"))
     return(TBfalse);
  if (!TBmatch(varlist,"ins(<term>,%s,%t)",&n,&tl))
     error("existscons expects varlist",varlist);
  if (existsconsvar(n,conslist))
```

```
     return(TBtrue);
  return(existscons(tl,conslist));
}

term *mkterms(term *varlist)
{ term *tl;
  char *x;
  if (TBmatch(varlist,"emv"))
     return(TBmake("emt"));
  if (!TBmatch(varlist,"ins(%s,<str>,%t)",&x,&tl))
     error("mkterms expects varlist",varlist);
  return(TBmake("ins(t(%s,emt),%t)",x,mkterms(tl)));
}

long no=0;
char strng[10];

static void makestring(void)
{ long stringpos;
  long n=no;
  strcpy(strng,"x");
  for(stringpos=1;(n>0);stringpos++)
  {strng[stringpos]=(48+(n % 10));
   n= n / 10;
 };
 strng[stringpos]='\0';
 no++;
}

char *newvar(term *vl, term *sig)
{ term *dummy;
  for(makestring;
         (exists(strng,TBmake("ems"),sig,&dummy) &&
               existvar(strng,vl,&dummy)) ; makestring){;}
  return(strng);
}

term *fresh_rec(term *args, term *vl, term *sig)
{
  term *tail;
  char *hd;
  if (TBmatch(args,"ems"))
     return(TBmake("emv"));
  if (!TBmatch(args,"ins(%s,%t)",&hd,&tail))
     error("Fresh expects sortlist",args);
  return(TBmake("ins(%s,%s,%t)",newvar(vl,sig),hd,fresh_rec(tail,vl,sig)));
}

term *fresh(term *args, term *vl, term *sig)
{ no=0;
  return(fresh_rec(args,vl,sig));
}


term *Expand1(char *s, char *x,
```

```
              term *vl1, term *flist, term *vl2, term *insts, term *t,
              term *sig, term *next)
{ term *args, *fns, *freshvars, *freshterm, *vl;
  char *n,*targ;

  if (TBmatch(flist,"emf"))
     return(next);
  if (!TBmatch(flist,"ins(f(%s,%t,%s),%t)",&n,&args,&targ,&fns))
     error("Expand1 expects functionlist",flist);
  if (strcmp(s,targ)==0)
     { vl=conc_var(vl1,vl2);
       freshvars=fresh(args,vl,sig);
       freshterm=TBmake("t(%s,%t)",n,mkterms(freshvars));
       return(Expand1(s,x,vl1,fns,vl2,insts,t,sig,
          TBmake("ins(%t,%t,%t,%t)",
               conc_var(vl,freshvars),
               lrepl(x,freshterm,insts),
               repl(x,freshterm,t),
               next)));
     }
  return(Expand1(s,x,vl1,fns,vl2,insts,t,sig,next));
}

term *Expand(term *vl1, term *vl2, term* insts, term *t, term *sig, term *nextround)
{ term *tail, *constr;
  char *name, *sort;
  if (TBmatch(vl2,"emv"))
        return(nextround);
  if (!TBmatch(vl2,"ins(%s,%s,%t)",&name,&sort,&tail))
     error("Expand expects variablelist",vl2);
  if (!TBmatch(sig,"s(<term>,%t,<term>)",&constr))
     error("Expand expects signature",sig);
  return(Expand1(sort,name,vl1,constr,tail,insts,t,sig,
         Expand(TBmake("ins(%s,%s,%t)",name,sort,vl1),tail,insts,t,sig,nextround)));
}

void main(int argc, char *argv[])
{ term *t1=NULL, *t2=NULL, *t3=NULL, *t4=NULL,
       *t5=NULL, *t6=NULL, *c=NULL, *sig=NULL,
       *itl=NULL, *nextround=NULL, *results=NULL;
  char *s="This is not an error message, everything should be ok     ";
  int n=0;
  TBinit("enumerator",
          argc, argv,
          handler, NULL);

  TBprotect(&in);

  TBprotect(&t1);
  TBprotect(&t2);
  TBprotect(&t3);
  TBprotect(&t4);
  TBprotect(&t5);
  TBprotect(&t6);
  TBprotect(&c);
```

```
TBprotect(&sig);
TBprotect(&itl);
TBprotect(&nextround);
TBprotect(&results);

EnI0: receive_e1();
     if (TBmatch(in,"e1(d(%t,%t))",&sig,&t1))
        { send(TBmake("r1(d(%t,%t))",sig,t1));
          goto EnI1;}
     if (TBmatch(in,"e1(restart)"))
        { goto EnI0; }
     if (TBmatch(in,"e1(terminate)"))
        { send(TBmake("r1(terminate)"));
          goto Finito; }
     error("Enumerate expects a datatype or terminate via e",in);
EnI1: receive_r2();
     if (TBmatch(in,"r2(ok)"))
        { send(TBmake("e2(ok)"));
          goto EnI2; }
     if (TBmatch(in,"r2(notok(%s))",&s))
        { send(TBmake("e2(notok(%s))",s));
          goto EnI0; }
     error("Enumerate expects ok or notok via r2",in);
EnI2: receive_e1();
     if (TBmatch(in,"e1(restart)"))
        { send(TBmake("r1(restart)"));
          goto EnI0; }
     if (TBmatch(in,"e1(terminate)"))
        { send(TBmake("r1(terminate)"));
          goto Finito; }
     if (TBmatch(in,"e1(%t)",&c))
        { if (correctopenboolterm(c,sig))
          { if (!TBmatch(c,"o(%t,%t)",&t2,&t1))
               error("Enumerator expects openterm",c);
            itl=TBmake("ins(%t,%t,%t,emit)",t1,mkterms(t1),t2);
            results=TBmake("emtll");
            n=no_rounds;
            nextround=TBmake("emit");
            goto EnI3; }
          send(TBmake(
               "e2(notok(\"Enumerate: openterm not Bool or Welltyped\"))",s));
          goto EnI2 ; }
     error("Expect stop, terminate or open term in enumerator",in);
EnI3 :
      if (TBmatch(itl,"ins(%t,<term>,%t,<term>)",&t1,&t2))
         { send(TBmake("r1(o(%t,%t))",t2,t1)) ;
           goto EnI4; }
      if (TBmatch(itl,"emit"))
         { if (TBmatch(nextround,"emit"))
           { send(TBmake("e2(%t)",results));
             goto EnI2 ;}
         if (n==0)
            { send(TBmake("e2(notok(\"Enumerator: Too many terms\"))"));
              goto EnI2; }
         n=n-1;
```

```
          itl=nextround;
          nextround=TBmake("emit");
          goto EnI3; }
       error("Enumerate: itl should be an itlist",itl);
 EnI4 :
       receive_r2();
       if (TBmatch(in,"r2(notok(%t))",&s))
          { send(TBmake("e2(notok(%s))",s));
            goto EnI2; }
       if (!TBmatch(in,"r2(o(%t,<term>))",&t1))
          error("Enumerator expects openterm via r2",in);
       if (TBmatch(t1,"t(\"F\",emt)"))
          { if(!TBmatch(itl,"ins(<term>,<term>,<term>,%t)",&t2))
              error("Enumerate: itl should not be empty (2)",itl);
            itl=t2;
            goto EnI3; }
       if (!TBmatch(itl,"ins(<term>,%t,<term>,%t)",&t3,&t2))
            error("Enumerate: itl should not be empty (1)",itl);
       if ((TBmatch(t1,"t(\"T\",emt)")) && closedl(t3,sig))
          { itl=t2;
            results=TBmake("ins(%t,%t)",t3,results);
            goto EnI3; }
       if (closedl(t3,sig))
          { send(TBmake("e2(notok(\"Enumerator:\
        Closed term of sort Bool does not reduce to true or false\"))"));
            goto EnI2; }
       if (!TBmatch(sig,"s(<term>,%t,<term>)",&t6))
          error("Enumerator expects signature",sig);
       if (!TBmatch(itl,"ins(%t,<term>,<term>,<term>)",&t2))
          error("Enumerate: itl should not be empty (3)",itl);
       if (existscons(t2,t6))
          {
            if (!TBmatch(itl,"ins(%t,%t,<term>,%t)",&t3,&t4,&t5))
               error("Enumerate: itl should not be empty (3)",itl);
            itl=t5;
            nextround=Expand(TBmake("emv"),t3,t4,t1,sig,nextround);
            goto EnI3;
          }
       send(TBmake("e2(notok(\"Enumerator: Sort with infinite size\"))"));
       goto EnI2;
 Finito: terminate();
         TBunprotect(&in);
         TBunprotect(&t1);
         TBunprotect(&t2);
         TBunprotect(&t3);
         TBunprotect(&t4);
         TBunprotect(&t5);
         TBunprotect(&t6);
         TBunprotect(&c);
         TBunprotect(&sig);
         TBunprotect(&itl);
         TBunprotect(&nextround);
         TBunprotect(&results);
}
```

C.8 stepper.c

```
#include "TB.h"
#include "datatypes.h"
#include <limits.h>
#define who "Stepper"
#include "communicate.c"


char *s="This is not an error message, everything should be ok     ";
term *d=NULL;

void main(int argc, char *argv[])
{ term *adt=NULL,*proc=NULL,*spec=NULL, *termlist=NULL,
       *sumlist=NULL,*summand=NULL,*tl=NULL,*results=NULL,*tll=NULL;
  term *t1=NULL,*t2=NULL,*t3=NULL, *t4=NULL, *t5=NULL,
       *t6=NULL,*t7=NULL; /* temporal stores */
  char *reason="Nothing is wrong, no really, nothing is wrong at all ";
  char *action="                                                  ";

  TBprotect(&in);
  TBprotect(&d);

  TBprotect(&tll);
  TBprotect(&results);
  TBprotect(&adt);
  TBprotect(&proc);
  TBprotect(&spec);
  TBprotect(&termlist);
  TBprotect(&sumlist);
  TBprotect(&summand);
  TBprotect(&tl);
  TBprotect(&t1);
  TBprotect(&t2);
  TBprotect(&t3);
  TBprotect(&t4);
  TBprotect(&t5);
  TBprotect(&t6);
  TBprotect(&t7);

  TBinit("stepper",
          argc, argv,
          handler, NULL);

  StI0:
       receive_c1();
       if (TBmatch(in,"c1(spec(%t,%t))",&adt,&proc))
          {TBmatch(in,"c1(%t)",&spec);
           send(TBmake("e1(%t)",adt));
           receive_e2();
           if (TBmatch(in,"e2(notok(%s))",&reason))
              {send(TBmake("c2(notok(%s))",reason));
               goto StI0;}
           if (TBmatch(in,"e2(ok)"))
              {if (ssc_spec(spec,&reason))
                  {send(TBmake("c2(ok)"));
```

```
                goto StI1;}
            send(TBmake("c2(notok(%s))",reason));
            goto StI0;
           }
        error("stepper expects ok or notok via e2",in);
       }
     if (TBmatch(in,"c1(restart)"))
        { goto StI0; }
     if (TBmatch(in,"c1(terminate)"))
        {send(TBmake("e1(terminate)"));
         goto Finito; }
     error("Stepper expects spec, restart or terminate via c1",in);
StI1:
     receive_c1();
     if (TBmatch(in,"c1(restart)"))
        {send(TBmake("e1(restart)"));
         goto StI0;}
     if (TBmatch(in,"c1(terminate)"))
        {send(TBmake("e1(terminate)"));
         goto Finito;}
     if (TBmatch(in,"c1(%t)",&termlist))
        { if (!correct(termlist,proc,adt,&reason))
            { send(TBmake("c2(notok(%s))",reason));
              goto StI1; }
         if (!TBmatch(proc,"proc(<term>,%t)",&sumlist))
            error("Stepper expects process",proc);
         results=TBmake("emsl");
         goto StI2;}
     error("Stepper expects restart, terminate or termlist via c1",in);
StI2:
     if (TBmatch(sumlist,"eml"))
        {send(TBmake("c2(%t)",results));
         goto StI1;}
     if (TBmatch(sumlist,"ins(%t,%t)",&summand,&tl))
        {if (!TBmatch(summand,"smd(%t,<str>,<term>,<term>,%t)",&t1,&t2))
            error("Stepper expects summand",summand);
         if (!TBmatch(proc,"proc(%t,<term>)",&t3))
            error("Stepper expects proc", proc);
         send(TBmake("e1(o(%t,%t))",substituteterm(termlist,t2,t3),t1));
         receive_e2();
         if (TBmatch(in,"e2(notok(%s))",&reason))
            { send(TBmake("c2(notok(%s))",reason));
              goto StI1;}
         if (TBmatch(in,"e2(%t)",&tll))
            { goto StI3;}
         error("Stepper expects message or Termlistlist via e2",in);}
     error("Stepper expects sumlist",sumlist);

StI3:
     if (TBmatch(tll,"emtll"))
        { sumlist=tl;
          goto StI2; }
     if (TBmatch(tll,"ins(%t,%t)",&t1,&t2))
        { if (!TBmatch(sumlist,"ins(%t,<term>)",&t3))
             error("Stepper expects Sumlist",sumlist);
```

```
            if (!TBmatch(t3,"smd(%t,%s,%t,%t,<term>)",&t5,&action,&t6,&t7))
               error("Stepper expects Summand",t3);
    if (!TBmatch(proc,"proc(%t,<term>)",&t4))
               error("Stepper expects ProcSpec",proc);
            results=TBmake("ins(st(%s,%t,%t),%t)",
                action,
                lsubstituteterm(termlist,lsubstituteterm(t1,t6,t5),t4),
                lnsubstituteterm(termlist,lnsubstituteterm(t1,t7,t5),t4),
                results);
            tll=t2;
            goto StI3; }
       error("Stepper expects Termlistlist",tll);

  Finito: terminate() ;
          TBunprotect(&results);
          TBunprotect(&adt);
          TBunprotect(&proc);
          TBunprotect(&spec);
          TBunprotect(&termlist);
          TBunprotect(&sumlist);
          TBunprotect(&summand);
          TBunprotect(&tl);
          TBunprotect(&t1);
          TBunprotect(&t2);
          TBunprotect(&t3);
          TBunprotect(&t4);
          TBunprotect(&t5);
          TBunprotect(&t6);
          TBunprotect(&t7);
          TBunprotect(&in);
          TBunprotect(&tll);
}
```

C.9 instantiator.c

```
#include "TB.h"
#include "datatypes.h"
#include <limits.h>
#define who "Instantiator"
#include "communicate.c"
#define MAX_TS_SIZE 1000


void main(int argc, char *argv[]) /* main program of instantiator */
{ term *spec=NULL,*adt=NULL,*proc=NULL,
       *s=NULL,*g=NULL,*b=NULL,
       *tl=NULL,*stpl=NULL,*hdstpl=NULL,*tlstpl=NULL,
       *hdg=NULL,*tlg=NULL,
       *actterms=NULL,
       *t1=NULL,*t2=NULL,*u=NULL;
  char *m="This is the initial value                 ",
       *mm="This is the initial value                ",
       *mess14="This is the initial value            ",
       *mess15="Transition system too large          ",
```

```
     *act="This is the initial value               ";
 long len = 0;

 TBprotect(&in);

 TBprotect(&spec);
 TBprotect(&adt);
 TBprotect(&proc);
 TBprotect(&s);
 TBprotect(&g);
 TBprotect(&b);
 TBprotect(&tl);
 TBprotect(&stpl);
 TBprotect(&hdstpl);
 TBprotect(&tlstpl);
 TBprotect(&hdg);
 TBprotect(&tlg);
 TBprotect(&actterms);
 TBprotect(&t1);
 TBprotect(&t2);
 TBprotect(&u);

 TBinit("instantiator",
         argc, argv,
         handler, NULL);

InI0:
 receive_u1();
 if (TBmatch(in,"u1(spec(%t,%t))",&adt,&proc)){
   TBmatch(in,"u1(%t)",&spec);
   send(TBmake("c1(%t)",spec));
   send(TBmake("s1(%t)",adt));
   goto InI1;
 }
 if (TBmatch(in,"u1(restart)")){
   goto InI0;
 }
 if (TBmatch(in,"u1(terminate)")){
   send(TBmake("c1(terminate)"));
   send(TBmake("s1(terminate)"));
   goto Finito;
 }
 error("Instantiator expects spec, restart or terminate via u1",in);
InI1:
 receive_c2();
 if (TBmatch(in,"c2(ok)")){
   receive_s2();
   if (TBmatch(in,"s2(ok)")){
     { send(TBmake("u2(ok)"));
       goto InI2;}
   }
   if (TBmatch(in,"s2(notok(%s))",&m)){
     send(TBmake("u2(notok(%s))",m));
     send(TBmake("c1(restart)"));
     goto InI0;
```

```
    }
    error("Instantiator expects ok or notok(...) via s2",in);
  }
  if (TBmatch(in,"c2(notok(%s))",&m)){
    receive_s2();
    if (TBmatch(in,"s2(ok)")){
      send(TBmake("s1(restart)"));
      send(TBmake("u2(notok(%s))",m));
      goto InI0;
    }
    if (TBmatch(in,"s2(notok(%s))",&mm)){
      send(TBmake("u2(notok(%s))",strcat(m,mm)));
      goto InI0;
    }
    error("Instantiator expects ok or notok(...) via s2",in);
  }
  error("Instantiator expects ok or notok(...) via c2",in);
InI2:
  receive_u1();
  if (TBmatch(in,"u1(restart)")){
    send(TBmake("c1(restart)"));
    send(TBmake("s1(restart)"));
    goto InI0;
  }
  if (TBmatch(in,"u1(terminate)")){
    send(TBmake("c1(terminate)"));
    send(TBmake("s1(terminate)"));
    goto Finito;
  }
  if (TBmatch(in,"u1(%t)",&s)){
    if (correct(s,proc,adt,&mess14)){
      g=TBmake("ins(%t,emtll)",s);
      b=TBmake("emtll");
      tl=TBmake("emts");
      len=0;
      goto InI3;
    }
    else{
      send(TBmake("u2(notok(%s))",mess14));
      goto InI2;
    }
  }
  error("Instantiator expects a termlist, restart or terminate via u1",in);
InI3:
  if (isemptytll(g)){
    send(TBmake("u2(ts(%t,%t,%t))",adt,s,tl));
    send(TBmake("c1(restart)"));
    send(TBmake("s1(restart)"));
    goto InI0; }
  if (len >= MAX_TS_SIZE){
    send(TBmake("u2(notok(%s))",mess15));
    send(TBmake("c1(restart)"));
    send(TBmake("s1(restart)"));
    goto InI0; }
  TBmatch(g,"ins(%t,%t)",&hdg,&tlg);
```

```
  send(TBmake("c1(%t)",hdg));
  receive_c2();
  if (TBmatch(in,"c2(notok(%s))",&m)){
send(TBmake("u2(notok(%s))",m));
send(TBmake("c1(restart)"));
send(TBmake("s1(restart)"));
goto InI0; }
  /* it must be a steplist */
  TBmatch(in,"c2(%t)",&stpl);
  g=tlg;
  b=TBmake("ins(%t,%t)",hdg,b);
  goto InI4;
 InI4:
  if (TBmatch(stpl,"emsl")){
    goto InI3;
  };
  if (TBmatch(stpl,"ins(%t,%t)",&hdstpl,&tlstpl)){
    if (TBmatch(hdstpl,"st(%s,%t,terminated)",&act,&actterms)){
      stpl=tlstpl;
      tl=TBmake("ins(tr(%t,%s,%t,terminated),%t)",hdg,act,actterms,tl);
      len++;
      goto InI4;
    }
    if (TBmatch(hdstpl,"st(%s,%t,i(%t))",&act,&actterms,&t1)){
      t2=TBmake("emt");
      goto InI5;
    }
    error("Instantiator expects non-empty termlist to start with
               st(..., ..., terminate) or st(..., ..., i(...))",hdstpl);
  };
  error("Instantiator expected steplist via c2",in);
 InI5:
  if (isemptytl(t1)){                 /* t2 contains normalised state */
    if (!(iseltll(t2,g) || iseltll(t2,b))){
      g=TBmake("ins(%t,%t)",t2,g);
    };
    stpl=tlstpl;
    tl=TBmake("ins(tr(%t,%s,%t,i(%t)),%t)",hdg,act,actterms,t2,tl);
    len++;
    goto InI4;
  }
  else{                               /* t1 non-empty */
    send(TBmake("s1(%t)",toe(t1)));
    receive_s2();
    if (TBmatch(in,"s2(notok(%s))",&m)){
      send(TBmake("u2(notok(%s))",m));
      send(TBmake("c1(restart)"));
      send(TBmake("s1(restart)"));
      goto InI0;
    }
    if (TBmatch(in,"s2(%t)",&u)){
      t1=untoe(t1);
      t2=TBmake("ins(%t,%t)",u,t2);
      goto InI5;
    }
```

```
    error("Instantiator expects notok(...) or a term via s2",in);
  }

 Finito:
  terminate() ;
  TBunprotect(&spec);
  TBunprotect(&adt);
  TBunprotect(&proc);
  TBunprotect(&s);
  TBunprotect(&g);
  TBunprotect(&b);
  TBunprotect(&tl);
  TBunprotect(&stpl);
  TBunprotect(&hdstpl);
  TBunprotect(&tlstpl);
  TBunprotect(&hdg);
  TBunprotect(&tlg);
  TBunprotect(&actterms);
  TBunprotect(&t1);
  TBunprotect(&t2);
  TBunprotect(&u);

}
```

References

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

[2] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings Concur'94*, Uppsala, Sweden, Lecture Notes in Computer Science no. 836, pages 401-416, Springer Verlag, 1994.

[3] G.A. Blaauw. *Digital System Implementation.* Prentice-Hall. 1976.

[4] D.J.B. Bosscher and A. Ponse. Translating a process algebra with symbolic data values to linear format. In U.H. Engberg, K.G. Larsen and A. Skou, editors, Proceedings of the Workshop on Tools and Algorithms for the Construction and the Analysis of Systems (TACAS), BRICS Notes Series NS-95-2, pages 119-130, Aarhus, Denmark, 1995.

[5] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[6] R. De Simone and V. Roy. Auto/Autograph. In E.M. Clarke and R.P. Kurshan, editors, *Proceedings of the 2nd International Conference on Computer-Aided Verification,* New Brunswick, NJ, USA, volume 531 of *Lecture Notes in Computer Science*, pages 65–75. Springer-Verlag, 1991.

[7] A.J.P.M. Engel, L.M.G. Feijs, J.F. Groote, J.C. van de Pol and J. Springintveld. Specification, design and simulation of services and protocols for a PDA using the infra red medium (confidential). Technical report RWB-510-re-95012, Dept. of Information and Software Technology, Philips, Eindhoven, 1995.

[8] J.-C. Fernandez. ALDEBARAN: un système de vérification par réduction de processus communicants. PhD. Thesis, Univ. de Grenoble, 1988.

[9] J.F. Groote. Implementations of events in LOTOS-specifications. Technical Report 009/88EN, Philips CFT, Eindhoven, 1988.

[10] J.F. Groote, L. Helmink, R.P. Jagt, Y.M. Lau, and J. Springintveld. Analysis and PSF model of the project 50 application layer for TV and VCR. Technical Report RWB-508-re-94081. Department of Information and Software Technology. Philips Research, Eindhoven, The Netherlands, 1994. *Confidential.*

[11] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In Andrews et al. *Proceedings of the International Workshop on Semantics of Specification Languages*. Workshops in Computing, pages 231–250. Springer Verlag, 1994.

[12] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds, *Algebra of Communicating Processes*, Workshops in Computing, pp. 26–62, 1994.

[13] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. Technical Report 142, Logic Group Preprint Series, Utrecht University, 1995. An early version appeared in *Models and Proofs, proceedings of AMAST workshop on Real-Time systems and Opération Inter-PRC "Modèles et Preuves"*, Bordeaux, 1995.

[14] J.A. Hillebrand and H.P. Korver. A well-formedness checker for μCRL. Technical Report P9501, University of Amsterdam, Programming Research Group, 1995.

[15] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. AT&T Technical Journal 69(1):44–59, 1990.

[16] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

Index