Building a simulator in the mu CRL toolbox -- A case-study in modern software engineering

H.P. Korver

Computer Science/Department of Software Technology

CS-R9632 1996

Report CS-R9632
ISSN 0169-118X

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

Building a Simulator in the μCRL Toolbox

A case-study in modern software engineering

Henri Korver

CWI

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

henri@cwi.nl

Abstract

In this paper we report on the current status of our development of a simulator tool in the μCRL Toolbox. The tool is built from (reusable) software components which are implemented in different languages like C, TCL/TK and ASF+SDF. The components communicate with each other via the ToolBus which is a new software architecture for building large, heterogeneous and distributed systems.

AMS Subject Classification (1991): 68N15, 68Q60.
CR Subject Classification (1991): D.2.1, D.2.2, D.2.6.
Keywords & Phrases: μCRL, simulator, ToolBus.

1 Introduction

It is a major problem to keep large software projects manageable. Typically, such projects end up in a situation where there is a lot of code, a lot of inconsistent documentation and many experts who have knowledge about parts of the software. In such situation, it is rather hard to adapt the software and rather expensive to train new experts. This makes the maintenance of software a costly operation.

At the CWI where we are developing a toolbox for μCRL we are facing similar problems. The language μCRL [8, 9] is a process-algebraic language for specifying and analysing distributed systems. The development of a toolkit for this language, e.g. consisting of editors, well-formedness checkers, simulators, bisimulation checkers, modal checkers, etc, will take several years and many different people will be involved.

We decided to build the software of such toolbox by using modern techniques that are advocated by the formal-method community (universities, research institutes and research departments of companies) of which we are part ourselves. Our approach includes the following topics:

Formal techniques Communicating systems are often rather complex. To control their complexity it may be useful to specify their interactions with mathematical precision.

Object orientation In this approach, large monolithic systems are decomposed into a number of cooperating components such that the modularity and flexibility of the system can be improved.

Heterogeneity By connecting existing tools (developed by others using different implementation languages and run on different hardware platforms) we can re-use their implementation and built new systems at lower costs.

This paper can be considered a case-study in building large, heterogeneous and distributed software systems. In particular, we describe how we constructed a simulator in the μCRL Toolbox by applying the techniques mentioned above. This is a tool which can be used for interactively exploring the state space of a μCRL specification. The tool is provided with a user interface written in TCL/TK.

We have specified most parts of the simulator in μCRL. Throughout the paper, μCRL plays a double role. On the one hand, it is the target language of our toolbox. E.g. the simulator tool explores μCRL specifications. On the other hand, μCRL is used for specifying the software of such tools. Our ultimate goal is that the simulator can simulate its own μCRL specification.

The simulator is built from eight independent components. In fact, we only built two components ourselves. The other six components already existed (see [6]) and could be reused. In particular, several components were developed by different people in different implementation languages. Four of the components of the simulator are implemented in C. The user-interface component is implemented in TCL/TK and the component which is used for linearising μCRL specifications is implemented in ASF+SDF [12].

The components of the simulator communicate via the ToolBus [2, 1, 13]. This new software architecture allows for connecting tools (software components) by exchanging terms. This is convenient because it is very close to the μCRL specification style, where processes also communicate terms. In fact μCRL is well suited for specifying the interfaces between the components in the Toolbox.

The paper is organised as follows. In Section 2, we present the global structure of the simulator tool, i.e. its component configuration. In the following two sections we describe the new components that are added to the Toolbox for constructing the simulator. In particular, the User Interface component is treated in Section 3 and the Simulator component is treated in Section 4. In Section 5, it is discussed how the components are connected by using the ToolBus. Finally, in Section 6, we draw the conclusions of our work.

Acknowledgements

David Griffioen for teaching me some fine details about C. Pieter Olivier and Steven Klusener are thanked for explaining other fine details about the ToolBus and TCL/TK. Doeko Bosscher and Jan Friso Groote are thanked for adapting their Toolbox components whenever needed. Finally, Annius Groenink is thanked for handing over a handy trick concerning screen dumps.

2 The general structure

The simulator consists of eight components as is depicted in Figure 1. The components that are dashed are not yet fully integrated in the Toolbox but will be in the near future. The reason for this is that the SSC tool [11] and the Linearisator [5] are implemented in ASF+SDF for which currently no tool adapter is available. An adapter is needed for each tool to adapt it to the common data representation and message protocols of the ToolBus. For the moment only tools that are written in C, Perl, Python or TCL/TK can communicate via the ToolBus.

The User Interface (UI) allows for exploring the state space of a μCRL process by clicking in a menu display with a mouse button. In doing so it relies on the actual Simulator component (Sim) which manages the actual exploration, e.g. computation of the menu and the next state. In turn the Simulator depends on two existing components: the Stepper (St) and the Representant Generator (RG) (see Appendix B). These components are extensively described in [6]. Given a state s, the Stepper returns a list of its outgoing action-state pairs. In this list, also called the steplist, the states and the actions are in internal ToolBus-format. The simulator uses the RG component for reducing data terms to their normal form. For instance, the data terms in the steplist generated by the Stepper are not always in normal form. Finally there is an Enumerator which returns valid instantiations of a given boolean term. This component is used by the Stepper to enumerate all data terms in the sum construct of μCRL that satisfy a given conditional. The Enumerator also uses the RG component.

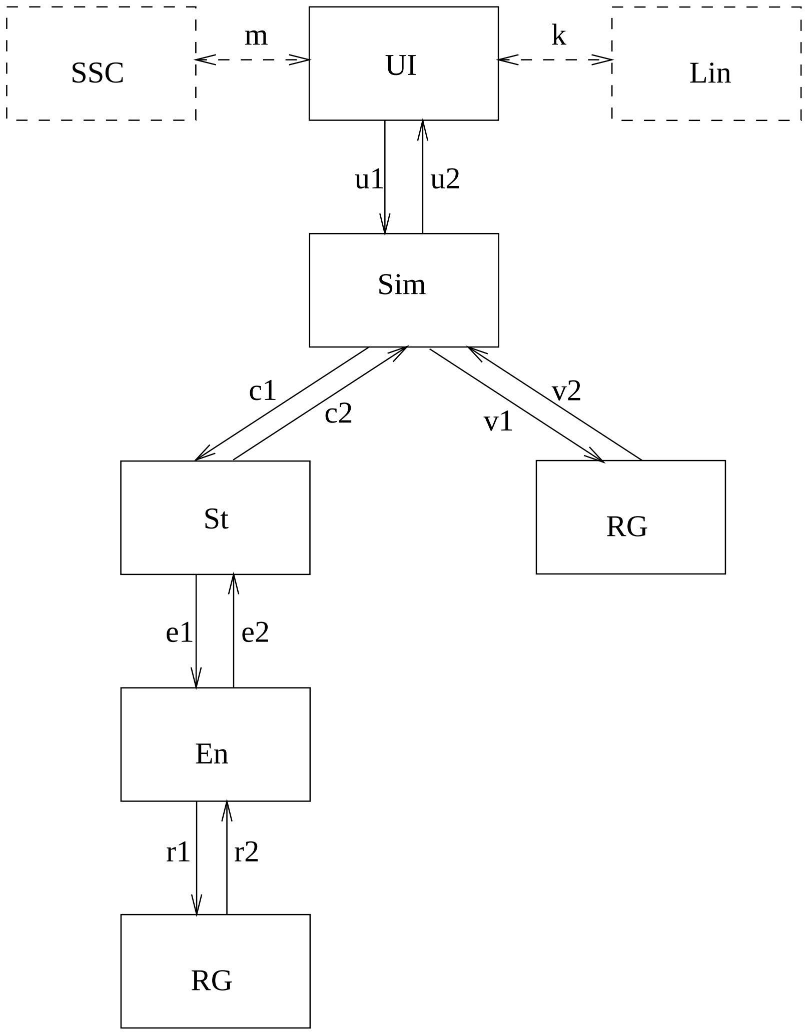

Figure 1: The components of the simulator.

It is assumed that the μCRL specifications are in a linear form. This allows for simplifications in the design of the simulator. In order to make the toolset available for arbitrary μCRL specifications we use the Linearisation tool developed by Doeko Bosscher [5]. Before we feed a μCRL specification to the simulator tool, we first check its statical correctness by using the SSC tool and then we translate it to a linear format with the Linearisation tool. For the moment these two operations are performed outside the ToolBus environment by starting the tools by hand from the Unix-shell.

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 [4]. 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 Sim component and the User Interface the implementations should provide the functionality prescribed by the specification. This can formally be described by:

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

  $$SimS_0 = \tau_{\{c_{c1},c_{c2},c_{v1},c_{v2}\}}\partial_{\{r_{c1},r_{c2},s_{c1},s_{c2},r_{v1},r_{v2},s_{v1},s_{v2}\}}(SimI_0 \parallel StS_0 \parallel \rho_{\{r1\to v1, r2\to v2\}}(RgS_0))$$

  The $SimS_0$ specification is given in Section 4.2. The μCRL code of $SimI_0$ is given in Section 3.3. The specifications of StS_0 and RgS_0 are given in Appendix B.

- Likewise, the specification of the Simulator and the implementation of the User Interface form the specification of the User Interface (UI).

  $$UI_0 = \tau_{\{c_{e1},c_{e2}\}}\partial_{\{r_{e1},r_{e2},s_{e1},s_{e2}\}}(UII_0 \parallel SimS_0)$$

  Both the μCRL code for the specification UI_0 and the code for its implementation UII_0 are not yet available. The $SimS_0$ specification is given in Section 4.2.

By techniques put forward in [10] we can prove and computer-check these equations. However this is not in the scope of this paper.

3 The User Interface (UI)

3.1 Informal description

In Figure 2 a depiction is given of the user interface of the simulator.

The session concerns a simulation of two parallel queues with unbounded capacity. The first queue Q reads on channel 1 and sends its output via an internal channel to the second queue R. The latter queue reads on the internal channel and sends its elements to channel 2. The μCRL specification of this queue is given in Appendix A. Note that the top-level process in this specification is called *Top*. In this way the simulator can recognise the initial state of the specified process. The specification has been type checked by the SSC tool.

There are two data elements $d1$ and $d2$, and both queue can hold arbitrary many copies of these two elements. In the situation of Figure 2 the first queue Q contains $d1$ and the other queue R contains $d2$. In this situation queue Q can read both elements $d1$ and $d2$ from the external world as it has unbounded capacity. Moreover, it can communicate its contence $d1$ to the second queue by the internal action `tau`. The second queue R can send its contence $d2$ to the outside world. The four possible actions that can be performed by the parallel queues Q and R are listed in the menu display. One can select an action by moving the mouse pointer to the desired item and clicking the left mouse button.

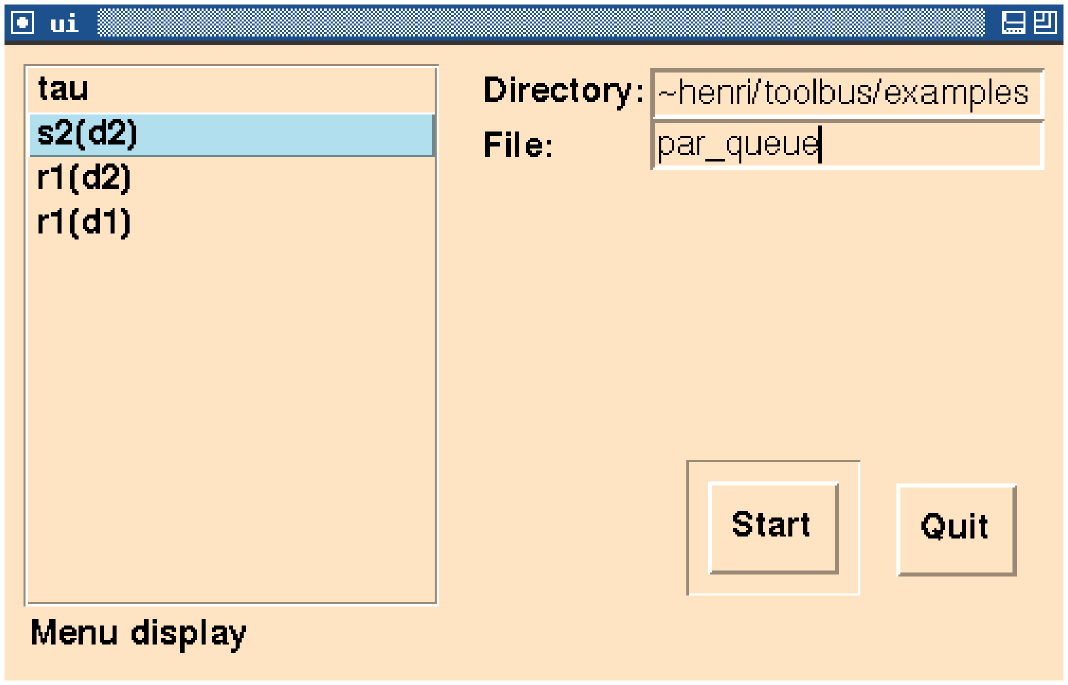

Figure 2: User interface of the simulator.

We have translated the μCRL specification given in Appendix A to a linear format by using the Linearisator. The file `par_queue` in directory `~henri/toolbus/examples` contains the output of this tool. This file serves as the input of the simulator.

The functionality of the buttons is simple: with the `Start` button one can (re)start the simulation and with the `Quit` button one can terminate the simulation session.

3.2 Specification

We did not yet provide a μCRL specification of the user interface of the simulator. This is a pity because such specification would exactly determine the external behaviour of the simulator as a whole which, in principle, is the only thing people are interested in. Still we think we have two good reasons why we have postponed this work. First, the UI is a top-level component and here not used as a building block for constructing other tools. For the other components it is more urgent to be provided with a formal specification as the software engineer must have detailed knowledge for knowing how to connect them.

Second, it is sometimes difficult deciding up to which abstraction level you want to specify the behaviour of the UI as many things can happen on your screen. For example, if you resize the window in Figure 2 the size of the display menu will change but the size of the `Directory` and `File` fields will remain same. Or another example: When the size of an action exceeds the boundary of the menu display you can scroll it back by using the middle mouse-button. Should we record all this in our μCRL specification? If we want to document all such features, the specification may become rather long-winded.

3.3 Implementation

For the same reasons as given above, we did not yet specify the implementation of the UI component.

3.4 Realisation

Without using formal specification techniques at forehand, we directly realised the UI component in TCL/TK which is a high-level language for programming user interfaces under X Windows.

4 The Simulator

4.1 Informal description

In order to determine the steps that are possible from a given state the simulator needs a process specification first. Then it asks for the initial state of this process specification. For this state the simulator returns its menu. This is a list of actions that are possible from this state. Then it receives a choice from the menu, e.g. a natural number corresponding to particular action that was selected by the user. Subsequently, it computes the next state of this action and returns the menu of this state. Then, the procedure just described is repeated until the simulation successfully terminates or evolves into a deadlock. It is also possible that the simulated process has an infinite behaviour. In that case the simulator never terminates unless you force it to stop. In other words: the Sim component ends its execution when it receives a *terminate* command.

4.2 Specification of the simulator

Below the external behaviour of the *Sim* component is specified in μCRL. In state $SimS_0$ the simulator receives a μCRL specification *spec* from the user interface. By condition c_6 it is required that the specification must be statically semantically correct. If this is not the case $message_9$ is reported which indicates what is wrong.

In state $SimS_1$ the initial state of the specification is received. Then in $SimS_2$, condition c_{sim0} checks whether the state *state* is correct with respect to the specification. If this is the case the menu of this state is sent to the UI. Otherwise, an error message is reported. The menu function returns a string containing the outgoing actions of *state*. The actions are separated by a newline and every menu string is ended with a newline. A newline is represented by the constant *NEWLINE* in the μCRL code below. It corresponds with the decimal number 10 in ASCII notation. For example the menu in The actual μCRL specification of the *menu* function is based on the the *steplist* function which returns a list of action-state pairs. All items in this list are represented in internal ToolBus format (see Appendix B and [6]). The μCRL specification of this function can also be found in the just mentioned paper. In state $SimS_3$ a choice from the menu is received. A correct choice should be a natural number in the interval $1, \dots, n$ where n it the total number of items in the menu. If this is not the case the message *wrongchoice* is reported.

In $SimS_4$, given *choice* the simulator extracts the corresponding next state from the steplist. If the next state is a *termination* state, this is reported to the user interface. Otherwise the simulator jumps back to state $SimS_2$ passing over the new state and the just described procedure is repeated.

func $menu : Termlist \times Spec \to String$
$menu' : Steplist \to String$
$length : Steplist \to Nat$
$nextstate : Steplist \times Nat \to NTermlist$
$next : Steplist \times Nat \times Nat \to NTermlist$
$wrongchoice, termination : \to Status$

$ok : String \to Status$
$between : Nat \times Nat \times Nat \to Bool$
$peel : Term \to String$
$peel : Termlist \to String$
$compose : String \times String \times String \to String$
$compose : String \times String \times String \times String \to String$
$compose : String \times String \times String \times String \times String \to String$
$compose : String \times String \times String \times String \times String \times String \to String$
$\texttt{""}, \texttt{"("}, \texttt{")"}, NEWLINE :\to String$
$\texttt{","} :\to String$
$if : Bool \times NTermlist \times NTermlist \to NTermlist$
$message_{sims0} : Spec \to String$
$message_{sims2} : Termlist \times Spec \to String$
$c_{sims0} : Spec \to Bool$
$c_{sims2} : Termlist \times Spec \to Bool$
$c'_{sims2} : Termlist \times Spec \to Bool$
$c' : Steplist \times Datatype \to Bool$
$c'' : Step \times Datatype \to Bool$
$c''' : Termlist \times Datatype \to Bool$

var $x, y, z : Nat$
$spec : Spec$
$d : Datatype$
$sl : Steplist$
$st : Step$
$string, s, s1, s2, s3, s4, s5 : String$
$t : Term$
$L, state : Termlist$
$nstate : NTermlist$

rew $compose(s1, s2, s3) = compose(s1, compose(s2, s3))$
$compose(s1, s2, s3, s4) = compose(s1, compose(s2, compose(s3, s4)))$
$compose(s1, s2, s3, s4, s5) = compose(s1, compose(s2, compose(s3, compose(s4, s5))))$
$compose(s, \texttt{""}) = s$
$compose(\texttt{""}, s) = s$
$between(x, y, z) = and(geq(y, x), geq(z, y))$
$length(emsl) = 0$
$length(ins(st, sl)) = S(length(sl))$
$nextstate(sl, x) = next(sl, S(0), x)$
$next(emsl, x, y) = terminated$
$next(ins(st, sl), x, y) =$
    $if(between(S(0), y, S(length(sl))),$
        $if(eq(x, y), get_{state}(st), next(sl, S(x), y)),$
        $terminated)$
$peel(T(string, L)) = if(empty(L), string, compose(string, \texttt{"("}, peel(L), \texttt{")"}))$
$peel(emt) = \texttt{""}$
$peel(ins(t, L)) =$
    $if(empty(L), peel(t), compose(peel(t), \texttt{","}, peel(L)))$
$menu(L, spec) = menu'(steplist(L, spec))$
$menu'(emsl) = \texttt{""}$
$menu'(ins(st, sl)) =$

$$
\begin{aligned}
&\quad if(empty(get_{actterms}(st)),\\
&\qquad compose(get_{act}(st), NEWLINE, menu'(sl)),\\
&\qquad compose(get_{act}(st), \texttt{"("}, peel(get_{actterms}(st)), \texttt{")"}, NEWLINE, menu'(sl)))\\
&implies(c_{sims0}(spec), and(c_0(get_{adt}(spec)), c_6(spec))) = \mathsf{t}\\
&implies(c_{sims2}(state, spec), and(c'_{sims2}(state, spec), c_7(state, spec))) = \mathsf{t}\\
&c'_{sims2}(state, spec) = c'(steplist(state, spec), get_{adt}(spec))\\
&c'(emsl, d) = \mathsf{t}\\
&c'(ins(st, sl), d) = and(c''(st, d), c'(sl, d))\\
&c''(ST(string, L, nstate), d) = if(empty(L), \mathsf{t}, c'''(L, d))\\
&c'''(emt, d) = \mathsf{t}\\
&c'''(ins(t, L), d) = and(c_1(t, d), c'''(L, d))
\end{aligned}
$$

act $r_{u2}, s_{u1} : Command$
$r_{u1}, r_{u2}, s_{u1} : Status$
$r_{u1}, r_{u2}, s_{u1}, s_{u2} : Nat$
$s_{u2} : String$
$r_{v2} : Status$
$r_{v2}, s_{v1} : Term$

proc $SimS_0 =$

$$
\begin{aligned}
&\sum_{spec:Spec} r_{u1}(spec) \cdot\\
&\qquad (s_{u2}(ok) \cdot SimS_1(spec) \lhd c_{sims0}(spec) \rhd s_{u2}(notok(message_{sims0}(spec))) \cdot SimS_0)+\\
&r_{u1}(terminate)+\\
&r_{u1}(restart) \cdot SimS_0
\end{aligned}
$$

$SimS_1(spec : Spec) =$

$$
\begin{aligned}
&\sum_{init:Termlist} r_{u1}(init) \cdot SimS_2(spec, init)+\\
&r_{u1}(terminate)+\\
&r_{u1}(restart) \cdot SimS_0
\end{aligned}
$$

$SimS_2(spec : Spec, state : Termlist) =$

$$
\begin{aligned}
&(s_{u2}(notok(message_{sims2}(state, spec))) \cdot SimS_1(spec)\\
&\qquad \lhd c_{sims2}(state, spec) \rhd s_{u2}(menu(state, spec)) \cdot SimS_3(spec, steplist(state, spec)))
\end{aligned}
$$

$SimS_3(spec : Spec, stpl : Steplist) =$

$$
\begin{aligned}
&(\sum_{choice:Nat} r_{u1}(choice) \cdot SimS_4(spec, stpl, choice)\\
&\qquad \lhd between(S(0), choice, length(stpl)) \rhd s_{u2}(wrongchoice) \cdot SimS_3(spec, stpl))+\\
&r_{u1}(terminate)+\\
&r_{u1}(restart) \cdot SimS_0
\end{aligned}
$$

$SimS_4(spec : Spec, stpl : Steplist, choice : Nat) =$

$$
\begin{aligned}
&SimS_2(spec, r(nextstate(stpl, choice)))\\
&\qquad \lhd not(eq(nextstate(stpl, choice), terminated)) \rhd s_{u2}(termination) \cdot SimS_1(spec)
\end{aligned}
$$

4.3 Implementation of the simulator

We have implemented the Sim component that is specified above by letting it communicate with the Stepper and the RG component. Roughly and informally speaking, the simulator is implemented as follows. For computing the menu the Sim component receives a steplist from the Stepper. Then it extracts all the actions that are in this list. The arguments of these actions may be data terms which are not in normal form. The Sim component sends these terms to the Representant Generator which reduces them to their "most simple form". These normalised terms are still represented in internal ToolBus format. When Sim receives these terms from the RG component it converts them to μCRL format and sends a list of simplified actions to the UI component which displays this list on the computer screen. The detailed specification is given below.

func $menu : String \to Status$
$head : Termlist \to Term$
$tail : Termlist \to Termlist$
$one : Termlist \to Bool$

act $r_{v1}, s_{v1} : Command$
$r_{v1}, s_{v1} : Status$
$r_{v1}, s_{v1} : Datatype$

var $t : Term$
$L : Termlist$

rew $head(emt) = T(\texttt{""}, emt)$
$head(ins(t, L)) = t$
$tail(emt) = emt$
$tail(ins(t, L)) = L$
$one(emt) = \mathsf{f}$
$one(ins(t, L)) = empty(L)$

proc $SimI_0 =$

$\quad \sum_{spec:Spec} r_{u1}(spec) \,.\, s_{c1}(spec) \,.$
$\qquad (\sum_{s:String} r_{c2}(notok(s)) \,.\, s_{u2}(notok(s)) \,.\, SimI_0 +$
$\qquad r_{c2}(ok) \,.\, s_{u2}(ok) \,.\, SimI_1(get_{adt}(spec))) +$
$\quad r_{u1}(terminate) \,.\, s_{c1}(terminate) \,.\, s_{v1}(terminate) +$
$\quad r_{u1}(restart) \,.\, s_{c1}(restart) \,.\, s_{v1}(restart) \,.\, SimI_0$

$SimI_1(d : Datatype) =$
$\quad s_{v1}(d) \,.$
$\qquad (\sum_{s:String} r_{v2}(notok(s)) \,.\, s_{u2}(notok(s)) \,.\, SimI_0 +$
$\qquad r_{v2}(ok) \,.\, s_{u2}(ok) \,.\, SimI_2)$

$SimI_2 =$
$\quad r_{u1}(restart) \,.\, s_{c1}(restart) \,.\, s_{v1}(restart) \,.\, SimI_0 +$
$\quad r_{u1}(terminate) \,.\, s_{c1}(terminate) \,.\, s_{v1}(terminate) +$
$\quad \sum_{init:Termlist} r_{u1}(init) \,.\, s_{c1}(init) \,.\, SimI_3$

$SimI_3 =$
$\quad \sum_{s:String} r_{c2}(notok(s)) \,.\, s_{u2}(notok(s)) \,.\, SimI_2 +$
$\quad \sum_{stpl:Steplist} r_{c2}(stpl) \,.\, SimI_4(stpl, stpl, \texttt{""})$

$SimI_4(stpl' : Steplist, stpl : Steplist, buffer : String) =$
$\quad (SimI_5(get_{actterms}(head(stpl')), tail(stpl'), stpl, compose(buffer, get_{act}(head(stpl')), \texttt{"("}))$
$\qquad \lhd not(empty(get_{actterms}(head(stpl')))) \rhd$
$\qquad\qquad SimI_4(tail(stpl'), stpl, compose(buffer, get_{act}(head(stpl')), NEWLINE)))$
$\qquad\quad \lhd not(empty(stpl')) \rhd s_{u2}(menu(buffer)) \,.\, SimI_6(stpl)$

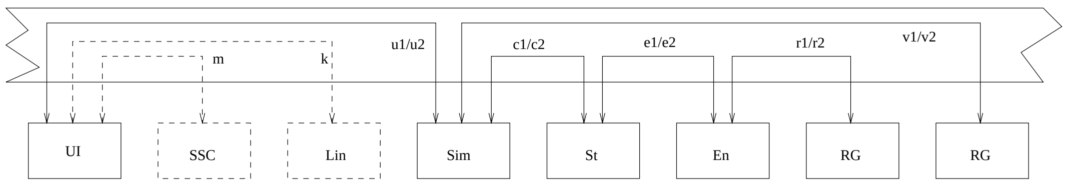

Figure 3: Communication via the ToolBus

$$
\begin{aligned}
&SimI_5(actterms : Termlist, stpl' : Steplist, stpl : Steplist, buffer : String) = \\
&\quad s_{v1}(head(actterms)) \,.\, \\
&\qquad (\textstyle\sum_{rep:\,Term} r_{v2}(rep) \,.\, (SimI_4(stpl', stpl, compose(buffer, peel(rep), \texttt{")"}, NEWLINE)) \\
&\qquad\quad \triangleleft one(actterms) \triangleright SimI_5(tail(actterms), stpl', stpl, compose(buffer, peel(rep), \texttt{","})))+ \\
&\qquad \textstyle\sum_{s:String} r_{v2}(notok(s)) \,.\, s_{u2}(notok(s)) \,.\, SimI_0)
\end{aligned}
$$

$$
\begin{aligned}
&SimI_6(stpl : Steplist) = \\
&\quad r_{u1}(restart) \,.\, s_{c1}(restart) \,.\, s_{v1}(restart) \,.\, SimI_0 + \\
&\quad r_{u1}(terminate) \,.\, s_{c1}(terminate) \,.\, s_{v1}(terminate) + \\
&\quad \textstyle\sum_{choice:Nat} r_{u1}(choice) \,.\, \\
&\qquad (SimI_7(stpl, choice) \triangleleft between(S(0), choice, length(stpl)) \triangleright s_{u2}(wrongchoice) \,.\, SimI_6(stpl))
\end{aligned}
$$

$$
\begin{aligned}
&SimI_7(stpl : Steplist, choice : Nat) = \\
&\quad s_{c1}(r(nextstate(stpl, choice))) \,.\, SimI_3 \\
&\qquad \triangleleft not(eq(nextstate(stpl, choice), terminated)) \triangleright s_{u2}(termination) \,.\, SimI_2
\end{aligned}
$$

4.4 Realisation of the Simulator

We have realised the implementation specified above in C. The translation was rather straightforward by using the ToolBus primitives for term manipulation like `TBmake` and `TBmatch`. These primitives are included as C-libraries. The C-code can be found in Appendix D.

5 Realisation of the Toolset

5.1 Communication via the ToolBus

We let the tools communicate via the ToolBus, 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 3. 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 some more 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 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 [7].

- 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 $a_1 \dots a_n$ it sends `snd-event`(receive$a_1 a_2 \dots a_n$,""), 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$a_1 a_2 \dots a_n$).

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 simulator is given in Appendix C. The reader is not supposed to understand this script, it is only ment as an illustration.

5.2 Realisation of the components

Prototypes of the components St, En, RG and Sim have been constructed in C. The UI is realised in TCL/TK. The SSC checker and the Linearisator are developed in ASF+SDF. The code of St, En and RG can be found in [6]. The code of UI and Sim are given in appendices E and D. The code of all above mentioned components can be obtained in electronic form by contacting the author. It appeared that, given the implementations and the tool/ToolBus interaction scheme it is quite straightforward to make an initial realisation of the simulator. However, 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.

6 Conclusions

The development time of the simulator tool was rather short, about 1.5 month. This was mainly due to the fact that we only had to build two components, UI and SIM, by reusing six components that were already provided in [6]. Moreover, it helped that the existing components were formally specified and well documented. Roughly speaking, we have experimented with two techniques. First, we used the ToolBus as a software architecture for building large, heterogeneous and distributed systems. Second, we applied μCRL as a formalism for specifying and analysing software systems with mathematical precision. Next we discuss the results of our experiments:

- We benefited from using the ToolBus in the following ways. First, the mapping of the μCRL specification of the simulator on the architecture of the ToolBus was rather straightforward as μCRL and the ToolBus have similar communication schemes. Second, the implementation of several components was simplified by the ToolBus C-libraries. These libraries provide powerful primitives for composing and decomposing terms of arbitrary length, releaving the implementor from memory allocation issues. Third, besides reusability the ToolBus allows for replacability. In particular, we have replaced the Representant Generator by a new version three times. The last version of this component was delivered by a third party. Finally, the ToolBus allowed us to connect different software components implemented in different programming languages like TCL/TK and C.

  There are also points for improvement. At the moment the ToolBus is an advanced but experimental software platform. For instance, it is currently not very efficient: it can exchange about 100 messages between tools per second. When simulating the queue given in Appendix A this is no problem. But if we simulate the Alternating Bit Protocol it can sometimes take 6 seconds before the next menu appears on the screen. However this is under improvement. Furthermore, we found the set of predefined TCL functions provided by the ToolBus not expressive enough. It is not possible to receive a ToolBus term within TCL-code. For connecting the UI component to the ToolBus we had to write a rather unnatural ToolBus script. This can be improved by extending the functionality of the current TCL/TK tool-adapter.

- We benefited from applying μCRL in the following aspects. First, there were mistakes in the specification of the components. This meant that we could reveal design errors in an early stage. Second, formal specification leads to software documentation of high quality. For example, we experienced that precise and clear specifications of the interfaces between the components led to better inter-human communications.

  We also encountered several weak points in the μCRL approach. First, it is sometimes difficult to determine to what extent something has to be specified. For example, it is yet clear up to what abstraction level the User Interface must be specified. Second, it is not an easy job to foresee all required aspects of the external behaviour of a component during the specification phase. For instance, both the Stepper and the Representant Generator check a process specification on its static correctness. The Sim component uses both these components which means that the simulated process is checked for the same constraint two times. If we had considered this in advanced we had extended both the St and the RG component with an option to switch off their static checks.

A The specification of the parallel queues

The specification of the sort *Bool* which is used below can be found in Appendix B.3.

sort D

func $d1, d2 :\to D$
$eq : D \times D \to Bool$

rew $eq(d1,d1) = \mathsf{t}$
$eq(d2,d2) = \mathsf{t}$
$eq(d1,d2) = \mathsf{f}$
$eq(d2,d1) = \mathsf{f}$

sort $queue$

func $empty :\to queue$
$in : D \times queue \to queue$
$ne : queue \to Bool$
$toe : queue \to D$
$untoe : queue \to queue$

var $d : D$
$q : queue$
$e : D$

rew $ne(empty) = \mathsf{f}$
$ne(in(d,q)) = \mathsf{t}$
$toe(empty) = d1$

$toe(in(d, empty)) = d$
$toe(in(d, in(e, q))) = toe(in(e, q))$
$untoe(empty) = empty$
$untoe(in(d, empty)) = empty$
$untoe(in(d, in(e, q))) = in(d, untoe(in(e, q)))$

act $s1 : D$
$r1 : D$
$r2 : D$
$s2 : D$
$c : D$

comm $s1|r2 = c$

proc $Top = \tau_{\{c\}}\partial_{\{s1,r2\}}Q(empty) \parallel R(empty)$

$$Q(q : queue) = \textstyle\sum_{d:D} r1(d) \,.\, Q(in(d, q)) + (s1(toe(q)) \,.\, Q(untoe(q)) \triangleleft ne(q) \triangleright \delta)$$

$$R(q : queue) = \textstyle\sum_{d:D} r2(d) \,.\, R(in(d, q)) + (s2(toe(q)) \,.\, R(untoe(q)) \triangleleft ne(q) \triangleright \delta)$$

B Existing components

The material in this section is taken from [6] and contains the specifications of the existing components, the Representant Generator and the Stepper, on which we have built the simulator.

B.1 The Representant Generator

B.1.1 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.

B.1.2 Specification of the Representant Generator

The detailed description of the Representant Generator is below. It first receives a datatype, which is described in appendix B.3, 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 $message_0 : \mathit{Datatype} \to \mathit{String}$
$message_1 : \mathit{Term} \times \mathit{Datatype} \to \mathit{String}$
$message_2 : \mathit{Openterm} \times \mathit{Datatype} \to \mathit{String}$
$c_0 : \mathit{Datatype} \to \mathit{Bool}$
$c_1 : \mathit{Term} \times \mathit{Datatype} \to \mathit{Bool}$
$c_2 : \mathit{Openterm} \times \mathit{Datatype} \to \mathit{Bool}$
$rep : \mathit{Term} \times \mathit{Datatype} \to \mathit{Term}$
$rep : \mathit{Openterm} \times \mathit{Datatype} \to \mathit{Openterm}$

var $d : \mathit{Datatype}$
$t : \mathit{Term}$
$t' : \mathit{Openterm}$

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

proc $RgS_0 =$

$\sum_{d:\mathit{Datatype}} r_{r1}(d) \cdot (s_{r2}(ok) \cdot RgS_1(d) \triangleleft c_0(d) \triangleright s_{r2}(notok(message_0(d))) \cdot RgS_0) +$
$r_{r1}(restart) \cdot RgS_0 +$
$r_{r1}(terminate)$

$RgS_1(d{:}\mathit{Datatype}) =$

$\sum_{t:\mathit{Term}} r_{r1}(t) \cdot$
$\quad (s_{r2}(rep(t,d)) \cdot RgS_1(d) \triangleleft c_1(t,d) \triangleright s_{r2}(notok(message_1(t,d))) \cdot RgS_1(d)) +$
$\sum_{t:\mathit{Openterm}} r_{r1}(t) \cdot$

$$(s_{r2}(rep(t,d))\,.\,RgS_1(d) \triangleleft c_2(t,d) \triangleright s_{r2}(notok(message_2(t,d)))\,.\,RgS_1(d)) +$$
$$r_{r1}(restart)\,.\,RgS_0 +$$
$$r_{r1}(terminate)$$

B.2 The Stepper

B.2.1 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.4.

B.2.2 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.

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

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: Each step in the steplist must have a corresponding summand in the process, by which it is caused. I.e., for a step which is represented by a triple $\langle a, tl, ntl\rangle$ (giving the action name, action parameters and new state), there must be a summand (see the specification of *Summand* in section B.4) in which the *actname* is a, and whose *actterms* and (new-)*state* can be instantiated to *tl* and *ntl* respectively. The substitution doing this should be same for *tl* and *ntl*, and consists of a part corresponding to the current state s, and a part which represents a true-making instantiation of the summand's condition. More formally: 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'' (the true-making instantiation) with $whatsorts(tl'', emv, get_{sig}(get_{adt}(spec))) = get_{sorts}(vl)$ 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: Conversely, each such true-making instantiation tl'' should give rise to a step in $steplist(s, spec)$. I.e., 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

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

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.

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

B.3 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.

B.3.1 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.

sort $\quad Bool$
func $\quad \mathsf{t}, \mathsf{f} :\to Bool$
$\qquad not : Bool \to Bool$
$\qquad and, or, implies : Bool \times Bool \to Bool$
$\qquad if : Bool \times Bool \times Bool \to Bool$

var $\quad x, y : Bool$
rew $\quad not(\mathsf{t}) = \mathsf{f}$
$\qquad not(\mathsf{f}) = \mathsf{t}$
$\qquad and(\mathsf{f}, x) = \mathsf{f}$
$\qquad and(\mathsf{t}, x) = x$
$\qquad or(\mathsf{f}, x) = x$
$\qquad or(\mathsf{t}, x) = \mathsf{t}$
$\qquad implies(x, \mathsf{t}) = \mathsf{t}$
$\qquad implies(\mathsf{f}, x) = \mathsf{t}$
$\qquad implies(\mathsf{t}, \mathsf{f}) = \mathsf{f}$
$\qquad if(\mathsf{t}, x, y) = x$
$\qquad if(\mathsf{f}, x, y) = y$

B.3.2 Natural numbers

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

sort $\quad Nat$
func $\quad S : Nat \to Nat$
$\qquad 0 :\to Nat$
$\qquad eq : Nat \times Nat \to Bool$
$\qquad pred : Nat \to Nat$
$\qquad geq : Nat \times Nat \to Bool$
var $\quad x, y : Nat$
rew $\quad eq(0, 0) = \mathsf{t}$
$\qquad eq(S(x), 0) = \mathsf{f}$
$\qquad eq(0, S(x)) = \mathsf{f}$
$\qquad eq(S(x), S(y)) = eq(x, y)$
$\qquad pred(S(x)) = x$
$\qquad geq(x, 0) = \mathsf{t}$
$\qquad geq(0, S(x)) = \mathsf{f}$
$\qquad geq(S(x), S(y)) = geq(x, y)$

B.3.3 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 $\ldots :\to String$
$true, false, Bool :\to String$
$eq : String \times String \to Bool$
$if : Bool \times String \times String \to String$
$string : Nat \to String$
$compose : String \times String \to String$

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

B.3.4 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 $ems :\to Stringlist$
$ins : String \times Stringlist \to Stringlist$
$test : String \times Stringlist \to Bool$
$eq : Stringlist \times Stringlist \to Bool$
$subsetof : Stringlist \times Stringlist \to Bool$
$conc : Stringlist \times Stringlist \to Stringlist$
var $x, y : String$
$s, s_1, s_2 : Stringlist$
rew $test(x, ems) = \mathsf{f}$
$test(x, ins(y, s)) = or(eq(x, y), test(x, s))$
$eq(ems, ems) = \mathsf{t}$
$eq(ems, ins(x, s)) = \mathsf{f}$

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

B.3.5 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 $eq_{\ldots}$ 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 $ok :\rightarrow Status$
$notok : String \rightarrow Status$
$eq_{ok}, eq_{notok} : Status \rightarrow Bool$

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

B.3.6 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 $restart, terminate :\rightarrow Command$
$eq_{restart}, eq_{terminate} : Command \rightarrow Bool$

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

B.3.7 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 $F : String \times Stringlist \times String \rightarrow Function$
$get_{name} : Function \rightarrow String$
$get_{args} : Function \rightarrow Stringlist$

$get_{targ} : \mathit{Function} \rightarrow \mathit{String}$
$sorts : \mathit{Function} \rightarrow \mathit{Stringlist}$

var $x, y : \mathit{String}$
$s : \mathit{Stringlist}$

rew $get_{name}(F(x, s, y)) = x$
$get_{args}(F(x, s, y)) = s$
$get_{targ}(F(x, s, y)) = y$
$sorts(F(x, s, y)) = ins(y, s)$

B.3.8 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 $emf : \rightarrow Functionlist$
$ins : \mathit{Function} \times Functionlist \rightarrow Functionlist$
$conc : Functionlist \times Functionlist \rightarrow Functionlist$
$exists : \mathit{String} \times \mathit{Stringlist} \times \mathit{String} \times Functionlist \rightarrow \mathit{Bool}$
$exists : \mathit{String} \times \mathit{Stringlist} \times Functionlist \rightarrow \mathit{Bool}$
$uniquetarget : Functionlist \rightarrow \mathit{Bool}$
$sorts : Functionlist \rightarrow \mathit{Stringlist}$
$existsconstr : \mathit{Variablelist} \times Functionlist \rightarrow \mathit{Bool}$
$existsconstr : \mathit{String} \times Functionlist \rightarrow \mathit{Bool}$

var $x, n, n_1, n_2, n_3 : \mathit{String}$
$L, L_1 : \mathit{Stringlist}$
$F_0, F_1 : Functionlist$
$f : \mathit{Function}$
$vl_1 : \mathit{Variablelist}$

rew $conc(emf, F_0) = F_0$
$conc(ins(f, F_0), F_1) = ins(f, conc(F_0, F_1))$
$exists(n, L, n_1, emf) = \mathsf{f}$
$exists(n, L, n_1, ins(F(n_2, L_1, n_3), F_0)) =$
$\quad or(and(eq(n, n_2), and(eq(L, L_1), eq(n_1, n_3))), exists(n, L, n_1, F_0))$
$exists(n, L, emf) = \mathsf{f}$
$exists(n, L, ins(F(n_2, L_1, n_3), F_0)) = or(and(eq(n, n_2), eq(L, L_1)), exists(n, L, F_0))$
$uniquetarget(emf) = \mathsf{t}$
$uniquetarget(ins(F(n, L, n_1), F_0)) = and(not(exists(n, L, F_0)), uniquetarget(F_0))$
$sorts(emf) = ems$
$sorts(ins(f, F_0)) = conc(sorts(f), sorts(F_0))$
$existsconstr(emv, F_0) = \mathsf{f}$
$existsconstr(ins(x, n, vl_1), F_0) = or(existsconstr(n, F_0), existsconstr(vl_1, F_0))$
$existsconstr(n, emf) = \mathsf{f}$
$existsconstr(n, ins(F(x, L, n_1), F_0)) = or(eq(n, n_1), existsconstr(n, F_0))$

B.3.9 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 $get_{sorts}(S(L_1, F_1, F_2)) = L_1$
$get_{cons}(S(L_1, F_1, F_2)) = F_1$
$get_{func}(S(L_1, F_1, F_2)) = F_2$
$boolsexist(S(L_1, F_1, F_2)) = and(exists(true, ems, Bool, F_1), exists(false, ems, Bool, F_1))$
$sortsexist(S) = subsetof(sorts(conc(get_{cons}(S), get_{func}(S))), get_{sorts}(S))$

B.3.10 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 *Variablelist*s. 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 $test_1(n, emv) = \mathsf{f}$

$test_1(n, ins(n_1, n_2, V)) = or(eq(n, n_1), test_1(n, V))$

$test_2(n, emv) = \mathsf{f}$

$test_2(n, ins(n_1, n_2, V)) = or(eq(n, n_2), test_1(n, V))$

$get_{sort}(n, ins(n_1, n_2, V)) = if(eq(n, n_1), n_2, get_{sort}(n, V))$

$get_{sorts}(emv) = ems$

$get_{sorts}(ins(n, n_1, V)) = ins(n_1, get_{sorts}(V))$

$targetsort(n, L_1, emv, S(L_2, F_1, F_2)) = targetsort(n, L_1, emv, conc(F_1, F_2))$

$targetsort(n, L_1, ins(n_1, n_2, V), S) =$
$\quad if(and(eq(n, n_1), eq(L_1, ems)), n_2, targetsort(n, L_1, V, S))$

$targetsort(n, L_1, V, ins(F(n_1, L_2, n_2), F_1)) =$
$\quad if(and(eq(n, n_1), eq(L_1, L_2)), n_2, targetsort(n, L_1, V, F_1))$

$noclash(emv, S) = \mathsf{t}$

$noclash(ins(n, n_1, V), S) =$
$\quad and(not(exists(n, ems, conc(get_{func}(S), get_{cons}(S)))), noclash(V, S))$

$sortsexist(emv, L_1) = \mathsf{t}$

$sortsexist(ins(n, n_1, V), L_1) = and(test(n_1, L_1), sortsexist(V, L_1))$

$nooverlap(emv, V) = \mathsf{t}$

$nooverlap(ins(n, n_1, V), V_1) = and(not(test_1(n, V)), nooverlap(V, V_1))$

$fresh(ems, V, S) = emv$

$fresh(ins(n, L_1), V, S) = ins(fresh(V, S, 0), n, fresh(L_1, ins(fresh(V, S, 0), n, V), S))$

$fresh(V, S, m) = if(or(exists(string(m), ems, conc(get_{cons}(S), get_{func}(S))),$
$\quad test_1(string(m), V)), fresh(V, S, S(m)), string(m))$

$conc(emv, V) = V$
$conc(ins(n_1, n_2, V), V_1) = ins(n_1, n_2, conc(V, V_1))$
$eq(emv, emv) = \mathsf{t}$
$eq(emv, ins(n, n_1, V)) = \mathsf{f}$
$eq(ins(n, n_1, V), emv) = \mathsf{f}$
$eq(ins(n, n_1, V), ins(n_2, n_3, V_1)) = and(eq(n, n_2), and(eq(n_1, n_3), eq(V, V_1)))$

B.3.11 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 $T : String \times Termlist \rightarrow Term$
$emt : \rightarrow Termlist$
$ins : Term \times Termlist \rightarrow Termlist$
$empty : Termlist \rightarrow Bool$

$toe : Termlist \rightarrow Term$
$untoe : Termlist \rightarrow Termlist$
$eq : Term \times Term \rightarrow Bool$
$eq : Termlist \times Termlist \rightarrow Bool$
$welltyped : Term \times Variablelist \times Signature \rightarrow Bool$
$welltyped : Termlist \times Variablelist \times Signature \rightarrow Bool$
$closed : Term \times Signature \rightarrow Bool$
$closed : Termlist \times Signature \rightarrow Bool$
$whatsort : Term \times Variablelist \times Signature \rightarrow String$
$whatsorts : Termlist \times Variablelist \times Signature \rightarrow Stringlist$
$subst : Termlist \times Variablelist \times Term \rightarrow Term$
$subst : Termlist \times Variablelist \times Termlist \rightarrow Termlist$
$subst : Term \times String \times Term \rightarrow Term$
$subst : Term \times String \times Termlist \rightarrow Termlist$
$repl : Termlist \times Variablelist \times String \rightarrow Term$
$mk_{terms} : Variablelist \rightarrow Termlist$
$if : Bool \times Term \times Term \rightarrow Term$
$if : Bool \times Termlist \times Termlist \rightarrow Termlist$
$constructorterm : Term \times Signature \rightarrow Bool$
$constructorterm : Termlist \times Signature \rightarrow Bool$

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

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

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

B.3.12 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	$emtll :\to Termlistlist$
	$ins : Termlist \times Termlistlist \to Termlistlist$
	$empty : Termlistlist \to Bool$
	$head : Termlistlist \to Termlist$
	$tail : Termlistlist \to Termlistlist$
	$test : Termlist \times Termlistlist \to Bool$
	$rem : Termlist \times Termlistlist \to Termlistlist$
	$\mathit{if} : Bool \times Termlistlist \times Termlistlist \to Termlistlist$
var	$L, L_1 : Termlist$
	$LL, LL_1 : Termlistlist$
rew	$empty(emtll) = \mathsf{t}$
	$empty(ins(L, LL)) = \mathsf{f}$
	$head(ins(L, LL)) = L$
	$tail(ins(L, LL)) = LL$
	$test(L, emtll) = \mathsf{f}$
	$test(L, ins(L_1, LL)) = \mathit{if}(eq(L, L_1), \mathsf{t}, test(L, LL))$
	$rem(L, emtll) = emtll$
	$rem(L, ins(L_1, LL)) = \mathit{if}(eq(L, L_1), rem(L, LL), ins(L_1, rem(L, LL)))$
	$\mathit{if}(\mathsf{t}, LL, LL_1) = LL$
	$\mathit{if}(\mathsf{f}, LL, LL_1) = LL_1$

B.3.13 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 $get_{term}(O(t, V)) = t$
$get_{vars}(O(t, V)) = V$
$correctopenterm(O(t, V), S) =$
$\quad and(noclash(V, S), and(sortsexist(V, get_{sorts}(S)), welltyped(t, V, S)))$
$whatsort(O(t, V), S) = whatsort(t, V, S)$

B.3.14 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 $get_{vars}(E(v, t_1, t_2)) = v$
$get_{left}(E(v, t_1, t_2)) = t_1$
$get_{right}(E(v, t_1, t_2)) = t_2$

B.3.15 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 $S : \mathit{Signature}$
$V : \mathit{Variablelist}$
$t_1, t_2 : \mathit{Term}$
$L : \mathit{Equationlist}$
rew $correcteqs(S, eme) = \mathsf{t}$
$correcteqs(S, ins(E(V, t_1, t_2), L)) =$
$\quad and(and(and(sortsexist(V, get_{sorts}(S)), noclash(V, S)),$
$\quad and(welltyped(t_1, V, S), welltyped(t_2, V, S))), correcteqs(S, L))$

B.3.16 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 $\mathit{Datatype}$
func $D : \mathit{Signature} \times \mathit{Equationlist} \to \mathit{Datatype}$
$get_{sig} : \mathit{Datatype} \to \mathit{Signature}$
$get_{eqs} : \mathit{Datatype} \to \mathit{Equationlist}$
$ssc : \mathit{Datatype} \to \mathit{Bool}$
$equal : \mathit{Openterm} \times \mathit{Openterm} \times \mathit{Datatype} \to \mathit{Bool}$
$equal : \mathit{Term} \times \mathit{Term} \times \mathit{Datatype} \to \mathit{Bool}$
$equal : \mathit{Termlist} \times \mathit{Termlist} \times \mathit{Datatype} \to \mathit{Bool}$
var $S : \mathit{Signature}$
$EL : \mathit{Equationlist}$
$t_1, t_2 : \mathit{Term}$
$L, L_1 : \mathit{Termlist}$
$d : \mathit{Datatype}$
rew $get_{sig}(D(S, EL)) = S$
$get_{eqs}(D(S, EL)) = EL$
$ssc(D(S, EL)) =$
$\quad and(uniquetarget(conc(get_{func}(S), get_{cons}(S))), and(sortsexist(S),$
$\quad\quad and(correcteqs(S, EL), boolsexist(S))))$
$equal(emt, emt, d) = \mathsf{t}$
$equal(ins(t_1, L), emt, d) = \mathsf{f}$
$equal(emt, ins(t_1, L), d) = \mathsf{f}$
$equal(ins(t_1, L), ins(t_2, L_1), d) = and(equal(t_1, t_2, d), equal(L, L_1, d))$

B.3.17 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.4 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 [3].

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.4.1 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.4.2 Summand

We now start defining a linear process. The general definition of a linear process equation, based on the linear process of [3] is the following:

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

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

var $\quad v, v_1 : Variablelist$
$\quad an, an_1 : String$
$\quad at, at_1 : Termlist$
$\quad st, st_1 : NTermlist$
$\quad c, c_1 : Term$
rew $\quad get_{vars}(SMD(v, an, at, st, c)) = v$
$\quad get_{actname}(SMD(v, an, at, st, c)) = an$
$\quad get_{actterms}(SMD(v, an, at, st, c)) = at$
$\quad get_{state}(SMD(v, an, at, st, c)) = st$
$\quad get_{cond}(SMD(v, an, at, st, c)) = c$
$\quad eq(SMD(v, an, at, st, c), SMD(v_1, an_1, at_1, st_1, c_1)) =$
$\qquad and(eq(v, v_1), and(eq(an, an_1), and(eq(at, at_1), and(eq(st, st_1), eq(c, c_1)))))$

B.4.3 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 $\quad Sumlist$
func $\quad eml : \to Sumlist$
$\quad ins : Summand \times Sumlist \to Sumlist$
$\quad eq : Sumlist \times Sumlist \to Bool$
$\quad empty : Sumlist \to Bool$
$\quad head : Sumlist \to Summand$
$\quad tail : Sumlist \to Sumlist$
$\quad ssc : Sumlist \times Variablelist \times Signature \to Bool$

var $\quad vl, vl_1 : Variablelist$

	$sl, sl_1 : \mathit{Sumlist}$
	$sm, sm_1 : \mathit{Summand}$
	$adt : \mathit{Datatype}$
	$s : \mathit{String}$
	$tl : \mathit{Termlist}$
	$ntl : \mathit{NTermlist}$
	$t : \mathit{Term}$
	$sig : \mathit{Signature}$
	$eql : \mathit{Equationlist}$
rew	$eq(eml, eml) = \mathsf{t}$
	$eq(ins(sm, sl), eml) = \mathsf{f}$
	$eq(sl, ins(sm, sl)) = \mathsf{f}$
	$eq(ins(sm, sl), ins(sm_1, sl_1)) = and(eq(sm, sm_1), eq(sl, sl_1))$
	$empty(eml) = \mathsf{t}$
	$empty(ins(sm, sl)) = \mathsf{f}$
	$head(ins(sm, sl)) = sm$
	$tail(ins(sm, sl)) = sl$
	$ssc(eml, vl, sig) = \mathsf{t}$
	$ssc(ins(SMD(vl_1, s, tl, ntl, t), sl), vl, sig) =$
	$\quad and(matchsort(vl, ntl, vl_1, sig),$
	$\quad and(eq(whatsort(t, conc(vl, vl_1), sig), Bool),$
	$\quad and(welltyped(t, conc(vl, vl_1), sig),$
	$\quad and(welltyped(tl, conc(vl, vl_1), sig),$
	$\quad and(noclash(vl, sig),$
	$\quad and(noclash(vl_1, sig),$
	$\quad and(nooverlap(vl, vl_1),$
	$\quad and(welltyped(ntl, conc(vl, vl_1), sig), ssc(sl, vl, sig))))))))$

B.4.4 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	$PROC : \mathit{Variablelist} \times \mathit{Sumlist} \rightarrow ProcSpec$
	$get_{vars} : ProcSpec \rightarrow \mathit{Variablelist}$
	$get_{sums} : ProcSpec \rightarrow \mathit{Sumlist}$
var	$v : \mathit{Variablelist}$
	$s : \mathit{Sumlist}$
rew	$get_{vars}(PROC(v, s)) = v$
	$get_{sums}(PROC(v, s)) = s$

B.4.5 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	$SPEC : \mathit{Datatype} \times ProcSpec \rightarrow Spec$

$get_{adt} : Spec \rightarrow Datatype$
$get_{proc} : Spec \rightarrow ProcSpec$
$ssc : Spec \rightarrow Bool$

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.4.6 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 $ST : String \times Termlist \times NTermlist \rightarrow Step$
$get_{act} : Step \rightarrow String$
$get_{actterms} : Step \rightarrow Termlist$
$get_{state} : Step \rightarrow NTermlist$
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.4.7 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 $empty(emsl) = \mathsf{t}$
$empty(ins(st,sl)) = \mathsf{f}$
$head(ins(st,sl)) = st$
$tail(ins(st,sl)) = sl$

C ToolBus script

```
process UI is
let C : ui,
    V : int,
    T : term,
    T2 : term
in
   execute(ui, C?) .
   (rec-event(C,button(enterFile)) .
    snd-eval(C,get-spec) .
      (rec-value(C,file-error) .
       snd-ack-event(C,button(enterFile))
           +
       rec-value(C,spec(T?,T2?)) .
       snd-msg(u1(spec(T,T2))) .
         (rec-msg(u2(notok(T?))) .
          snd-do(C,not-ok(T)) .
          snd-ack-event(C,button(enterFile))
              +
          rec-msg(u2(ok)) .
            (rec-msg(u2(ok)) .
             snd-eval(C,get-init) .
             rec-value(C,T?) .
             snd-msg(u1(T)) .
             snd-ack-event(C,button(enterFile))
                      +
             rec-msg(u2(notok(T?))) .
             snd-do(C,not-ok(T)) .
             snd-ack-event(C,button(enterFile))
            )
         )
      )
     +
    rec-event(C,button(restart)) .
    snd-msg(u1(restart)) .
    snd-ack-event(C,button(restart))
        +
    rec-event(C,button(choice)) .
    snd-eval(C,nop) .
    rec-value(C,choice(V?)) .
    snd-msg(u1(V)) .
    snd-ack-event(C,button(choice))
        +
    rec-msg(u2(menu(T?))) .
    snd-do(C,display-menu(T))
        +
    rec-msg(u2(notok(T?))) .
    snd-do(C,not-ok(T))
        +
    rec-msg(u2(s-termination)) .
    snd-do(C,s-termination)
   ) * rec-event(C,quit) . snd-msg(u1(terminate))
endlet


process SIMULATOR is
let C : simulator,
    T : term
in
   execute(simulator, C?) .
   (rec-event(C,receiveu1,"") .
    rec-msg(u1(T?)) .
    snd-do(C,u1(T)) .
    snd-ack-event(C,receiveu1)
       +
    rec-event(C,send,c1(T?)) .
    snd-msg(c1(T)) .
    snd-ack-event(C,send)
       +
    rec-event(C,receivec2,"") .
    rec-msg(c2(T?)) .
    snd-do(C,c2(T)) .
    snd-ack-event(C,receivec2)
       +
    rec-event(C,send,u2(T?)) .
    snd-msg(u2(T)) .
    snd-ack-event(C,send)
       +
    rec-event(C,receivev2,"") .
    rec-msg(v2(T?)) .
    snd-do(C,v2(T)) .
    snd-ack-event(C,receivev2)
       +
    rec-event(C,send,v1(T?)) .
    snd-msg(v1(T)) .
    snd-ack-event(C,send)
   ) * 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)) .
            snd-ack-event(S,send) +
    rec-event(S,send,c2(T?)) . snd-msg(c2(T)) .
            snd-ack-event(S,send) +
    rec-event(S,receivee2,"") .
            rec-msg(e2(T?)) .
            snd-do(S,e2(T)).  snd-ack-event(S,receivee2) +
    rec-event(S,receivec1,"") . rec-msg(c1(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?)) .
         snd-msg(e2(T)) . snd-ack-event(E,send) +
    rec-event(E,send,r1(T?)) .
              snd-msg(r1(T)) .
              snd-ack-event(E,send) +
    rec-event(E,receiver2,"") . rec-msg(r2(T?)) .
              snd-do(E,r2(T)) .
        snd-ack-event(E,receiver2) +
    rec-event(E,receivee1,"") . rec-msg(e1(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?)) .
               snd-msg(r2(T)) . snd-ack-event(R1,send) +
    rec-event(R1,receiver1,"") .  rec-msg(r1(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?)) .
               snd-msg(v2(T)) . snd-ack-event(R2,send) +
    rec-event(R2,receiver1,"") .  rec-msg(v1(T?))
            . snd-do(R2,r1(T)) .
        snd-ack-event(R2,receiver1)) *
    rec-disconnect(R2)
endlet
```

```
tool ui is {command = "wish-adapter -script ui.tcl"}
tool representant is {command = "representant"}
tool enumerator is {command = "enumerator"}
tool stepper is {command = "stepper"}
tool simulator is {command = "simulator"}

toolbus(STEPPER,ENUMERATOR,REPRESENTANT1,SIMULATOR,REPRESENTANT2,UI)
```

D simulator.c

In the C-code below four files are included. The file `TB.h` can be found in [1]. The files `datatypes.h` and `communicate.c` can be found in [6].

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

void peel_term(term *t);
void peel_termlist(term *termlist);

int bufsize, ml=0;
char *buffer;

void bufferinit(void) {
  buffer = malloc(1);
  bufsize = 1;
  buffer[0] = 0;
}

void bufferadd(char *str) {
char *tmp;
  if (strlen(str) + strlen(buffer) + 1 > bufsize) {
    tmp = malloc(strlen(str) + strlen(buffer) + 1);
    strcpy(tmp,buffer);
    free(buffer);
    buffer = tmp;
    bufsize = strlen(tmp) + strlen(str) + 1;
  }
  strcat(buffer, str);
}

void bufferempty(void) {
  buffer[0] = 0;
}

void peel_term(term *t)
{ term *termlist;
```

```
  char *string;

  TBmatch(t,"t(%s,%t)",&string,&termlist);
  if (isemptytl(termlist)) {
      bufferadd(string);
  } else {
      bufferadd(string);
      bufferadd("(");
      peel_termlist(termlist);
      bufferadd(")");
  }
}

void peel_termlist(term *tl)
{ term *t,*tl1;

  if (TBmatch(tl,"ins(%t,%t)",&t,&tl1))
       if (isemptytl(tl1)) {
          peel_term(t);
       } else {
           peel_term(t);
           bufferadd(",");
           peel_termlist(tl1);
       }
}


char *menu(term *stpl)
{ term *hdstpl,*tlstpl,
       *arguments,*nextstate;
  char *actname;
  int  i=1;

  bufferempty();
  while (!TBmatch(stpl,"emsl")) {
    TBmatch(stpl,"ins(%t,%t)",&hdstpl,&tlstpl);
    TBmatch(hdstpl,"st(%s,%t,%t)",&actname,&arguments,&nextstate);
    if (isemptytl(arguments)) {
      bufferadd(actname);
      bufferadd("\n");
    } else {
      bufferadd(actname);
      bufferadd("(");
      peel_termlist(arguments);
      bufferadd(")");
      bufferadd("\n");

    }
    stpl = tlstpl;
    i = i + 1;
  };
  ml = i-1;
  return(buffer);
}

term *next_state(term *stpl, int j)
{ term *hdstpl,*tlstpl,
       *arguments,*nextstate;
  char *actname;
  int  i=1;

  while (i <= j) {
    TBmatch(stpl,"ins(%t,%t)",&hdstpl,&tlstpl);
    TBmatch(hdstpl,"st(%s,%t,%t)",&actname,&arguments,&nextstate);
    stpl = tlstpl;
    i = i + 1;
  };
  return(nextstate);
}

void main(int argc, char *argv[]) /* main program of the simulator */
{ term *spec=NULL,*adt=NULL,*proc=NULL,
       *s=NULL,*g=NULL,*b=NULL,
       *tl=NULL,*stpl=NULL,*stpl1=NULL,*hdstpl=NULL,*tlstpl=NULL,
       *hdg=NULL,*tlg=NULL,
       *hdactterms=NULL,*tlactterms=NULL,
       *actterms=NULL,
       *rep=NULL,
       *t1=NULL,*t2=NULL,*u=NULL,
       *arguments=NULL,
       *state=NULL,*nextstate=NULL,*init=NULL,
       *tl2=NULL,*t3=NULL,*tl3=NULL,*t4=NULL;
  char *m="This is the initial value                 ",
       *str, /* these are allocated later on */
       *actname;
  int  choice=0;

  TBprotect(&in);
  TBprotect(&spec);
  TBprotect(&adt);
  TBprotect(&proc);
  TBprotect(&s);
  TBprotect(&g);
  TBprotect(&b);
  TBprotect(&tl);
  TBprotect(&stpl);
  TBprotect(&stpl1);
  TBprotect(&hdstpl);
  TBprotect(&tlstpl);
  TBprotect(&hdg);
  TBprotect(&tlg);
  TBprotect(&actterms);
  TBprotect(&hdactterms);
  TBprotect(&tlactterms);
  TBprotect(&t1);
  TBprotect(&t2);
  TBprotect(&u);
  TBprotect(&arguments);
  TBprotect(&state);
  TBprotect(&nextstate);
  TBprotect(&init);
  TBprotect(&tl2);
  TBprotect(&t3);
  TBprotect(&tl3);
  TBprotect(&t4);

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

  bufferinit();
Sim0:
  receive_u1();
  TBmatch(in,"u1(%t)",&spec);
  if (TBmatch(spec,"terminate")) {send(TBmake("c1(terminate)"));
                                  send(TBmake("v1(terminate)"));
                                  goto Finito;}
  if (TBmatch(spec,"restart")) {send(TBmake("c1(restart)"));
                                send(TBmake("v1(restart)"));
                                goto Sim0;}
  send(TBmake("c1(%t)",spec));
  receive_c2();
  if (TBmatch(in,"c2(notok(%s))",&m)) {
        send(TBmake("u2(notok(%s))",m));
        goto Sim0;}
  else {TBmatch(spec,"spec(%t,%t)",&adt,&proc); send(TBmake("u2(ok)")); goto Sim1;}
Sim1:
  send(TBmake("v1(%t)",adt));
  receive_v2();
  if (TBmatch(in,"v2(notok(%s))",&m)) {
        send(TBmake("u2(notok(%s))",m));
        goto Sim0;}
  else {send(TBmake("u2(ok)")); goto Sim2;}
Sim2:
  ml=0;
  receive_u1();
  TBmatch(in,"u1(%t)",&init);
  if (TBmatch(init,"restart")) {send(TBmake("c1(restart)"));
                                send(TBmake("v1(restart)")); goto Sim0;}
  if (TBmatch(init,"terminate"))
        {send(TBmake("c1(terminate)")); send(TBmake("v1(terminate)"));
         goto Finito;}
  send(TBmake("c1(%t)",init));
  goto Sim3;
```

```
Sim3:
  receive_c2();
  TBmatch(in,"c2(%t)",&stpl);
  stpl1=stpl;
  if (TBmatch(stpl,"notok(%s)",&m)) {
        send(TBmake("u2(notok(%s))",m)); goto Sim2;}
  bufferempty();
  goto Sim4;

Sim4:
  if (TBmatch(stpl,"emsl")) {send(TBmake("u2(menu(%s))",buffer)); goto Sim6;}
  else {TBmatch(stpl,"ins(%t,%t)",&hdstpl,&tlstpl);
        TBmatch(hdstpl,"st(%s,%t,%t)",&actname,&actterms,&nextstate);
        if (TBmatch(actterms,"emt")) {stpl=tlstpl;
                                    bufferadd(actname);
                           bufferadd("\n"); ml=ml+1;
                                    goto Sim4;}
        else {stpl=tlstpl; bufferadd(actname); bufferadd("("); goto Sim5;}
 }

Sim5:
  TBmatch(actterms,"ins(%t,%t)",&hdactterms,&tlactterms);
  send(TBmake("v1(%t)",hdactterms));
  receive_v2();
  if (TBmatch(in,"v2(notok(%s))",&str))
       {send(TBmake("u2(notok(%s))",str)); goto Sim0;}
  else
       {TBmatch(in,"v2(%t)",&rep);
        if (TBmatch(tlactterms,"emt"))
             {peel_term(rep);
              bufferadd(")");
              bufferadd("\n"); ml=ml+1;
              goto Sim4;
        } else {actterms=tlactterms;
                peel_term(rep);
                bufferadd(",");
                goto Sim5;}
  }

Sim6:
  receive_u1();
  TBmatch(in,"u1(%t)",&init);
  if (TBmatch(init,"restart")) {send(TBmake("c1(restart)"));
                              send(TBmake("v1(restart)")); goto Sim0;}
  if (TBmatch(init,"terminate"))
        {send(TBmake("c1(terminate)")); send(TBmake("v1(terminate)"));
         goto Finito;}
  TBmatch(in,"u1(%d)",&choice);
  if (1 <= choice && choice <= ml) {goto Sim7;}
  else {send(TBmake("u2(notok(wrongchoice))")); goto Sim6;}
```

```
Sim7:
  if (TBmatch(next_state(stpl1,choice),"i(%t)",&state))
          {send(TBmake("c1(%t)",state)); goto Sim3;}
  else {send(TBmake("u2(s-termination)")); goto Sim2;}

 Finito:
  terminate() ;
  TBunprotect(&spec);
  TBunprotect(&adt);
  TBunprotect(&proc);
  TBunprotect(&s);
  TBunprotect(&g);
  TBunprotect(&b);
  TBunprotect(&tl);
  TBunprotect(&stpl);
  TBunprotect(&stpl1);
  TBunprotect(&hdstpl);
  TBunprotect(&tlstpl);
  TBunprotect(&hdg);
  TBunprotect(&tlg);
  TBunprotect(&actterms);
  TBunprotect(&t1);
  TBunprotect(&t2);
  TBunprotect(&u);
}
```

E ui.tcl

```
proc BindSSL { Box } {
  bind $Box <B1-Motion> {%W select from [%W nearest %y]}
  bind $Box <Shift-B1-Motion> {%W select from [%W nearest %y]}
  bind $Box <Shift-Button-1> {%W select from [%W nearest %y]}
  bind $Box <Double-Button-1> {%W select from [%W nearest %y]}
}


proc dialog {w title text bitmap default args} {
        global button

        # 1. Create the top-level window and divide it into top
        # and bottom parts.

        toplevel $w -class Dialog
        wm title $w $title
        wm iconname $w Dialog
        frame $w.top -relief raised -bd 1
        pack $w.top -side top -fill both
        frame $w.bot -relief raised -bd 1
        pack $w.bot -side bottom -fill both
        # 2. Fill the top part with the bitmap and message.

        message $w.top.msg -width 3i -text $text\
                        -font -Adobe-Times-Medium-R-Normal-*-180-*
        pack $w.top.msg -side right -expand 1 -fill both\
                        -padx 3m -pady 3m
        if {$bitmap != ""} {
                label $w.top.bitmap -bitmap $bitmap
                pack $w.top.bitmap -side left -padx 3m -pady 3m
        }

        # 3. Create a row of buttons at the bottom of the dialog.

        set i 0
        foreach but $args {
                button $w.bot.button$i -text $but -command\
                                "set button $i"
                if {$i == $default} {
                        frame $w.bot.default -relief sunken -bd 1
                        raise $w.bot.button$i
                        pack $w.bot.default -side left -expand 1\
                                        -padx 3m -pady 2m
                        pack $w.bot.button$i -in $w.bot.default\
                                        -side left -padx 2m -pady 2m\
                                        -ipadx 2m -ipady 1m
                } else {
                        pack $w.bot.button$i -side left -expand 1\
                                        -padx 3m -pady 3m -ipadx 2m -ipady 1m
                }
                incr i
        }

        # 4. Set up a binding for <Return>, if there's a default,
        # set a grab, and claim the focus too.

        if {$default >= 0} {
                bind $w <Return> "$w.bot.button$default flash; \
                        set button $default"
        }
        set oldFocus [focus]
        grab set $w
        focus $w

        # 5. Wait for the user to respond, then restore the focus
        # and return the index of the selected button.

        tkwait variable button
        destroy $w
        focus $oldFocus
        return $button
}
```

```
set dn "~henri/toolbus/examples"
set gdn [format "%s%s" $dn "/"]
set fn "queue2"
set gfn [format "%s%s" $gdn "queue2"]


proc nop {} {}

proc rec-ack-event {n} {}

proc rec-terminate {n} { destroy . }


proc strcat {s1 s2} {
    return [format "%s%s" $s1 $s2]
}


wm geometry . 500x300
wm sizefrom . ""
wm maxsize . 1152 900

frame .left
frame .right

frame .right_top
frame .right_bottom

pack  .left -side left -expand yes -fill both -padx 2m -pady 2m
pack  .right -side right -fill y -padx 2m -pady 2m

pack .right_top -in .right -side top
pack .right_bottom -in .right -side bottom -expand yes -fill both

frame .right_top_left
frame .right_top_right

pack .right_top_left .right_top_right -in .right_top -side left

label .fileLabel -text "File:"
entry .fileEntry -width 20 -relief sunken -textvariable fn
global gfn

bind  .fileEntry <Return> {
   alldisable
   .menu delete 0 [.menu size]
   enable-menu
   TBsend "snd-event(button(restart))"
   TBsend "snd-event(button(enterFile))"
}
```

```
label .dirLabel -text "Directory:"
entry .dirEntry -width 20 -relief sunken -textvariable dn

bind  .dirEntry <Return> {
   alldisable
   .menu delete 0 [.menu size]
   enable-menu
   TBsend "snd-event(button(restart))"
   TBsend "snd-event(button(enterFile))"
}


pack .dirLabel .fileLabel -in .right_top_left -side top -anchor nw
pack .dirEntry .fileEntry -in .right_top_right -side top -anchor nw


label  .canvasLabel -text "Menu display"
listbox .menu -relief groove
pack .menu -in .left -side top -expand yes -fill both -padx 0m -pady 0m \
     -anchor nw
pack .canvasLabel -in .left -side top -padx 0m -pady 0m -anchor nw


# proc TBsend {str} {puts $str}

frame .qquit
frame .rrestart -relief sunken -bd 1

button .quit -text "Quit" -command {
   destroy .
   TBsend "snd-event(quit)"
}

proc alldisable {} {
    .restart configure -state disabled
    .quit configure -state disabled
    .dirEntry configure -state disabled
    .fileEntry configure -state disabled
    bind  .fileEntry <Return> {nop}
    bind  .dirEntry  <Return> {nop}
}

proc allenable {} {
    .restart configure -state normal
    .quit configure -state normal
    .fileEntry configure -state normal
    .dirEntry configure -state normal
    bind  .fileEntry <Return> {
     alldisable
     .menu delete 0 [.menu size]
    enable-menu
    TBsend "snd-event(button(restart))"
    TBsend "snd-event(button(enterFile))" }

   bind  .dirEntry <Return> {
    alldisable
     .menu delete 0 [.menu size]
    enable-menu
    TBsend "snd-event(button(restart))"
    TBsend "snd-event(button(enterFile))" }
}



button .restart -text "Start" -command {
   .menu delete 0 [.menu size]
   enable-menu
   alldisable
   TBsend "snd-event(button(restart))"
   TBsend "snd-event(button(enterFile))"
}

pack  .qquit -in .right_bottom -side right  -padx 1m -pady 7m \
      -anchor se
pack  .rrestart -in .right_bottom -side right -padx 1m -pady 7m \
      -anchor se

pack  .restart -in .rrestart -padx 2m -pady 2m \
      -ipadx 2m -ipady 2m
pack  .quit -in .qquit -padx 2m -pady 2m \
      -ipadx 2m -ipady 2m


proc display {str} { .menu insert end $str}

proc display-menu {str} {
if {$str == ""} {
   display "deadlock"
   disable-menu
   allenable
} else {
     set i 1
     while {$str != ""} {
        set break [string first "\n" $str]
        display [string range $str 0 [expr $break-1]]
        set str [string range $str [expr $break+1] end]
     }
     allenable
}
}
```

```
proc not-ok {str} {
   dialog .d {Error in specification} $str {} -1 OK
   allenable
   TBsend "snd-event(button(restart))"
}

proc disable-menu {} {
    bind .menu <Any-Button> {.menu select clear}
    bind .menu <B1-Motion>  {.menu select clear}
    bind .menu <ButtonRelease-1>  {.menu select clear}
    bind .menu <Double-Button>  {.menu select clear}
    bind .menu <Double-1>  {.menu select clear}
    bind .menu <Any-Triple-1>  {.menu select clear}
}

proc enable-menu {} {
   bind .menu <Any-Button> {}
   BindSSL .menu
   bind .menu <ButtonRelease-1> {
   if {[.menu curselection] == ""} {
   } else {
      alldisable
      TBsend "snd-event(button(choice))"
      TBsend "snd-value(choice([expr [.menu curselection] + 1]))"
      .menu delete 0 [.menu size]
   }
}
}

proc s-termination {} {
   disable-menu
   display-menu "Succesfull termination\n"
}


proc get-file-descriptor {fnn} {
    if {[file exists $fnn]} {
          return [open $fnn r]
          } else {return "error"}
}


proc get-spec {} {
    global gfn
    global fd2
    global dn
    global fn

    if {$dn != ""} {set gdn [strcat $dn "/"]} {set gdn $dn}
    set gfn [strcat $gdn $fn]
    set gfn.init [strcat $gfn ".init"]
    if {[file exists $gfn] && [file exists $gfn.init]} {
       set fd1 [open $gfn r]
       set fd2 [open $gfn.init r]
       TBsend "snd-value([read -nonewline $fd1])"
    } else {
       set mess [format "The file \"%s\" or the file \"%s\" does not exist." \
                $gfn $gfn.init]
       dialog .d {File} $mess {} -1 OK
       allenable
       TBsend "snd-value(file-error)"
    }
}

proc get-init {} {
    global fd2
    TBsend "snd-value([read -nonewline $fd2])"
}

bind .menu <ButtonRelease-1> {
   alldisable
   if {[.menu curselection] == ""} {
   } else {
     TBsend "snd-event(button(choice))"
     TBsend "snd-value(choice([expr [.menu curselection] + 1]))"
     .menu delete 0 [.menu size]
   }
}

BindSSL .menu
```

References

[1] J.A. Bergstra, P. Klint. The ToolBus– a Component Interconnection Architecture–. Technical Report P9408. Programming Research Group. University of Amsterdam, March 1994.

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

[3] 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.

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

[5] 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.

[6] D. Dams and J.F. Groote. Specification and Implementation of Components of a μCRL Toolbox. Logic Group Preprint Series 152. Utrecht Univ., Dept. of Philosophy. Heidelberglaan 8, 3584 CS Utrecht, The Netherlands. December 1995.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] P. Klint. A meta-environment for generating programming environments. ACM Transactions on Software Engineering and Methodology, 2(2):176-201, 1993.

[13] P. Klint. A Guide to ToolBus Programming (Version 0.9). November 15, 1995. This paper can be obtained by contacting the author.