Embedded System Simulation

Testdriving the ToolBus

Pieter A. Olivier

University of Amsterdam

Programming Research Group

email: olivierp@wins.uva.nl

Abstract

We have investigated the use of the ToolBus application architecture[BK95] for the development of embedded system simulators. We developed a flexible and intuitive simulator, stable enough to be used in a commercial setting, with an execution speed comparable to that of the simulated hardware. This extensible simulator is based on a loosely coupled set of tools that form a framework for efficient construction of new simulators.

Because this project is one of the first serious applications of the ToolBus, we have also set out to gain experience with ToolBus programming in a project of substantial proportions.

Contents

1 Introduction

In this paper, we use the term "embedded system" to identify any computer system for which the primary development tools do not run on the system itself. Typical examples are the computer systems that are build into household appliances, cars, and airplanes.

Many embedded systems are produced in small quantities and use non-standard hardware, for which hardly any programming support is available. The costs for buying or building tools such as, for instance, a simulator are in such cases only warranted if they can be amortized over a wide range of embedded systems. This means we must achieve a significant amount of reuse, both in design and coding efforts.

This is why we deployed a software interconnection architecture called *the ToolBus* for developing a simulator framework for embedded systems. The ToolBus application architecture was designed to control a number of heterogeneous components, possibly in a distributed environment. The ToolBus enforces formalization of the communication behaviour between the components of the system, making the interaction between them completely explicit. This leads to a set of loosely coupled tools with a well-defined input/output behaviour, greatly improving reusability. The ToolBus also provides us with the opportunity to implement each component in the most appropriate language.

An ideal simulator provides all the information about the state of the running program the user wants, and no more. To provide this information in a clear and concise way, a good and extensive user interface is a must. The considerable effort that must be invested into the creation of appealing and easy to use user interface components, logically focussed our attention on making these components reusable.

2 The ToolBus

The ToolBus is a software application architecture developed at the University of Amsterdam by J.A. Bergstra and P. Klint[BK95]. The ToolBus utilizes a scripting language based on process algebra[BW90] to describe the communication between software tools. A ToolBus *script* describes a number of processes that can communicate with each other and with tools living outside the ToolBus (Figure 1). A language-dependent adapter that translates between the internal ToolBus data format and the data format used by the individual tools makes it possible to write every tool in the language best suitable for the task(s) it has to perform.

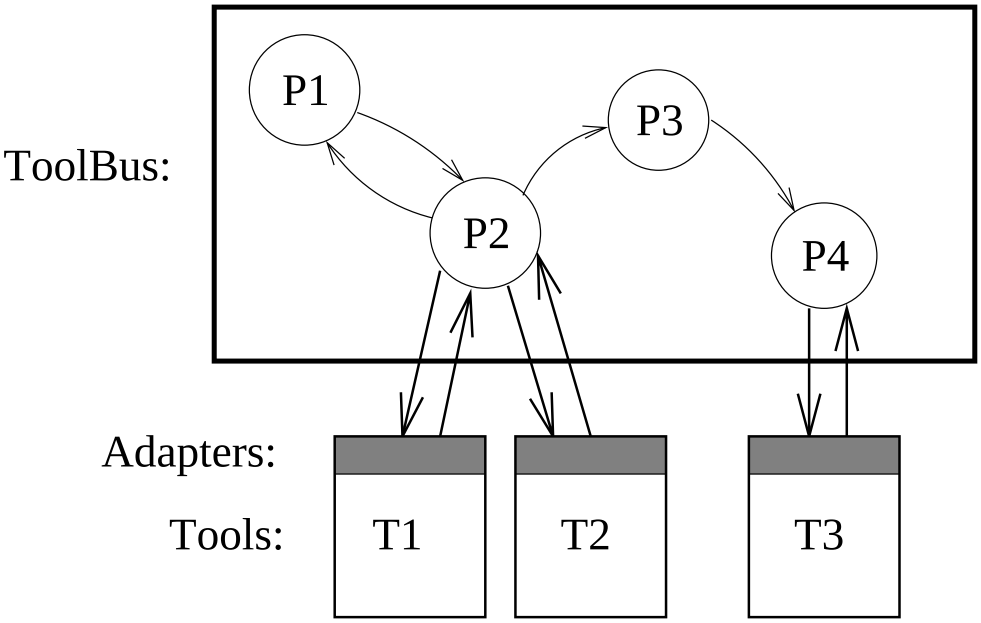

Figure 1: The ToolBus software application architecture

The following sections give an overview of the most important ToolBus features. For a more complete description of ToolBus expressions and primitives see appendix C.

2.1 Communication inside the ToolBus

There are two mechanisms available for processes in the ToolBus to communicate with each other, message passing and selective broadcasting. A process can synchronously send a *message* using the `snd-msg` primitive which must be received by another process using the `rec-msg` primitive. A process can send a *note* using `snd-note` to all processes that have subscribed, using `subscribe`, to that particular note type. The receiving processes read notes asynchronously using `rec-note`, at low priority. Transmitting notes amounts to *asynchronous selective broadcasting*.

2.2 Communication between ToolBus and tools

A ToolBus process can initiate communication with a tool by sending a message to a tool using `snd-do`, or `snd-eval` when an answer is expected. A tool can initiate communication by sending an *event* to the ToolBus. A ToolBus process receives this event using using the `rec-event` primitive and must acknowledge the event using the `snd-ack-event` primitive.

The execution and termination of the tools attached to the ToolBus, as well as their connection/disconnection can be controlled explicitly by appropriate primitives (See Section C.4).

2.3 Process composition

More complex processes can be created using process composition operators for *choice* (`+` operator), *sequential composition* (`.` operator), *parallel composition* (`||` operator), *iteration* (`*` operator) and *guarded (conditional) execution* (the `if/then/fi` operator). The *process creation* primitive `create` can be used to create new processes.

2.4 Types and variables

All terms within the ToolBus are *typed*. The ToolBus defines a number of basic types for booleans, integers, reals, strings, and binary strings. Complex types can be formed using a list constructor or function application. The type term is a supertype of all other types. The `let/in/endlet` construction makes it possible to declare variables.

2.5 A small example

To familiarize the reader with the syntax and semantics of ToolBus scripts, this section contains a small example. This example also familiarizes the reader with the use of noweb to present code fragments. Noweb is a system for literate programming as described in [Knu84], and is used to keep this document consistent with the actual system.

In this example, a ToolBus script is presented that connects two tools, a calculator tool that calculates expressions and a user interface tool that asks the user for expressions and presents the result of calculating that expression.

This ToolBus script contains two processes, a `UI` process that handles user interface events and a `CALC` process communicating with the calculator tool. The ToolBus configuration primitive `toolbus(UI,CALC)` starts the two processes in parallel.

⟨*calc.tb* ⟩≡
```
  ⟨process UI⟩
  ⟨process CALC⟩

  toolbus(UI,CALC)
```

Tool declarations make it possible for the ToolBus to find and execute tools. A tool declaration introduces a new type, that can later be used to declare tool identifier variables of that type.

⟨*process UI*⟩≡
```
  tool ui is { command = "wish-adapter -script ui.tcl" }
```

The `UI` process uses three variables. The first one, `UI`, is a tool identifier of type `ui`. The second variable `E` contains any expressions to be calculated, the third variable `V` contains the calculated result.

⟨*process UI*⟩+≡
```
  process UI is
  let
    UI : ui,
    E : str,
    V : str
  in
```

The `UI` process first starts the user interface tool and then handles `expr` events until a `quit` event is received.

⟨*process UI*⟩+≡
```
    execute(ui, UI?) .
    ( rec-event(UI, expr(E?)) .
      snd-msg(calc, expr(E)) .
      rec-msg(calc, expr(E, V?)) .
      snd-ack-event(UI, expr(E, V))
    ) *
    rec-event(UI, quit) .
    snd-ack-event(UI, quit) .
    shutdown("Goodbye!")
  endlet
```

This process also illustrates the use of the iteration using the binary Kleene star (`*`).

The `CALC` process starts the calculator tool and waits for calculation requests. It relays them to the calculator tool and sends the result back.

⟨*process CALC*⟩≡

```
tool calculator is { command = "calc" }

process CALC is
let
  Calc : calculator,
  E : str,
  V : str
in
  execute(calculator, Calc?) .
  ( rec-msg(calc, expr(E?)) .
    snd-eval(Calc, expr(E)) .
    rec-value(Calc, expr(E, V?)) .
    snd-msg(calc, expr(E, V))
  ) * delta
endlet
```

3 The UPI-10 project

There exists a wide range of devices equipped with a serial port for external communication. Examples include terminals, (radio-)modems, mobile printers, lcd displays, mobile phones, etc. Although the majority of these devices use the RS232 protocol at the lowest level, no consensus exists about which higher level protocol to use. Consequently, most manufacturers choose a protocol best suited for the applications they are interested in. Examples include the vt100 and ANSI escape codes for terminals, and a whole range of protocols for mobile communication, like MOBITEX, MAP 27, and RD LAP Ardis. The UPI-10 (Universal Protocol Interface) was designed as a universal way to interconnect these devices.

The UPI-10 is a device that offers three serial ports, a powerful microcontroller, 128Kb Flash ROM and 32Kb RAM. This makes it possible to interconnect three completely different RS232 devices[1] and program the UPI-10 to translate between the different protocols used by these devices.

3.1 The software

A piece of hardware like the UPI-10 is useless without a suitable development environment. Because the UPI-10 was meant to be a low cost product, using third party tools was not an option. We developed an assembler, a linker and a back end for *lcc*, a retargetable C compiler described in [FH95]. In addition, we implemented a runtime library containing a subset of the ANSI-C routines[KR88] and some communication protocols.

The only thing missing was a powerful simulator. The simulator framework described in this paper was used to implement such a simulator that has been used to debug and test both the development and the actual programs developed for the UPI-10.

3.2 The simulator

This section is used to give an impression of the functionality offered by the simulator we created, to help the reader understand the description of the framework architecture in the following sections. Figure 2 gives an overview of the user interface components of the simulator.

After starting the simulator, the user is confronted with the main *control* window, shown in the center of Figure 2. From here, the user can load executables and/or symbol tables and start up the other components, which can be divided into three categories:

- Assembler level components.
- Source level components.
- Communication components.

The assembler level debugging components can be used to inspect and interact with the system at the level of the CPU.

- The memory viewer is used to inspect and change the contents of the memory of the simulated UPI-10.
- The CPU viewer enables the user to inspect and change the contents of the CPU registers.
- The assembler viewer displays the disassembled memory contents. It allows the user to step through the assembler code and set and clear breakpoints.

The source level debugging components handle debugging at the source code level.

- The source viewer tracks the current source file and the current point of execution within this file. Breakpoints are highlighted and can be changed.
- The variable viewer tracks the contents of variables and lets the user change them.

The communication components simulate the communication between the UPI-10 and any connected RS232 devices. So far, we implemented two of these components:

- The communication spy keeps track of all the connections established in the system and shows a picture of these connections, together with a list of all the device types present in the system.
- The terminal emulator is the only RS232 device we developed so far. It emulates a simple terminal that can be connected to one of the UPI-10 serial ports or to another emulated terminal.

[1]even more when a serial multiplexer is used.

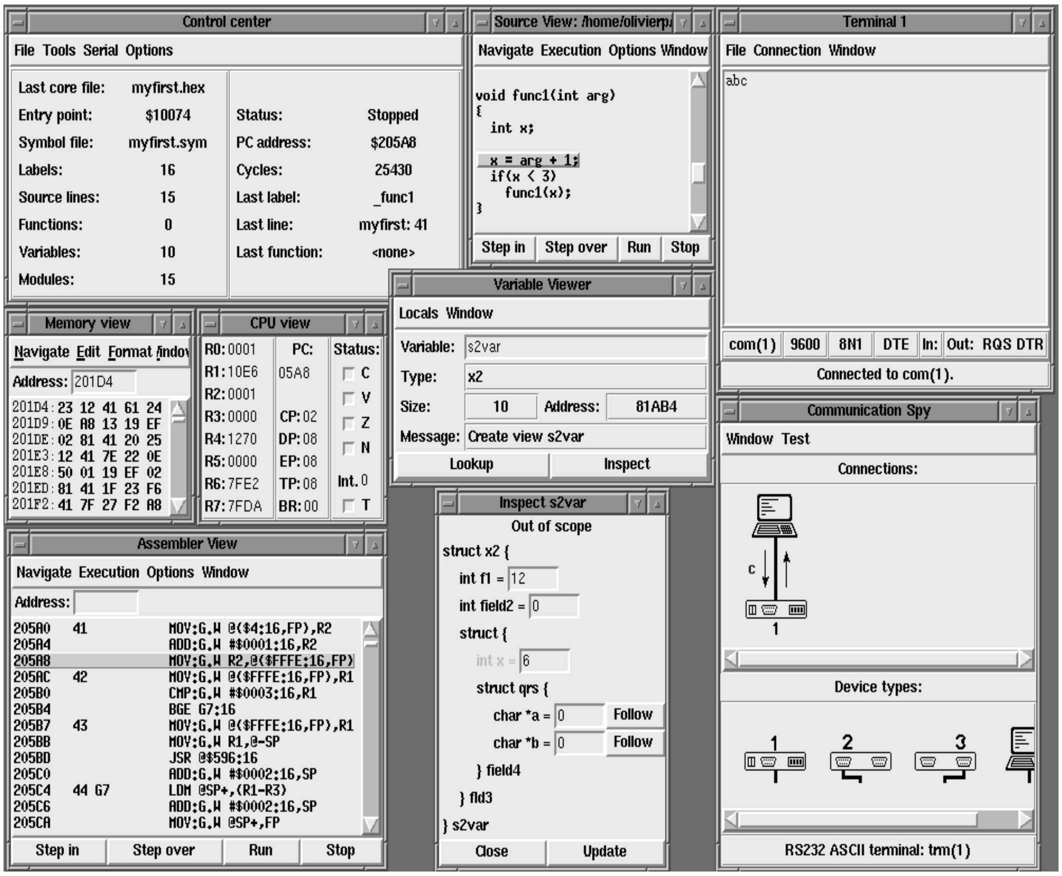

Figure 2: An overview of the simulator

4 Simulator architecture

Because we aim to create a simulator framework that is highly modular and contains a number of reusable components, the basic architecture should be general enough to handle a wide variety of computer systems. In this section, we will present our framework by outlining the tools and presenting the ToolBus script controlling the framework. The tools are considered to be of the *black box* variety, we will not discuss their implementation but only their input/output behaviour.

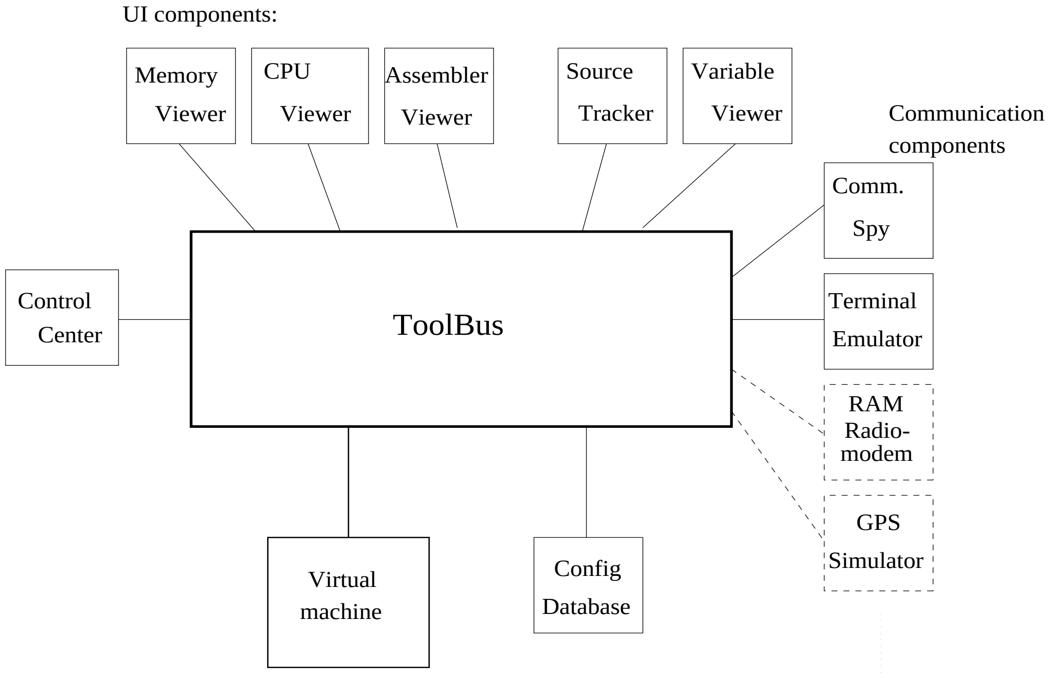

Figure 3: Simulator design: component overview

Figure 3 shows an overview of our design. We will discuss all the components that make up the design, by introducing the functionality they provide and presenting those fragments of the ToolBus script that control the component.

The basic outline of the controlling ToolBus script is presented below:

9a ⟨*sim.tb** 9a⟩≡
⟨*Initialize* 9b⟩
⟨*Configuration management* 33⟩
⟨*Virtual machine* 10⟩
⟨*Control center* 20a⟩
⟨*Memory viewer* 25⟩
⟨*CPU viewer* 22a⟩
⟨*Assembler viewer* 23a⟩
⟨*Source viewer* 26b⟩
⟨*Variable viewer* 27⟩
⟨*Communication spy* 30⟩
⟨*Terminal emulator* 31c⟩

```
toolbus(SIM-INIT)
```

This code is written to file `sim.tb`.

Every tool has its own section. The `Initialize` section starts the necessary processes to get things going. The simple structure of this section makes it easy to 'comment out' some of these processes during debugging, significantly decreasing the startup time.

9b ⟨*Initialize* 9b⟩≡ (9a)

```
process SIM-INIT is
let
  PVm       : int,
  PConfig   : int,
  PControl  : int,
  PMemView  : int, PCpuView  : int, PAsmView  : int,
  PSrcView  : int, PVarView  : int,
  PComSpy   : int, PTerminal : int
in
  create(CONFIG,  PConfig?) .
  create(VM, PVm?) .
  create(CONTROL, PControl?) .
  create(MEMVIEW, PMemView?) .
  create(CPUVIEW, PCpuView?) .
  create(ASMVIEW, PAsmView?) .
  create(SRCVIEW, PSrcView?) .
  create(VARVIEW, PVarView?) .
  create(COMSPY, PComSpy?) .
  create(TERMINAL, PTerminal?)
endlet
```

4.1 Virtual machine

Central to the architecture is the notion of a *virtual machine*. This component has no user interface, but takes care of the actual execution of simulated programs. The virtual machine must simulate all parts of the system that have to run at full speed, so no ToolBus communication is needed during the high-speed execution of a program section.[2] To accomplish this, the virtual machine has to keep track of at least the following information:

- CPU register contents;
- Memory contents;
- The status of any simulated special hardware needed during execution, like any switches, LED's, communication hardware etc;
- All breakpoint activity;
- Debug symbols (to map addresses to line numbers etc.);

In addition to the 'bookkeeping' tasks needed to maintain this information, the virtual machine performs the following operations:

- Simulate actual execution of machine language instructions;
- Disassemble the contents of memory on request;

The ToolBus code section controlling the virtual machine very much reflects the virtual machines responsibilities:

10 ⟨*Virtual machine* 10⟩≡ (9a)
  ⟨*VM tool definition* 11a⟩
  ⟨*Initialize VM* 11b⟩
  ⟨*CPU* 11d⟩
  ⟨*Memory* 12b⟩
  ⟨*UPI Hardware* 14a⟩
  ⟨*Breakpoints* 16c⟩
  ⟨*Symbols* 17a⟩
  ⟨*Execution* 19a⟩
  ⟨*Disassembly* 19c⟩

[2]We use the term 'program section' in this context to indicate a portion of the program to be executed without a need for visual feedback to the user. For instance, execution of the code generated by one line of source code or by an entire routine depending on the command issued by the user.

Because the efficiency of the *virtual machine tool* determines the actual execution speed of a simulated program, we have opted to implement it using C++.

11a ⟨*VM tool definition* 11a⟩≡ (10)

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

The initialization of the virtual machine is done by executing *vmtool* and starting the processes responsible for handling all virtual machine related tasks.

11b ⟨*Initialize VM* 11b⟩≡ (10) 11c ▷

```
process VM is
let
  Vm      : vmtool,
  PCpu    : int,    PMem    : int,
  PDis    : int,    PHard   : int,
  PBreaks : int,    PSyms   : int,
  PExec   : int
in
  execute(vmtool,  Vm?) .
  ( create(CPU(Vm), PCpu?) .
    create(MEMORY(Vm), PMem?) .
    create(UPI-HARDWARE(Vm), PHard?) .
    create(BREAKPOINTS(Vm), PBreaks?) .
    create(SYMBOLS(Vm), PSyms?) .
    create(EXECUTION(Vm), PExec?) .
    create(DISASM(Vm), PDis?)
    ||
```

Every ToolBus script is first pulled through the C preprocessor before being parsed. This opens up the possibility of using macros and using include files to get a basic form of modularity. In the simulator script, the preprocessor is used to define the macro `CONFIG_HANDLER` that defines a standard way to handle configuration management (see Section 4.10).

11c ⟨*Initialize VM* 11b⟩+≡ (10) ◁ 11b

```
    CONFIG_HANDLER(Vm)
  )
endlet
```

4.1.1 `CPU` process

The `CPU` process monitors the status of the CPU registers. It handles register change requests from other processes and broadcasts any changes made. The virtual machine must implement three special registers:

- The *program counter* (`pc`), which indicates the current point of execution.
- The *cycle counter* (`cycles`), that counts the number of CPU cycles used and can be used for profiling purposes.
- The *frame pointer* (`fp`), which points to the current stack frame. Using this register, tools can locate local (stack based) variables.

11d ⟨*CPU* 11d⟩≡ (10) 12a ▷

```
process CPU(Vm : vmtool) is
let
  Reg  : str,  Val  : int,
  Lo   : int,  Hi   : int,
  PC   : int,
  Regs : list, Pair : list,
  Stat : list,
  RegT : term, ValT : term
in
  ( rec-msg(cpu, set-register(Reg?, Val?)) .
    snd-do(Vm, set-register(Reg, Val))
    +
```

```
      rec-msg(cpu, set-registers(Regs?)) .
      snd-do(Vm, set-registers(Regs))
      +
      rec-msg(cpu, get-registers(Regs?)) .
      snd-eval(Vm, get-registers(Regs)) .
      rec-value(Vm, registers(Regs?)) .
      snd-msg(cpu, registers(Regs))
    ) * delta
    ||
    (
      rec-event(Vm, reg-changed(Reg?, Val?)) .
      snd-note(reg-changed(Reg, Val)) .
      snd-ack-event(Vm, reg-changed(Reg, Val))
      +
      rec-event(Vm, regs-changed(Regs?)) .
      snd-note(regs-changed(Regs)) .
      snd-ack-event(Vm, regs-changed(Regs))
      +
      rec-event(Vm, pc-changed(Val?)) .
      snd-note(pc-changed(Val)) .
      snd-ack-event(Vm, pc-changed(Val))
      +
      rec-event(Vm, fp-changed(Val?)) .
      snd-note(fp-changed(Val)) .
      snd-ack-event(Vm, fp-changed(Val))
      +
      rec-event(Vm, cycles-changed(Val?)) .
      snd-note(cycles-changed(Val)) .
      snd-ack-event(Vm, cycles-changed(Val))
      +
```

When the execution status of the virtual machine changes, for instance from 'stopped' to 'running', it generates a `status` event to notify the control center.

12a ⟨*CPU* 11d⟩+≡ (10) ◁11d

```
      rec-event(Vm, status(Stat?)) .
      snd-msg(control, status(Stat)) .
      rec-msg(control, status) .
      snd-ack-event(Vm, status(Stat))
    ) * delta
  endlet
```

4.1.2 Memory issues

The `MEMORY` process controls changes in the memory of the simulator. It receives change requests from other processes and broadcasts any changes made. Changes of different granularity are allowed.

12b ⟨*Memory* 12b⟩≡ (10) 12c▷

```
  process MEMORY(Vm : vmtool) is
  let
    Adr   : int,  Val     : int,
    Start : int,  End     : int,  Mem     : str,
    Blocks: list, Result  : list,
    File  : str,  Stat    : list
  in
```

The value of a byte at an address where no physical memory is present is always considered to be $0xFF$, exactly like in the hardware case. The smallest granularity of memory access possible is one byte:

12c ⟨*Memory* 12b⟩+≡ (10) ◁12b 13a▷

```
  ( rec-msg(memory, get-byte(Adr?)) .
    snd-eval(Vm, get-byte(Adr)) .
    rec-value(Vm, byte(Adr, Val?)) .
    snd-msg(memory, get-byte(Adr, Val))
    +
    rec-msg(memory, set-byte(Adr?, Val?)) .
    snd-do(Vm, set-byte(Adr, Val))
    +
```

The most common granularity is one word (two bytes). This is also the smallest size that is broadcasted when a change has taken place.

13a ⟨*Memory* 12b⟩+≡ (10) ◁12c 13b▷

```
    rec-msg(memory, get-word(Adr?)) .
    snd-eval(Vm, get-word(Adr)) .
    rec-value(Vm, word(Adr, Val?)) .
    snd-msg(memory, get-word(Adr, Val))
    +
    rec-msg(memory, set-word(Adr?, Val?)) .
    snd-do(Vm, set-word(Adr, Val))
    +
    rec-event(Vm, word-changed(Adr?, Val?)) .
    snd-note(word-changed(Adr, Val)) .
    snd-ack-event(Vm, word-changed(Adr, Val))
    +
```

Some tools need to retrieve a block of memory. To avoid a whole series of requests we introduce the `get-block` request.

13b ⟨*Memory* 12b⟩+≡ (10) ◁13a 13c▷

```
    rec-msg(memory, get-block(Start?, End?)) .
    snd-eval(Vm, get-block(Start, End)) .
    rec-value(Vm, block(Start, End, Mem?)) .
    snd-msg(memory, block(Start, End, Mem))
    +
```

If a process needs the value of a number of non-consecutive memory blocks it can use the `get-block-list` request.

13c ⟨*Memory* 12b⟩+≡ (10) ◁13b 13d▷

```
    rec-msg(memory, get-block-list(Blocks?)) .
    snd-eval(Vm, get-block-list(Blocks)) .
    rec-value(Vm, block-list(Result?)) .
    snd-msg(memory, block-list(Result))
    +
```

Another source of changes to the simulator memory is the loading of a core file. This causes the virtual machine to generate a `mem-changed` event, indicating the memory area that has changed. This makes it possible for any tool to request only those bytes of memory it actually needs.

13d ⟨*Memory* 12b⟩+≡ (10) ◁13c

```
    rec-msg(vm, load-core(File?)) .
    snd-eval(Vm, load-core(File)) .
    rec-value(Vm, core-loaded(File, Stat?)) .
    snd-msg(vm, core-loaded(File, Stat))
    +
    rec-event(Vm, mem-changed(Start?, End?)) .
    snd-note(mem-changed(Start, End)) .
    snd-ack-event(Vm, mem-changed(Start, End))
  ) * delta
endlet
```

4.1.3 Hardware woes

When developing a simulator component for a new piece of hardware, the designer is left with the following tasks:

- Design a suitable ToolBus interface for the new component.
- Implement (only) those parts of the component that have to run at top speed in the virtual machine. Preferably, this would exclude the user interface part.
- Implement the remaining part of the component in a separate tool.

4.1.4 RS232 hardware

Sometimes it is possible to design the interface for a whole class of hardware components, as is the case for the UPI-10. The special hardware in this case mainly consists of the communication hardware needed to drive the three RS232 ports.

Because the communication between the UPI-10 and other RS232 devices is several orders of magnitude slower than the communication between components within the UPI-10 (for instance between the CPU and the memory subsystem, or between the CPU and the communication hardware), the RS232 devices can be implemented as separate tools. By providing a general RS232 protocol simulation interface in the ToolBus script, and by implementing the UPI-10 communication hardware in the virtual machine tool, we can implement every single RS232 device as a separate tool. All that is needed to incorporate a new RS232 device in the simulator is to implement it as a separate tool, which is not a big investment for most devices. At this point we have only done this for a simple terminal emulator (see Section 4.9).[3]

In the ToolBus RS232 protocol, we have included some simple logging facilities to be able to track the communication that occurs. These facilities are used by the *communication spy* (Section 4.8) to provide a visual display of the communication between the different RS232 devices present.

Every RS232 device needs a process that handles the protocol at the ToolBus side. All these processes have roughly the same structure, shown below for the UPI-10 case.

14a ⟨*UPI Hardware* 14a⟩≡ (10)
  ⟨*Init UPI serial ports* 14b⟩
  ⟨*UPI serial communication* 15a⟩

Initializing a RS232 device amounts to registering its presence with the system, and starting the process that takes care of the actual communication. In the UPI-10 case, we do this three times, once for every port.

14b ⟨*Init UPI serial ports* 14b⟩≡ (14a)
```
process UPI-HARDWARE(Vm : vmtool) is
let
  Port1 : int, Port2 : int, Port3 : int,
  Pins  : int
in
  snd-msg(comspy, register-device(com(1), "UPI COM1", "clipart/upi1-a.xbm",
          "clipart/upi1-b.xbm", "UPI RS232 Port 1")) .
  snd-msg(comspy, register-device(com(2), "UPI COM2", "clipart/upi2-a.xbm",
          "clipart/upi2-b.xbm", "UPI RS232 Port 2")) .
  snd-msg(comspy, register-device(com(3), "UPI COM3", "clipart/upi3-a.xbm",
          "clipart/upi3-b.xbm", "UPI RS232 Port 3")) .
  create(UPI-PORT(Vm,1), Port1?) .
  create(UPI-PORT(Vm,2), Port2?) .
  create(UPI-PORT(Vm,3), Port3?)
endlet
```

The registration mechanism is explained in Section 4.8. Here we just assume that some process waits for `comspy` messages.

The RS232 protocol is for serial communication. To achieve the highest level of realism, we would actually have to send bits. Because this would seriously affect the performance (effectively slowing it

[3] We are planning to write tools for some other devices, including a serial printer, a radio-modem, GPS (vehicle location) hardware and a hand-held terminal.

down by a factor of eight or more), and because most applications do not need this kind of realism, we have opted to send *characters* instead. To cope with format errors, like wrong parity and framing errors we send along the format information with every character so applications can check for errors. We also keep track of the pin status of both the remote and local ports to be able to check valid DTE/DCE configurations and other RS232 features like the RING indicator.

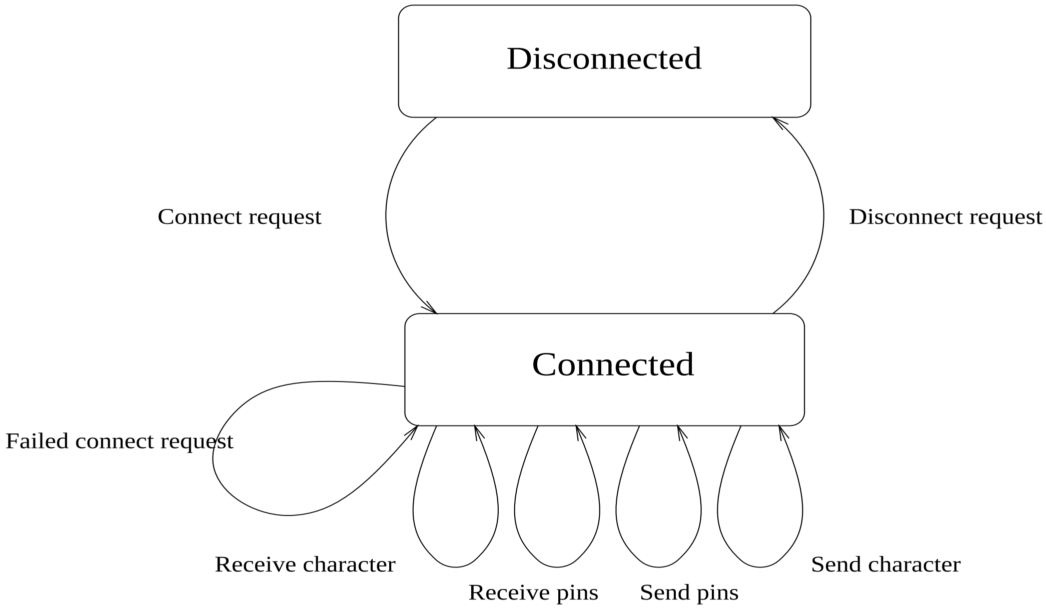

Figure 4: State diagram of RS232 protocol simulation

The protocol used to simulate RS232 devices is pictured by a simple state diagram (Figure 4). The implementation of this protocol for the UPI-10 ports is captured in the process `UPI-PORT`:

15a ⟨*UPI serial communication* 15a⟩≡ (14a) 15b ▷

```
process UPI-PORT(Vm : vmtool, Port : int) is
let
  Char    : str,
  PinsIn  : int,  PinsIn2 : int,  PinsOut : int,
  Baud    : int,  Fmt     : str,  PinsOut2: int,
  Remote  : term, Remote2 : term
in
```

We start in the *Disconnected* state. When a connection request is received, we enter the *Connected* state. To enable logging, we send a note signalling a new connection. Notice that the UPI-10 never initiates a connection, because we choose to connect devices to the UPI-10 rather than connecting the UPI-10 to some device.

15b ⟨*UPI serial communication* 15a⟩+≡ (14a) ◁ 15a 15c ▷

```
  PinsOut := 0 .
  ( rec-msg(connect(Remote?, com(Port), PinsIn?)) .
    snd-do(Vm, pins(Port, PinsIn)) .
    snd-msg(connected(Remote, com(Port), PinsOut)) .
    snd-note(connection(Remote, com(Port), PinsIn)) .
```

When connected, another connection request should be refused.

15c ⟨*UPI serial communication* 15a⟩+≡ (14a) ◁ 15b 16a ▷

```
    ( rec-msg(connect(Remote2?, com(Port), PinsIn2?)) .
      snd-msg(connection-refused(Remote2, com(Port)))
      +
```

During a connection, we can receive pin indications and characters sent by the remote device. The UPI-10 can also initiate transmission.

16a ⟨*UPI serial communication* 15a⟩+≡ (14a) ◁15c 16b▷

```
        rec-msg(pins(Remote, com(Port), PinsIn?)) .
        snd-do(Vm, pins(Port, PinsIn))
        +
        rec-msg(send-char(Remote, com(Port), Char?, Baud?, Fmt?)) .
        snd-do(Vm, send-char(Port, Char, Baud, Fmt))
        +
        rec-event(Vm, char-to-send(Port, Char?, Baud?, Fmt?)) .
        snd-msg(send-char(com(Port), Remote, Char, Baud, Fmt)) .
        snd-note(send-char(com(Port), Remote, Char, Baud, Fmt)) .
        snd-ack-event(Vm, char-to-send(Port, Char, Baud, Fmt))
        +
        rec-event(Vm, pins(Port, PinsOut?)) .
        snd-msg(pins(com(Port), Remote, PinsOut)) .
        snd-ack-event(Vm, pins(Port, PinsOut))
```

Only the remote device can initiate a disconnect. The UPI-10 is just a humble slave. Do not forget to send a note signalling the termination of this connection.

16b ⟨*UPI serial communication* 15a⟩+≡ (14a) ◁16a

```
      ) *
      rec-msg(disconnect(Remote, com(Port))) .
      snd-note(disconnected(Remote, com(Port)))
      +
      rec-event(Vm, pins(Port, PinsOut?)) .
      snd-ack-event(Vm, pins(Port, PinsOut))
    ) * delta
  endlet
```

The implementation of an 'active' RS232 device that can also initiate connects and disconnects can be found in Section 4.9, where we describe a virtual terminal component.

4.1.5 Breakpoints

To facilitate debugging at two levels, the assembler level and the source level, we allow breakpoints at addresses and lines. A line breakpoint is translated by the virtual machine into several address breakpoints to compensate for the fact that one source line can have more than one entry point. For instance the code generated for a C 'for' loop generally has an initialization part and a body part and it is desirable to break on both.

When the breakpoints change, a note is send containing all the breakpoints and their addresses.

16c ⟨*Breakpoints* 16c⟩≡ (10)

```
  process BREAKPOINTS(Vm : vmtool) is
  let
    Adr           : int,
    Breaks        : list,
    Module        : str,
    Line          : int
  in
   ( rec-msg(breakpoints, toggle(Adr?)) .
     snd-do(Vm, toggle-break(Adr))
     +
     rec-msg(breakpoints, clear-all) .
     snd-do(Vm, clear-all-breaks)
     +
     rec-msg(breakpoints, line-off(Module?, Line?)) .
     snd-do(Vm, line-break-off(Module, Line))
     +
     rec-msg(breakpoints, line-on(Module?, Line?)) .
```

```
    snd-do(Vm, line-break-on(Module, Line))
    +
    rec-msg(breakpoints, get-breaks) .
    snd-eval(Vm, get-breaks) .
    rec-value(Vm, breakpoints(Breaks?)) .
    snd-msg(breakpoints, breakpoints(Breaks))
    +
    rec-event(Vm, breakpoints(Breaks?)) .
    snd-note(breaks-changed(Breaks)) .
    snd-ack-event(Vm, breakpoints(Breaks))
  ) * delta
endlet
```

4.1.6 Debug symbols

Many tools need information about debug symbols from the symbol database maintained by the virtual machine. The SYMBOLS process handles requests for symbol information.

17a ⟨*Symbols* 17a⟩≡ (10)

```
process SYMBOLS(Vm : vmtool) is
let
  Stat  : list,
  Label : str, Adr    : int,
  File  : str, Line   : int,
  Var   : str, Scopes : list(<int>),
  Size  : int, Type   : str,  TypeList : str,
  Scope : int, Off    : int,  Result   : list,
  Locals: list
in
  (
    ⟨Symbol files 17b⟩
    +
    ⟨Label info 17c⟩
    +
    ⟨Source line info 17d⟩
    +
    ⟨Variable info 18a⟩
  ) * delta
endlet
```

The virtual machine loads debug information on demand.

17b ⟨*Symbol files* 17b⟩≡ (17a)

```
rec-msg(vm, load-sym(File?)) .
snd-eval(Vm, load-sym(File)) .
rec-value(Vm, sym-loaded(File, Stat?)) .
snd-msg(vm, sym-loaded(File, Stat))
```

The only information about a label you can retrieve, is its address.

17c ⟨*Label info* 17c⟩≡ (17a)

```
rec-msg(label, get-label-address(Label?)) .
snd-eval(Vm, get-label-address(Label)) .
rec-value(Vm, label-address(Label, Adr?)) .
snd-msg(label, label-address(Label, Adr))
```

The same goes for source lines.

17d ⟨*Source line info* 17d⟩≡ (17a)

```
rec-msg(label, get-line-address(File?, Line?)) .
snd-eval(Vm, get-line-address(File, Line)) .
rec-value(Vm, line-address(File, Line, Adr?)) .
snd-msg(label, line-address(File, Line, Adr))
```

Variable information is more complex. First of all, the scope from which you want to retrieve information is important, so the information request is accompanied by a variable name and optionally a list of scopes to search. We assume every scope is identified by a unique integer.

18a ⟨*Variable info* 18a⟩≡ (17a) 18b ▷

```
( rec-msg(label, lookup-var(Var?)) .
  snd-eval(Vm, lookup-global(Var))
  +
  rec-msg(label, lookup-var(Var?, Scopes?)) .
  snd-eval(Vm, lookup-local(Var, Scopes))
) .
```

When a variable with the indicated name cannot be found, `no-var(Var)` is returned (with Var the requested variable name).

18b ⟨*Variable info* 18a⟩+≡ (17a) ◁ 18a 18c ▷

```
( rec-value(Vm, no-var(Var?)) .
  snd-msg(label, no-var(Var))
  +
```

When the indicated variable has file scope, the variable is returned, together with its address, size, and type. The type of the variable is returned in two formats: a 'C' type declaration and a string that represents the type as a nested Tcl list. The last format is a small concession to keep the handling of variables by Tcl tools as simple as possible.

18c ⟨*Variable info* 18a⟩+≡ (17a) ◁ 18b 18d ▷

```
  rec-value(Vm, global-var(Var?, Adr?, Size?, Type?, TypeList?)) .
  snd-msg(label, global-var(Var, Adr, Size, Type, TypeList))
  +
```

When the variable is a local static, the information returned includes its scope:

18d ⟨*Variable info* 18a⟩+≡ (17a) ◁ 18c 18e ▷

```
  rec-value(Vm, static-var(Var?, Scope?, Adr?, Size?, Type?, TypeList?)) .
  snd-msg(label, static-var(Var, Scope, Adr, Size, Type, TypeList))
  +
```

The address of a local stack variable depends on the location of the current stack frame, so it is indicated by an offset from that stack frame. The location of the current stack frame is indicated by the special `frame-pointer` register.

18e ⟨*Variable info* 18a⟩+≡ (17a) ◁ 18d 18f ▷

```
  rec-value(Vm, local-var(Var?,Scope?,Off?,Size?,Type?,TypeList?)) .
  snd-msg(label, local-var(Var, Scope, Off, Size, Type, TypeList))
)
+
```

A tool might want to investigate which local variables are present in a list of scopes, to present the user with this list and let him choose one.

18f ⟨*Variable info* 18a⟩+≡ (17a) ◁ 18e

```
  rec-msg(label, get-locals(Scopes?)) .
  Result := [] .
  ( if not-equal(Scopes, []) then
      Scope := first(Scopes) .
      Scopes := next(Scopes) .
      snd-eval(Vm, get-locals(Scope)) .
      rec-value(Vm, locals(Locals?)) .
      Result := join(Result, Locals)
    fi
  ) * if equal(Scopes, []) then tau fi .
  snd-msg(label, locals(Result))
```

4.1.7 Program execution

When the execution of a program section is requested, the break conditions must be specified. First of all, execution always stops when a breakpoint is reached. Other stop conditions can be indicated by a level (`asm` or `src`), and an action (`step-in`, `step-over` or `run`). The 'run' action means there are no extra break conditions. The program just runs until a breakpoint is reached. At the assembler level, 'step in' just executes one assembler instruction and then stops. 'step over' executes one instruction, but if this instruction happens to be a subroutine call, the whole routine is executed. At the source code level, 'step in' executes the program until the next source line is reached. 'step over' does the same, executing any function calls encountered.

19a ⟨*Execution* 19a⟩≡ (10) 19b ▷

```
process EXECUTION(Vm : vmtool) is
let
  Level  : term,
  Action : term,
  Status : term,
  Module : str,
  Line   : int,
  Scopes : list(<int>)
in
  ( rec-msg(exec, Level?, Action?) .
    snd-do(Vm, exec(Level, Action))
    +
    rec-msg(exec, user-break) .
    snd-do(Vm, user-break)
    +
    rec-msg(exec, reset) .
    snd-do(Vm, reset-execution)
    +
    rec-msg(exec, get-exec-status) .
    snd-eval(Vm, get-exec-status) .
    rec-value(Vm, exec-status(Status?)) .
    snd-msg(exec, exec-status(Status))
  ) * delta
  ||
```

When execution stops, the current scope and source lines might have changed. If this is the case, the virtual machine generates an event indicating the new source line and/or *scope set*. A scope set is a list of the current and enclosing scopes with the smallest scope listed first.

19b ⟨*Execution* 19a⟩+≡ (10) ◁ 19a

```
  ( rec-event(Vm, line-number(Module?, Line?)) .
    snd-note(line-number(Module, Line)) .
    snd-ack-event(Vm, line-number(Module, Line))
    +
    rec-event(Vm, scopes-changed(Scopes?)) .
    snd-note(scopes-changed(Scopes)) .
    snd-ack-event(Vm, scopes-changed(Scopes))
  ) * delta
endlet
```

4.1.8 Memory disassembly

When a tool wants to show the user an assembler listing, it can send a request to disassemble part of the memory of the simulator. The virtual machine disassembles the number of instructions requested and returns them in a list.

19c ⟨*Disassembly* 19c⟩≡ (10)

```
process DISASM(Vm : vmtool) is
let
  Adr     : int,
  Count   : int,
  AsmList : list
in
```

```
    ( rec-msg(disasm, disassemble-list(Adr?, Count?)) .
      snd-eval(Vm, disassemble-list(Adr, Count)) .
      rec-value(Vm, disassembly-list(Adr?, Count?, AsmList?)) .
      snd-msg(disasm, disassembly-list(Adr, Count, AsmList))
    ) * delta
  endlet
```

4.2 Control center

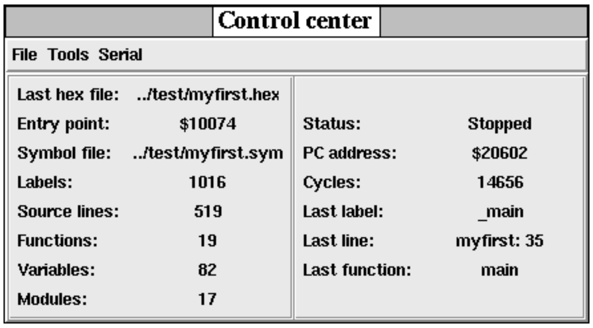

Figure 5: The simulator control center

The *control center* (Figure 5) provides a status display showing a summary of the current state of the simulator. It also enables the user to load/save configuration files, to load programs and/or symbol tables, and to create instances of the different viewers described in the following sections.

20a ⟨*Control center* 20a⟩≡ (9a) 20b ▷

```
  tool control is { command = "wish-adapter -script control.tcl" }

  process CONTROL is
  let
    Ctrl : control,
    A    : int,
    Stat : list,
    File : str,
    Msg  : str,
    Tool : term
  in
    execute(control, Ctrl?) .
```

The user can load/save the current configuration from within the control center.

20b ⟨*Control center* 20a⟩+≡ (9a) ◁ 20a 21a ▷

```
    ( rec-event(Ctrl, load-config(File?)) .
      snd-msg(config, load-config(File)) .
      ( rec-msg(config, config-loaded(File))
        +
        rec-msg(config, error(Msg?)) .
        snd-do(Ctrl, error-msg("Error while loading config file", Msg))
      ) .
      snd-ack-event(Ctrl, load-config(File))
      +
      rec-event(Ctrl, save-config(File?)) .
      snd-msg(config, save-config(File)) .
      ( rec-msg(config, config-saved(File))
        +
        rec-msg(config, error(Msg?)) .
        snd-do(Ctrl, error-msg("Error while saving config file", Msg))
```

```
) .
snd-ack-event(Ctrl, save-config(File))
+
```

The user can also load/save symbol and core files. A symbol file contains all the symbols of the simulated program, a core file contains the actual executable program.

21a ⟨*Control center* 20a⟩+≡ (9a) ◁20b 21b▷

```
rec-event(Ctrl, load-sym(File?)) .
snd-msg(vm, load-sym(File)) .
( rec-msg(vm, sym-loaded(File, Stat?)) .
  snd-do(Ctrl, sym-loaded(File, Stat))
  +
  rec-msg(vm, error(Msg?)) .
  snd-do(Ctrl, error-msg("Error while loading symbol file", Msg))
) .
snd-ack-event(Ctrl, load-sym(File))
+
rec-event(Ctrl, load-core(File?)) .
snd-msg(vm, load-core(File)) .
( rec-msg(vm, core-loaded(File, Stat?)) .
  snd-do(Ctrl, core-loaded(File, Stat))
  +
  rec-msg(vm, error(Msg?)) .
  snd-do(Ctrl, error-msg("Error while loading core image", Msg))
) .
snd-ack-event(Ctrl, load-core(File))
+
```

When the execution status of the virtual machine changes, the process `CPU` (See section 4.1.1) sends a `status` message, so the control center can show the change to the user.

21b ⟨*Control center* 20a⟩+≡ (9a) ◁21a 21c▷

```
rec-msg(control, status(Stat?)) .
snd-eval(Ctrl, status(Stat)) .
rec-value(Ctrl, status) .
snd-msg(control, status)
+
```

A `create-view` event is generated when the user selects one of the tools from the `Tools` menu in the control center.

21c ⟨*Control center* 20a⟩+≡ (9a) ◁21b 21d▷

```
rec-event(Ctrl, create-view(Tool?)) .
snd-msg(Tool, create-view) .
snd-ack-event(Ctrl, create-view(Tool))
+
```

To make it possible for other processes to initiate file loading, we handle some additional messages. This opens up the possibility of using extra ToolBus scripts to automate certain tasks. We have used this feature extensively during debugging/testing.

21d ⟨*Control center* 20a⟩+≡ (9a) ◁21c

```
rec-msg(control, load-sym(File?)) .
snd-msg(vm, load-sym(File)) .
rec-msg(vm, sym-loaded(File, Stat?)) .
snd-do(Ctrl, sym-loaded(File, Stat)) .
snd-msg(control, sym-loaded)
+
rec-msg(control, load-core(File?)) .
snd-msg(vm, load-core(File)) .
rec-msg(vm, core-loaded(File, Stat?)) .
snd-do(Ctrl, core-loaded(File, Stat)) .
snd-msg(control, core-loaded)
```

```
  ) * delta
endlet
```

4.3 CPU viewer

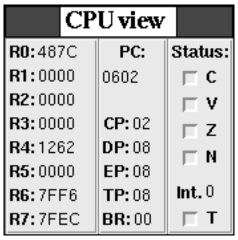

Figure 6: The CPU viewer

The *CPU viewer* (Figure 6) provides a view on the current register contents. It lists the registers and their values, and enables the user to edit them.

22a ⟨*CPU viewer* 22a⟩≡ (9a) 22b ▷

```
tool cpuview is { command = "wish-adapter -script cpuview.tcl" }

process CPUVIEW is
let
  Cpu  : cpuview,
  Reg  : str,  Val  : int,
  Regs : list, Result : list
in
  execute(cpuview, Cpu?) .
  subscribe(reg-changed(<str>, <int>)) .
  subscribe(regs-changed(<list>)) .
  ( rec-msg(cpuview, create-view) .
    snd-do(Cpu, create-view)
    +
```

When the CPU viewer wants to update the registers in the virtual machine, it generates a `set-register` event.

22b ⟨*CPU viewer* 22a⟩+≡ (9a) ◁ 22a 22c ▷

```
    rec-event(Cpu, set-registers(Regs?)) .
    snd-msg(cpu, set-registers(Regs)) .
    snd-ack-event(Cpu, set-registers(Regs))
    +
```

When it wants to retrieve the register values, the CPU viewer generates a `get-registers` event.

22c ⟨*CPU viewer* 22a⟩+≡ (9a) ◁ 22b 22d ▷

```
    rec-event(Cpu, get-registers(Regs?)) .
    snd-msg(cpu, get-registers(Regs)) .
    rec-msg(cpu, registers(Result?)) .
    snd-do(Cpu, registers(Result)) .
    snd-ack-event(Cpu, get-registers(Regs))
    +
```

The `reg-changed` and `regs-changed` notes are send by the `CPU` to indicate that one or more registers have changed.

22d ⟨*CPU viewer* 22a⟩+≡ (9a) ◁ 22c

```
    rec-note(reg-changed(Reg?, Val?)) .
    snd-do(Cpu, register(Reg, Val))
    +
    rec-note(regs-changed(Regs?)) .
```

```
    snd-do(Cpu, registers(Regs))
  ) * delta
endlet
```

4.4 Assembler viewer

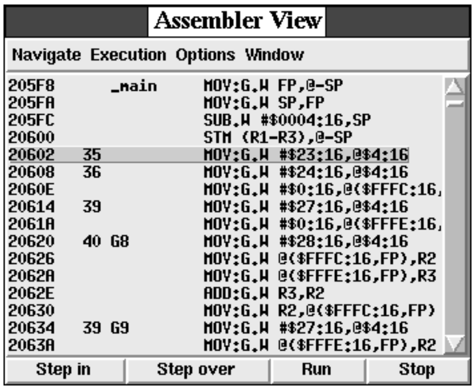

Figure 7: The assembler viewer

The *assembler viewer* (Figure 7) shows the simulator memory in disassembled form. The current point of execution is highlighted, and so are any breakpoints.

23a ⟨*Assembler viewer* 23a⟩≡ (9a) 23b▷

```
tool asmview is { command = "wish-adapter -script asmview.tcl" }

process ASMVIEW is
let
  Asm    : asmview,
  Name   : str,  Start   : int,  End   : int,
  Adr    : int,  ID      : int,  Count : int,
  Val    : int,  Label   : str,
  Block  : str,  AsmList : list,
  Level  : term, Action  : term,
  Breaks : list
in
  execute(asmview, Asm?) .
  subscribe(mem-changed(<int>, <int>)) .
  subscribe(word-changed(<int>, <int>)) .
  subscribe(pc-changed(<int>)) .
  subscribe(breaks-changed(<list>)) .
  ( rec-msg(asmview, create-view) .
    snd-do(Asm, create-view)
    +
```

When the memory of the simulated device has changed, the assembler viewer is notified by a `mem-changed` or `word-changed` note, so it can update its view on the assembler code accordingly.

23b ⟨*Assembler viewer* 23a⟩+≡ (9a) ◁23a 24a▷

```
    rec-note(mem-changed(Start?, End?)) .
    snd-do(Asm, mem-changed(Start, End))
    +
    rec-note(word-changed(Adr?, Val?)) .
    snd-do(Asm, word-changed(Adr, Val))
    +
    rec-note(pc-changed(Adr?)) .
    snd-do(Asm, pc-changed(Adr))
```

```
+
```

The assembler viewer asks the virtual machine to disassemble a list of instructions by generating an event. When the result comes back, the assembler viewer can present the disassembled code to the user.

24a ⟨*Assembler viewer* 23a⟩+≡ (9a) ◁23b 24b▷

```
rec-event(Asm, disassemble-list(ID?, Adr?, Count?)) .
snd-msg(disasm, disassemble-list(Adr, Count)) .
rec-msg(disasm, disassembly-list(Adr, Count?, AsmList?)) .
snd-do(Asm, disassembly-list(ID, Adr, Count, AsmList)) .
snd-ack-event(Asm, disassemble-list(ID, Adr, Count))
+
```

The assembler viewer needs to keep track of the breakpoints in order to highlight any lines containing a breakpoint.

24b ⟨*Assembler viewer* 23a⟩+≡ (9a) ◁24a 24c▷

```
rec-note(breaks-changed(Breaks?)) .
snd-do(Asm, breaks-changed(Breaks))
+
rec-event(Asm, toggle-break(Adr?)) .
snd-msg(breakpoints, toggle(Adr)) .
snd-ack-event(Asm, toggle-break(Adr))
+
rec-event(Asm, list-breaks) .
snd-msg(breakpoints, list) .
snd-ack-event(Asm, list-breaks)
+
rec-event(Asm, clear-breakpoints) .
snd-msg(breakpoints, clear-all) .
snd-ack-event(Asm, clear-breakpoints)
+
```

The user can control the execution from within the assembler viewer. The assembler viewer generates the apropriate events on behave of the user.

24c ⟨*Assembler viewer* 23a⟩+≡ (9a) ◁24b 24d▷

```
rec-event(Asm, exec(Level?, Action?)) .
snd-msg(exec, Level, Action) .
snd-ack-event(Asm, exec(Level,Action))
+
rec-event(Asm, reset-execution) .
snd-msg(exec, reset) .
snd-ack-event(Asm, reset-execution)
+
rec-event(Asm, user-break) .
snd-msg(exec, user-break) .
snd-ack-event(Asm, user-break)
+
```

The nav-label event is used to navigate the current focus in the assembler view window to a different portion of memory, the address of which is indicated by a label. The address of the label is retrieved and the focus is moved accordingly.

24d ⟨*Assembler viewer* 23a⟩+≡ (9a) ◁24c

```
rec-event(Asm, nav-label(ID?, Label?)) .
snd-msg(label, get-label-address(Label)) .
rec-msg(label, label-address(Label, Adr?)) .
if not(less(Adr, 0)) then
  snd-do(Asm, navigate(ID, Adr))
else
  tau
fi .
```

```
    snd-ack-event(Asm, nav-label(ID, Label))
  ) * delta
endlet
```

4.5 Memory viewer

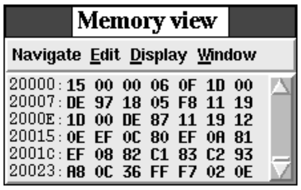

Figure 8: The memory viewer

To watch and/or change the raw contents of the memory of the simulated device, the *memory viewer* (Figure 8) can be used. It can show the memory contents in five different formats: binary, octal, decimal, hexadecimal or ASCII.

25 ⟨*Memory viewer* 25⟩≡ (9a) 26a ▷

```
tool memview is { command = "wish-adapter -script memview.tcl" }

process MEMVIEW is
let
  Mem   : memview,
  Adr   : int, Val   : int,
  Start : int, End   : int,
  ID    : int, Name  : str,
  Block : str, Label : str
in
  execute(memview, Mem?) .
  subscribe(word-changed(<int>, <int>)) .
  subscribe(mem-changed(<int>, <int>)) .
  ( rec-msg(memview, create-view) .
    snd-do(Mem, create-view)
    +
    rec-note(word-changed(Adr?, Val?)) .
    snd-do(Mem, word-changed(Adr, Val))
    +
    rec-note(mem-changed(Start?, End?)) .
    snd-do(Mem, mem-changed(Start, End))
    +
    rec-event(Mem, get-byte(Adr?)) .
    snd-msg(memory, get-byte(Adr?)) .
    rec-msg(memory, byte(Adr, Val?)) .
    snd-ack-event(Mem, byte(Adr, Val))
    +
    rec-event(Mem, get-block(ID?, Start?, End?)) .
    snd-msg(memory, get-block(Start, End)) .
    rec-msg(memory, block(Start?, End?, Block?)) .
    snd-do(Mem, block(ID, Start, End, Block)) .
    snd-ack-event(Mem,get-block(ID, Start, End))
    +
    rec-event(Mem, set-byte(Adr?, Val?)) .
    snd-msg(memory, set-byte(Adr, Val))
    +
```

Just as with the assembler viewer, the nav-label event is used to focus another part of the UPI-10 memory.

26a ⟨*Memory viewer* 25⟩+≡ (9a) ◁25

```
      rec-event(Mem, nav-label(ID?, Label?)) .
      snd-msg(label, get-label-address(Label)) .
      rec-msg(label, label-address(Label, Adr?)) .
      if not(less(Adr, 0)) then
        snd-do(Mem, navigate(ID, Adr))
      else
        tau
      fi .
      snd-ack-event(Mem, nav-label(ID, Label))
    ) * delta
  endlet
```

4.6 Source viewer

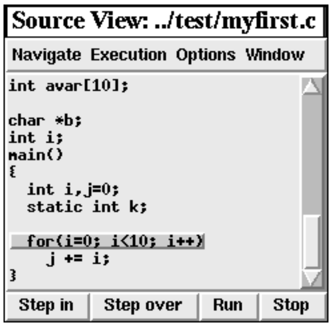

Figure 9: The source tracker

The *source viewer* (Figure 9) enables the user to watch the execution of a program at the source code level. The current point of execution is shown by highlighting the appropriate source line. Breakpoints are highlighted and can be changed by double clicking on a source line.

26b ⟨*Source viewer* 26b⟩≡ (9a)

```
  tool srcview is { command = "wish-adapter -script srcview.tcl" }

  process SRCVIEW is
  let
    Src    : srcview,
    Level  : term, Action : term,
    Module : str,  Line   : int,
    Breaks : list, Adr    : int,
    Name   : str
  in
    execute(srcview, Src?) .
    subscribe(line-number(<str>, <int>)) .
    subscribe(breaks-changed(<list>)) .
    subscribe(config-changed) .
    ( rec-msg(srcview, create-view) .
      snd-do(Src, create-view)
      +
      rec-note(line-number(Module?, Line?)) .
      snd-do(Src, line-number(Module, Line))
      +
      rec-note(breaks-changed(Breaks?)) .
```

```
    snd-do(Src, breaks-changed(Breaks))
    +
    rec-event(Src, break-line-on(Module?, Line?)) .
    snd-msg(breakpoints, line-on(Module, Line)) .
    snd-ack-event(Src, break-line-on(Module, Line))
    +
    rec-event(Src, break-line-off(Module?, Line?)) .
    snd-msg(breakpoints, line-off(Module, Line)) .
    snd-ack-event(Src, break-line-off(Module, Line))
    +
    rec-event(Src, clear-breakpoints) .
    snd-msg(breakpoints, clear-all) .
    snd-ack-event(Src, clear-breakpoints)
    +
    rec-event(Src, exec(Level?, Action?)) .
    snd-msg(exec, Level, Action) .
    snd-ack-event(Src, exec(Level, Action))
    +
    rec-event(Src, user-break) .
    snd-msg(exec, user-break) .
    snd-ack-event(Src, user-break)
  ) * delta
  ||
  CONFIG_HANDLER(Src)
endlet
```

4.7 Variable viewer

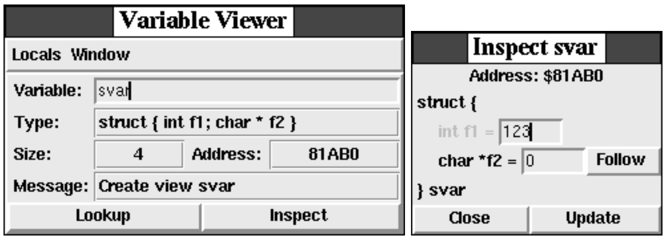

Figure 10: The variable viewer

The *variable viewer* (Figure 10) provides a means of watching the contents of memory at a more abstract level in comparison to the memory viewer. The contents of variables is shown in their 'native' representation, including structure, union, and enumeration variables. The variable viewer also provides the possibility to change the contents of variables.

27 ⟨*Variable viewer* 27⟩≡ (9a) 28a ▷

```
tool varview is { command = "wish-adapter -script varview.tcl" }

process VARVIEW is
let
  View   : varview,
  Var    : str,     Nr       : int,
  Type   : str,     TypeList : str,
  Adr    : int,     Size     : int,  Val     : int,
  Scope  : int,     Off      : int,
  ID     : int,     Start    : int,  End     : int,
  Block  : str,     Blocks   : list, RBlocks : list,
  Scopes : list,
```

```
    Reg    : str,     Regs     : list,
    Locals : list
  in
    execute(varview, View?) .
    subscribe(mem-changed(<int>, <int>)) .
    subscribe(word-changed(<int>, <int>)) .
    subscribe(scopes-changed(<list(<int>)>)) .
    subscribe(fp-changed(<int>)) .
    ( rec-msg(varview, create-view) .
      snd-do(View, create-view)
      +
```

A separation is made between the following three types of variables in C, because they all have different information associated with them.

- global variables
- local static variables
- local stack based (auto) variables

Notice that register variables are not supported at this time.

The `lookup-var` event retrieves information about a specific variable.

28a ⟨*Variable viewer* 27⟩+≡ (9a) ◁27 28b▷

```
      rec-event(View, lookup-var(Var?, Scopes?)) .
      if equal(Scopes, []) then
        snd-msg(label, lookup-var(Var))
      else
        snd-msg(label, lookup-var(Var, Scopes))
      fi .
      ( rec-msg(label, no-var(Var?)) .
        snd-do(View, no-var(Var))
        +
        rec-msg(label, global-var(Var?,Adr?,Size?,Type?,TypeList?)) .
        snd-do(View, lookup-global(Var, Adr, Size, Type, TypeList))
        +
        rec-msg(label, static-var(Var?,Scope?,Adr?,Size?,Type?,TypeList?)) .
        snd-do(View, lookup-static(Var,Scope,Adr,Size,Type,TypeList))
        +
        rec-msg(label, local-var(Var?,Scope?,Off?,Size?,Type?,TypeList?)) .
        snd-do(View, lookup-local(Var, Scope, Off, Size, Type, TypeList))
      ) .
      snd-ack-event(View, lookup-var(Var, Scopes))
      +
```

The `inspect-var` event retrieves information so a new variable window can be created.

28b ⟨*Variable viewer* 27⟩+≡ (9a) ◁28a 29a▷

```
      rec-event(View, inspect-var(Var?, Scopes?)) .
      if equal(Scopes, []) then
        snd-msg(label, lookup-var(Var))
      else
        snd-msg(label, lookup-var(Var, Scopes))
      fi .
      ( rec-msg(label, no-var(Var?)) .
        snd-do(View, no-var(Var))
        +
        rec-msg(label, global-var(Var?,Adr?,Size?,Type?,TypeList?)) .
        snd-do(View, inspect-global(Var, Adr, Size, Type, TypeList))
        +
        rec-msg(label, static-var(Var?,Scope?,Adr?,Size?,Type?,TypeList?)) .
        snd-do(View, inspect-static(Var,Scope,Adr,Size,Type,TypeList))
        +
```

```
    rec-msg(label, local-var(Var?,Scope?,Off?,Size?,Type?,TypeList?)) .
    snd-do(View, inspect-local(Var, Scope, Off, Size, Type, TypeList))
  ) .
  snd-ack-event(View, inspect-var(Var, Scopes))
  +
```

Off course, the variable viewer needs the contents of the virtual machine memory in order to display a variables value.

29a ⟨*Variable viewer* 27⟩+≡ (9a) ◁28b 29b▷

```
  rec-event(View, get-block-list(Blocks?)) .
  snd-msg(memory, get-block-list(Blocks)) .
  rec-msg(memory, block-list(RBlocks?)) .
  snd-do(View, block-list(RBlocks)) .
  snd-ack-event(View,get-block-list(Blocks))
  +
```

When the user edits the value of a variable, the `set-byte` and `set-word` events are used to change the appropriate memory locations.

29b ⟨*Variable viewer* 27⟩+≡ (9a) ◁29a 29c▷

```
  rec-event(View, set-byte(Adr?, Val?)) .
  snd-msg(memory, set-byte(Adr, Val))
         +
  rec-event(View, set-word(Adr?, Val?)) .
  snd-msg(memory, set-word(Adr, Val))
  +
```

The `list-locals` event retrieves a list of all local variables visible in the current scope. This list can then be presented to the user who can pick one of the variables to inspect.

29c ⟨*Variable viewer* 27⟩+≡ (9a) ◁29b 29d▷

```
  rec-event(View, list-locals(Scopes?)) .
  snd-msg(label, get-locals(Scopes)) .
  rec-msg(label, locals(Locals?)) .
  snd-do(View, list-locals(Scopes, Locals)) .
  snd-ack-event(View, list-locals(Scopes))
  +
```

The variable viewer reacts to memory changes by updating the contents of any variables that are affected by the change.

29d ⟨*Variable viewer* 27⟩+≡ (9a) ◁29c 29e▷

```
  rec-note(mem-changed(Start?, End?)) .
  snd-do(View, mem-changed(Start, End))
  +
  rec-note(word-changed(Adr?, Val?)) .
  snd-do(View, word-changed(Adr, Val))
  +
```

The current scope set is used by the variable viewer to keep track of the local variables that are currently in scope.

29e ⟨*Variable viewer* 27⟩+≡ (9a) ◁29d 29f▷

```
  rec-note(scopes-changed(Scopes?)) .
  snd-do(View, scopes-changed(Scopes))
  +
```

The frame pointer is used to track the address of stack based variables.

29f ⟨*Variable viewer* 27⟩+≡ (9a) ◁29e

```
  rec-note(fp-changed(Val?)) .
  snd-do(View, fp-changed(Val))
 ) * delta
endlet
```

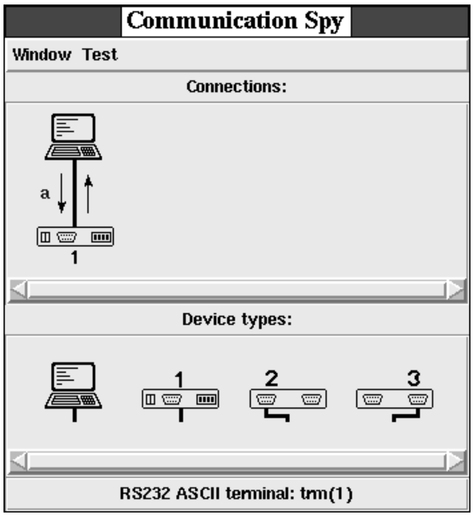

Figure 11: The communication spy

4.8 Communication spy

The main window of the *communication spy* (Figure 11) is divided into two parts. The bottom part shows a list of devices that can be simulated, the top part shows an overview of the connections that currently exist and animates the transmission of characters. Clicking on of the device icons shows a short description of the device in the message area.

30 ⟨*Communication spy* 30⟩≡ (9a) 31a ▷

```
tool comspy is { command = "wish-adapter -script comspy.tcl" }

process COMSPY is
let
  ComSpy   : comspy, Type     : term,
  TypeA    : term,   TypeB    : term,
  DevA     : term,   DevB     : term,
  Dir      : term,   Name     : str,
  BitFileA : str,    BitFileB : str,
  Info     : str,    Char     : str,
  Baud     : int,    Fmt      : str
in
  execute(comspy, ComSpy?) .
  subscribe(connection-established(<term>, <term>, <term>)) .
  subscribe(connection-destroyed(<term>, <term>)) .
  subscribe(send-char(<term>, <term>, <str>, <int>, <str>)) .
  ( rec-msg(comspy, create-view) .
    snd-do(ComSpy, create-view)
    +
```

Because we do not want to modify the communication spy when a new device is added to the simulator, a registration mechanism is needed. The following information must be included when registering a device:

- `Type`: this term must be a prefix of the term used when identifying a specific device. The virtual terminals for instance are indicated by `trm(0)`, `trm(1)`, etc. Their device type is `trm`.
- `Name`: The name of this device type. For instance *RS232 terminal*.
- `BitFileA` and `BitFileB`: Two bitmap files showing the device with a connection on the bottom and a connection on the top (See Figure 11).
- `Info`: A string providing a more verbose description of the device type. This string is shown in the message area when the user clicks on a device icon.

31a ⟨*Communication spy* 30⟩+≡ (9a) ◁30 31b▷
```
      rec-msg(comspy, register-device(Type?, Name?, BitFileA?, BitFileB?, Info?)) .
      snd-do(ComSpy, register-device(Type, Name, BitFileA, BitFileB, Info))
      +
```

When a connection is established, a character is send, or a connection is destroyed, a note is send so the communication spy is informed.

31b ⟨*Communication spy* 30⟩+≡ (9a) ◁31a
```
      rec-note(connection-established(DevA?, DevB?, Dir?)) .
      snd-do(ComSpy, add-connection(DevA, DevB, Dir))
      +
      rec-note(send-char(DevA?, DevB?, Char?, Baud?, Fmt?)) .
      snd-do(ComSpy, send-char(DevA, DevB, Char, Baud, Fmt))
      +
      rec-note(connection-destroyed(DevA?, DevB?)) .
      snd-do(ComSpy, remove-connection(DevA, DevB))
    ) * delta
  endlet
```

4.9 Terminal emulator

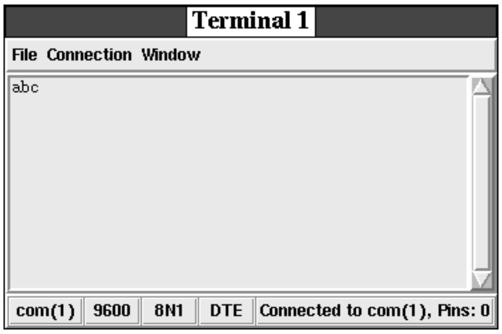

Figure 12: The terminal emulator

This tool is a simple terminal emulator that spawns virtual terminals that can be connected to any simulated serial device, for instance to one of the UPI-10 serial ports or even to another terminal. Any characters send by the other device are displayed in the main terminal window and any characters typed by the user are send to the "remote" device.

ASCII files can be uploaded and downloaded to make automated testing possible.

31c ⟨*Terminal emulator* 31c⟩≡ (9a) 31d▷
```
  tool terminal is { command = "wish-adapter -script terminal.tcl" }

  process TERMINAL is
  let
    Term  : terminal,
    ID    : int,
    PTerm : int
  in
    execute(terminal, Term?) .
```

After the `terminal` tool is created, it is registered with the communication spy so the user is aware of its presence.

31d ⟨*Terminal emulator* 31c⟩+≡ (9a) ◁31c 32a▷
```
    snd-msg(comspy, register-device(trm, "Terminal", "clipart/trm-a.xbm",
                  "clipart/trm-b.xbm", "RS232 ASCII terminal")) .
    ( rec-msg(terminal, create-view) .
```

```
      snd-do(Term, create-view)
      +
```

Every new terminal needs its own process to handle the communication protocol from Section 4.1.4.

32a ⟨*Terminal emulator* 31c⟩+≡ (9a) ◁31d

```
      rec-event(Term, new-trm(ID?)) .
      create(TERM(Term, ID), PTerm?) .
      snd-ack-event(Term, new-trm(ID))
    ) * delta
  endlet
```

⟨*TERM process* 32b⟩

In contrast to the UPI-10 ports, a terminal can initiate connections. This is done by sending a connection request to the remote device (see also Figure 4). The remote device responds by either sending a `connection-established` message or a `connection-refused` message. A terminal is also able to accept and reject connections initiated by a remote device. This makes it possible to connect two terminals. Note that the initiating party always sends notes about any interesting events, so tools like the `communication spy` can visualize these events.

32b ⟨*TERM process* 32b⟩≡ (32a) 32c▷

```
  process TERM(Term:terminal, ID:int) is
  let
    Peer   : term, Char    : str,
    PinsIn : int,  PinsOut : int,
    Baud   : int,  Fmt     : str
  in
    ( rec-event(Term, connect(ID, Peer?, PinsOut?)) .
      snd-msg(connect(trm(ID), Peer, PinsOut)) .
      ( rec-msg(connected(trm(ID), Peer, PinsIn?)) .
        snd-note(connection-established(trm(ID), Peer, bi)) .
        snd-do(Term, connected(ID, Peer, PinsIn))
        +
        rec-msg(connection-refused(trm(ID), Peer)) .
        snd-do(Term, connection-refused(ID, Peer))
      ) .
      snd-ack-event(Term, connect(ID, Peer, PinsOut))
      +
      rec-msg(connect(Peer?, trm(ID), PinsIn?)) .
      snd-eval(Term, connect(Peer, ID, PinsIn)) .
      ( rec-value(Term, connected(Peer, ID, PinsOut?)) .
        snd-msg(connected(Peer, trm(ID), PinsOut))
        +
        rec-value(Term, connection-refused(Peer, trm(ID))) .
        snd-msg(connection-refused(Peer, trm(ID)))
      )
      +
```

Both the remote device or the terminal can initiate disconnection.

32c ⟨*TERM process* 32b⟩+≡ (32a) ◁32b 32d▷

```
      rec-event(Term, disconnect(ID, Peer?)) .
      snd-msg(disconnect(trm(ID), Peer)) .
      snd-note(connection-destroyed(trm(ID), Peer)) .
      snd-ack-event(Term, disconnect(ID, Peer))
      +
      rec-msg(disconnect(Peer, trm(ID))) .
      snd-do(Term, disconnect(Peer, ID))
      +
```

When connected, both the remote device and the terminal can send pin changes and characters.

32d ⟨*TERM process* 32b⟩+≡ (32a) ◁32c

```
      rec-event(Term, pins(ID, Peer?, PinsOut?)) .
      snd-msg(pins(trm(ID), Peer, PinsOut)) .
      snd-ack-event(Term, pins(ID, Peer, PinsOut))
      +
      rec-event(Term, send-char(ID, Peer?, Char?, Baud?, Fmt?)) .
      snd-msg(send-char(trm(ID), Peer,Char,Baud,Fmt)) .
      snd-note(send-char(trm(ID), Peer, Char, Baud, Fmt)) .
      snd-ack-event(Term, send-char(ID, Peer, Char, Baud, Fmt))
      +
      rec-msg(send-char(Peer?, trm(ID), Char?, Baud?, Fmt?)) .
      snd-do(Term,send-char(Peer, ID, Char, Baud, Fmt))
      +
      rec-msg(pins(Peer?, trm(ID), PinsIn?)) .
      snd-do(Term, pins(Peer, ID, PinsIn))
    ) * rec-event(Term, close-trm(ID)) .
    snd-ack-event(Term, close-trm(ID))
  endlet
```

4.10 User preferences

Not everybody has the same style when it comes to software development, so it must be possible to tailor the system to the needs of the individual. The user must be able to maintain different sets of preferences to reflect different session types. We use a generic *configuration* tool that keeps track of a set of options as ⟨*key*, *value*⟩ pairs. The *key* is a single string, the *value* can be any term.

33 ⟨*Configuration management* 33⟩≡ (9a) 34a▷

```
  tool config is { command = "./config -config-file ~/.upisimrc" }

  process CONFIG is
  let
    File : str,
    Msg  : str,
    Conf : config,
    Key  : str,
    Keys : list,
    Opts : list,
    Val  : term
  in
    execute(config, Conf?) .
    ( rec-msg(config, load-config(File?)) .
      snd-eval(Conf, load-config(File)) .
      ( rec-value(Conf, config-loaded(File)) .
        snd-msg(config, config-loaded(File))
        +
        rec-value(Conf, error(Msg?)) .
        snd-msg(Conf, error(Msg))
      )
      +
      rec-msg(config, save-config(File?)) .
      snd-eval(Conf, save-config(File)) .
      ( rec-value(Conf, config-saved(File)) .
        snd-msg(config, config-saved(File))
        +
        rec-value(Conf, error(Msg?)) .
        snd-msg(Conf, error(Msg))
      )
      +
      rec-msg(config, set-option(Key?, Val?)) .
```

```
      snd-do(Conf, set-option(Key, Val))
      +
```

Option retrieval is done in batches, because tools often want to retrieve all of their options, and we want to handle this in one transaction.

34a ⟨*Configuration management* 33⟩+≡ (9a) ◁33 34b▷

```
      rec-msg(config, get-options(Keys?)) .
      snd-eval(Conf, get-options(Keys)) .
      rec-value(Conf, options(Opts?)) .
      snd-msg(config, options(Opts))
      +
```

When the configuration changes, all relevant processes must be notified. We do not want every tool polling all of the options it is interested in when a single option changes, so we introduce two different note types that are used when the configuration changes. The first type is send when a new set of options is loaded. It just indicates that all tools should poll the options they need.

34b ⟨*Configuration management* 33⟩+≡ (9a) ◁34a 34c▷

```
      rec-event(Conf, config-changed) .
      snd-note(config-changed) .
      snd-ack-event(Conf, config-changed)
      +
```

The second kind of note is send when a single option changes. When a tool knows it is the only one that ever changes an option, it can safely ignore this note type. The new value of the option is contained in the note, so there is no need to poll it.

34c ⟨*Configuration management* 33⟩+≡ (9a) ◁34b

```
      rec-event(Conf, option-changed(Key?, Val?)) .
      snd-note(option-changed(Key, Val)) .
      snd-ack-event(Conf, option-changed(Key, Val))
    ) * delta
  endlet
```

⟨*Configuration handling* 34d⟩

Almost all processes handle options in the same way. That is why we use the preprocessor to `#define` this stereotype behaviour. The strict typechecking of ToolBus scripts does not allow us to define this behaviour as a process.

34d ⟨*Configuration handling* 34d⟩≡ (34c)

```
  #define CONFIG_HANDLER(T)                        \
  let                                              \
    Keys    : list,                                \
    Options : list,                                \
    Key     : str,                                 \
    Val     : term                                 \
  in                                               \
    subscribe(config-changed) .                    \
    subscribe(option-changed(<str>,<term>)) .      \
    ( rec-note(config-changed) .                   \
      snd-do(T, config-changed)                    \
      +                                            \
      rec-note(option-changed(Key?, Val?)) .       \
      snd-do(T, option-changed(Key, Val))          \
      +                                            \
      rec-event(T, get-options(Keys?)) .           \
      snd-msg(config, get-options(Keys)) .         \
      rec-msg(config, options(Options?)) .         \
      snd-do(T, options(Options)) .                \
      snd-ack-event(T, get-options(Keys))          \
      +                                            \
      rec-event(T, set-option(Key?, Val?)) .       \
```

```
    snd-msg(config, set-option(Key, Val)) .      \
    snd-ack-event(T, set-option(Key, Val))       \
  ) * delta                                      \
endlet
```

5 Conclusions

5.1 Constraints

This section lists a number of constraints imposed on a system in order to fit in this simulator framework. When faced with a project that could benefit from this work, this section might be a good place to start your evaluation.

5.1.1 Hardware architecture

The framework is biased towards what you might call a *classic* architecture. This means a system containing a single processor[4] that reads its instruction and data streams from a central store. The simulated processor must at least implement a program counter so the other tools can track the execution, and a frame pointer if there are any stack based (local) variables.

5.1.2 Software support

To be able to make sense of the contents of the simulated memory and register contents, the simulator needs static symbol information. In most cases this information is generated by the linker. The following table shows the information needed and the tools that depend on it. A † in a certain column means that that particular tool uses the feature mentioned in the first column, but still works without it (albeit with some reduced functionality). A ‡ means that a tool *depends* on a feature and is useless without it.

	Basic tool				
VM tool provides:	Control Center	CPU Viewer	Memory Viewer	Source Viewer	Variable Viewer
static features					
address labels	†	†	†		
line number info	†	†	†	‡	
function info	†				
variable info					‡
scope info					‡
breakpoints		†		†	
dynamic features					
program counter	†	‡		‡	†
cycle counter	†				
frame pointer					†
cur. scope list					†
breakpoints		†		†	

5.2 Evaluation

At the start of this paper, some reusability and performance targets where presented. In this section we will evaluate which targets have been met and which have not.

5.2.1 Performance

Because the actual program execution takes place within in the virtual machine, the simulator performance in this area is not influenced by our distributed architecture.

Although the user interface components are all separate tools, the interactive response times are good, essentially justifying the choice for the ToolBus concept.

5.2.2 Reusability

The reusability of the system is best illustrated by some statistics about the size of the software.

[4] We do not exclude a multi-processor environment, but it requires some work.

UPI-10 development tools		
Component	Lines of code	Predominant language
lcc front end (existing)	13741	C
lcc back end	2546	C
assembler	5477	C++
linker	2502	C++
library	5632	Assembler
simulator:		
virtual machine (non reusable)	17420	C++
UI components (reusable)	8252	Tcl/Tk
communication control	1657	ToolBus script

Of course, these figures only provide a very rough estimate, not in the least because the implementation languages are so diverse.

An obvious conclusion is that the simulator is almost as big as the other development tools together, warranting a major investment to reuse parts of it.

In a new simulator, the UI components and the ToolBus script are directly reusable. This means an instant reuse of over 35 percent of high-level simulator code. If we take into account that both the ToolBus and Tcl/Tk scripting languages are much more compact than the C++ code that was used for the virtual machine, we can conclude that more than 50 percent of the design and coding effort can be reused when building a new simulator. Factoring out the machine independent components from the virtual machine would increase the reusability even further.

5.2.3 Extensibility

Because of the clear responsibilities of the different components and the open nature of the framework, the basic framework is very extensible. We have extended the simulator to simulate the special serial communication hardware, by adding the terminal emulator (Section 4.9) and the communication spy (Section 4.8). This only required extending the basic framework, not changing it.

5.3 ToolBus experience

One of the goals of this project was to gain experience with ToolBus programming. Now that we have implemented one of the first medium sized ToolBus applications, some preliminary conclusions can be drawn.

- The ToolBus is very good in separating components, especially if the traffic between them can be kept light.
- An interesting separation technique is the complete separation between the user interface and computational component. Because the ToolBus communication is very fast compared to a user interacting with a user interface, only a small performance penalty is paid by separating the components.
- The one to one mapping between tools and programs is very unsatisfactory. The need for interprocess communication for every toolbus/tool interaction seriously affects performance.

The only reason why we did not split up the system even further, for instance by separating the symbol table manager and the actual virtual machine, was a lack of performance. This suggests a general strategy when designing a system around the ToolBus.

- First start with a (existing ?) design where all components are located in a single program.
- Move all user interface components into separate tools.
- Try to isolate other components that can be turned into tools without seriously affecting the overall performance.

Several studies on micro-kernel operating systems (for instance the Amoeba operating system, [RVRT89]) have shown that it is possible to get a decent performance while using message passing. The possibility to incorporate some tools in the ToolBus instead of connecting them using interprocess communication primitives might make a big difference in this area.

5.4 Future research

We definitely need to develop more simulators, in order to refine and extend this framework.

An interesting experiment would be to replace the virtual machine tool with a symbol table manager and a program debugging interface to programs running on the *host* machine, so the tool can monitor and interact with these programs. This would extend the framework to incorporate debuggers as well as simulators *without changing the user interface components.*

On the ToolBus side, a number of interesting fields lay open for investigation:

- How much can we boost performance by moving tools into the ToolBus thus eliminating the interprocess communication?
- The dynamic loading of new ToolBus scripts seems to be a logical next step, complementing the dynamic creation of processes and the dynamic connection/disconnection of tools, both of which are already present.
- The ToolBus script for the simulator grew to considerable size (more than 1600 lines), but was still manageable. As we gain experience with the ToolBus, our scripts will grow and we will need some kind of modularization construct. We might want to look into a more object oriented ToolBus.
- The ToolBus uses a string representation of terms to communicate with the tools. This makes it very hard to share common subterms. Experimenting with an architecture independent term or graph representation that preserves sharing would be interesting.

A UPI-10 Hardware Specifications

Dimensions	- $130x100x30mm$.
Environmental	- -20 degree Celsius to +70 degree Celsius.
	- Humidity 40% to 90%.
	Unit is dust and water resistant.
Power supply	- 11 Vdc to 29 Vdc.
Microcontroller	- Hitachi H8/532 16 bit, running at 10 Mhz.
Boot ROM	- 32 Kb.
Flash EPROM	- standard 128 Kb, max. 475 Kb.
RAM	- standard 32 Kb, max. 512 Kb (Field upgradable).
Retention	- 10 year backup with maximum memory.
Communication Ports	- 2 RS 232C DTE serial ports.
	1 RS 232C DTE/DCE programmable port.
Connector	- 9 Pins sub-D.
Baudrates	- 19200, 9600, 4800, 2400, 1200 and many in between.
Format	- 1 start bit, 5-8 data bits, even or odd parity bits, 1 or 2 stop bits.

B The syntax of ToolBus scripts

B.1 Preprocessor directives

The *script_name* given as argument to the ToolBus is always preprocessed by the C preprocessor before it is parsed as ToolBus script. In this way, directives like, e.g., `#define`, `#include` and `#ifdef` can be used freely in ToolBus scripts. We summarize the most frequently used directives:

- `#define` ***`identifier token-sequence`*** causes the preprocessor to replace all occurrences of *identifier* by *token-sequence*.
- `#define` ***`identifier ( identifier-list ) token-sequence`*** is a macro definition with parameters given by *identifier-list*. Textual occurrences of the identifier followed by an argument list containing an appropriate number of tokens separated by comma's will be replaced by *token-sequence* after parameter substitution.
- `#include "`***`filename`***`"` will be replaced by the entire contents of the named file.
- `#if`, `#ifdef`, and `#ifndef` can be used for the conditional incorporation or exclusion of parts of a script.

We refer to any ANSI C manual for a detailed description of these directives.

B.2 Context-free syntax

```
exports
  sorts BOOL NAT INT SIGN EXP UNSIGNED-REAL REAL STRING ID NAME VNAME BSTR
        TERM TERM-LIST VAR GEN-VAR TYPE ATOM ATOMIC-FUN PROC PROC-APPL FORMALS
        TIMER-FUN FEATURE-ASG FEATURES TB-CONFIG DEF T-SCRIPT
  lexical syntax
        [ \t\n]                                           -> LAYOUT
        "%%" ~[\n]*                                       -> LAYOUT

        [0-9]+                                            -> NAT
        NAT                                               -> INT
        SIGN NAT                                          -> INT
        [+\-]                                             -> SIGN

        [eE] NAT                                          -> EXP
        [eE] SIGN NAT                                     -> EXP
        NAT "." NAT                                       -> UNSIGNED-REAL
        NAT "." NAT EXP                                   -> UNSIGNED-REAL
        UNSIGNED-REAL                                     -> REAL
        SIGN UNSIGNED-REAL                                -> REAL

        [a-z][A-Za-z0-9\-]*                               -> ID
        "\"" ~[\"]* "\""                                  -> STRING
        [A-Z][A-Za-z0-9\-]*                               -> NAME
        [A-Z][A-Za-z0-9\-]*                               -> VNAME
        [a-z][a-z\-]*                                     -> ATOMIC-FUN

        delay                                             -> TIMER-FUN
        abs-delay                                         -> TIMER-FUN
        timeout                                           -> TIMER-FUN
        abs-timeout                                       -> TIMER-FUN
  context-free syntax
        true                                              -> BOOL
        false                                             -> BOOL
        BOOL                                              -> TERM
        INT                                               -> TERM
        REAL                                              -> TERM
        STRING                                            -> TERM
```

```
      TERM                                          -> TYPE

      VNAME                                         -> VAR
      VNAME ":" TYPE                                -> VAR

      VAR                                           -> GEN-VAR
      VAR "?"                                       -> GEN-VAR
      GEN-VAR                                       -> TERM
      "<" TERM ">"                                  -> TERM
      ID                                            -> TERM
      ID "(" TERM-LIST ")"                          -> TERM
      {TERM ","}*                                   -> TERM-LIST
      "[" TERM-LIST "]"                             -> TERM

      NAME                                          -> VNAME

      ATOMIC-FUN "(" TERM-LIST ")"                  -> ATOM
      delta                                         -> ATOM
      tau                                           -> ATOM
      create "(" NAME "(" TERM-LIST ")" ","  TERM ")"  -> ATOM
      ATOM TIMER-FUN "(" TERM ")"                   -> ATOM
      VNAME ":=" TERM                               -> ATOM

      ATOM                                          -> PROC
      PROC "+" PROC                                 -> PROC  {left}
      PROC "." PROC                                 -> PROC  {right}
      PROC "||" PROC                                -> PROC  {right}
      PROC "*" PROC                                 -> PROC  {left}
      "(" PROC ")"                                  -> PROC  {bracket}
      if TERM then PROC else PROC fi                -> PROC
      if TERM then PROC fi                          -> PROC
      execute(TERM-LIST)                            -> PROC
      let {VAR ","}* in PROC endlet                 -> PROC

      NAME                                          -> PROC-APPL
      NAME "(" TERM-LIST ")"                        -> PROC-APPL
      PROC-APPL                                     -> PROC

      "(" {GEN-VAR ","}* ")"                        -> FORMALS
                                                    -> FORMALS

      process NAME FORMALS is PROC                  -> DEF
      ID "=" STRING                                 -> FEATURE-ASG
     "{" { FEATURE-ASG  ";"}* "}"                   -> FEATURES
      tool ID FORMALS is FEATURES                   -> DEF
      toolbus "("{PROC-APPL ","}+ ")"               -> TB-CONFIG
      DEF* TB-CONFIG                                -> T-SCRIPT

priorities
     PROC "*" PROC -> PROC > PROC "." PROC -> PROC >
     PROC "+" PROC -> PROC > PROC "||" PROC -> PROC
```

C Expressions in ToolBus scripts

C.1 Boolean and arithmetic functions

Function	Result type	Description
not(<bool>$_1$)	<bool>	$\neg$ <bool>$_1$
and(<bool>$_1$,<bool>$_2$)	<bool>	<bool>$_1$ $\wedge$ <bool>$_2$
or(<bool>$_1$,<bool>$_2$)	<bool>	<bool>$_1$ $\vee$ <bool>$_2$
equal(<term>$_1$, <term>$_2$)	<bool>	<term>$_1$ $\equiv$ <term>$_2$; for lists multi-set equality
not-equal(<term>$_1$, <term>$_2$)	<bool>	not(equal(<term>$_1$, <term>$_2$))
add(<int>$_1$,<int>$_2$)	<int>	<int>$_1$ $+$ <int>$_2$
sub(<int>$_1$,<int>$_2$)	<int>	<int>$_1$ $-$ <int>$_2$
mul(<int>$_1$,<int>$_2$)	<int>	<int>$_1$ $\times$ <int>$_2$
div(<int>$_1$,<int>$_2$)	<int>	<int>$_1$ $/$ <int>$_2$
mod(<int>$_1$,<int>$_2$)	<int>	<int>$_1$ **mod** <int>$_2$
abs(<int>$_1$)	<int>	absolute value $\mid$ <int>$_1$ $\mid$
less(<int>$_1$,<int>$_2$)	<bool>	<int>$_1$ $<$ <int>$_2$
less-equal(<int>$_1$,<int>$_2$)	<bool>	<int>$_1$ $\leq$ <int>$_2$
greater(<int>$_1$,<int>$_2$)	<bool>	<int>$_1$ $>$ <int>$_2$
greater-equal(<int>$_1$,<int>$_2$)	<bool>	<int>$_1$ $\geq$ <int>$_2$
radd(<real>$_1$,<real>$_2$)	<real>	<real>$_1$ $+$ <real>$_2$
rsub(<real>$_1$,<real>$_2$)	<real>	<real>$_1$ $-$ <real>$_2$
rmul(<real>$_1$,<real>$_2$)	<real>	<real>$_1$ $\times$ <real>$_2$
rdiv(<real>$_1$,<real>$_2$)	<real>	<real>$_1$ $/$ <real>$_2$
rabs(<real>$_1$)	<real>	absolute value $\mid$<real>$_1$$\mid$
rless(<real>$_1$,<real>$_2$)	<bool>	<real>$_1$ $<$ <real>$_2$
rless-equal(<real>$_1$,<real>$_2$)	<bool>	<real>$_1$ $\leq$ <real>$_2$
rgreater(<real>$_1$,<real>$_2$)	<bool>	<real>$_1$ $>$ <real>$_2$
rgreater-equal(<real>$_1$,<real>$_2$)	<bool>	<real>$_1$ $\geq$ <real>$_2$
sin(<real>$_1$)	<real>	sin(<real>$_1$)
cos(<real>$_1$)	<real>	cos(<real>$_1$)
atan(<real>$_1$)	<real>	tan^{-1}(<real>$_1$) in range $[-\pi/2, \pi/2]$
atan2(<real>$_1$, <real>$_2$)	<real>	tan^{-1}(<real>$_1$/<real>$_2$) in range $[-\pi, \pi]$
exp(<real>$_1$)	<real>	exponential function $e^{\texttt{<real>}_1}$
log(<real>$_1$)	<real>	natural logarithm ln(<real>$_1$), <real>$_1$ > 0
log10(<real>$_1$)	<real>	base 10 logarithm log_{10}(<real>$_1$), <real>$_1$ > 0
sqrt(<real>$_1$)	<real>	$\sqrt{\texttt{<real>}_1}$, <real>$_1$ ≥ 0

C.2 Functions on lists and multi-sets

Function	Result type	Description
first(<list>$_1$)	<term>	first element of <list>$_1$; [] for non-lists
next(<list>$_1$)	<list>	remaining elements of <list>$_1$; [] for non-lists
join(<term>$_1$,<term>$_2$)	<list>	concatenation of <term>$_1$ and <term>$_2$; for a list argument <term>$_i$ ($i = 1, 2$), the list elements are spliced into the new list; non-list arguments are included as single element of the new list.
size(<list>$_1$)	<int>	$\vert$<list>$_1\vert$ (number of elements in list)
index(<list>$_1$,<int>$_1$)	<term>	If $\vert$<list>$_1\vert \leq$ <int>$_1$ return the <int>$_1$th element from <list>$_1$; otherwise [] and give a warning.
replace(<list>$_1$,<int$_1$>,<term>$_1$)	<list>	If $\vert$<list>$_1\vert \leq$ <int>$_1$ replace the <int>$_1$th element of <list>$_1$ by <term>$_1$ and return the modified (and partially copied) version of <list>$_1$; otherwise return <list>$_1$ and give a warning.
get(<list>$_1$,<term>$_1$)	<term>	If <list>$_1$ contains a pair [<term>$_1$, <term>$'_1$] then <term>$'_1$; otherwise [].
put(<list>$_1$,<term>$_1$, <term>$_2$)	<list>	If <list>$_1$ contains a pair [<term>$_1$, <term>$'_1$] then replace it by [<term>$_1$, <term>$_2$]; otherwise add a new pair [<term>$_1$, <term>$_2$] to <list>$_1$.
member(<term>$_1$,<list>$_2$)	<bool>	<term>$_1$ $\in$ <list>$_2$ (membership in multi-set)
subset(<list>$_1$, <list>$_2$)	<bool>	<list>$_1$ $\subseteq$ <list>$_2$ (subset on multi-sets)
diff(<list>$_1$, <list>$_2$)	<list>	<list>$_1$ $-$ <list>$_2$ (difference on multi-sets)
inter(<list>$_1$, <list>$_2$)	<list>	<list>$_1$ $\cap$ <list>$_2$ (intersection on multi-sets)

C.3 Predicates and functions on terms

Function	Result type	Description
is-bool(<term>)	<bool>	If <term> is of type bool then true; otherwise false.
is-int(<term>)	<bool>	If <term> is of type int then true; otherwise false.
is-real(<term>)	<bool>	If <term> is of type real then true; otherwise false.
is-str(<term>)	<bool>	If <term> is of type str then true; otherwise false.
is-bstr(<term>)	<bool>	If <term> is of type bstr then true; otherwise false.
is-appl(<term>)	<bool>	If <term> is an application then true; otherwise false.
is-list(<term>)	<bool>	If <term> is a list then true; otherwise false.
is-empty(<term>)	<bool>	If <term> equals [] then true; otherwise false.
is-var(<term>)	<bool>	If <term> is a variable then true; otherwise false.
is-result-var(<term>)	<bool>	If <term> is a result variable then true; otherwise false.
is-formal(<term>)	<bool>	If <term> is a formal variable then true; otherwise false.
fun(<term>)	<str>	If <term> is an application then its function symbol; otherwise "".
args(<term>)	<list>	If <term> is an application then its argument list; otherwise [].

C.4 Miscellaneous functions

Function	Result type	Description
process-id	<int>	id of current process
process-name	<str>	name of current process
quote(<term>)	<term>	quoted (unevaluated) term, only variables are replaced by their value
functions	<list>	list of built-in functions
current-time	<list>	six-tuple describing current absolute time
sec(<int>$_1$)	<int>	convert <int>$_1$ in seconds
msec(<int>$_1$)†	<int>	convert <int>$_1$ in milli-seconds

†Not yet implemented in the current version

D Synopsis of primitives available in ToolBus scripts

Primitive	Description
`delta`	inaction (deadlock)
`tau`	internal step
P_1`+`P_2	choice
P_1`.`P_2	sequential composition
P_1`||`P_2	parallel composition
P_1`*`P_2	iteration
`if` T `then` P `fi`	guarded command
`if` T `then` P_1 `else` P_2 `fi`	conditional
`create(`Pnm`(`T`,...), `Pid`?)`	process creation[1]
V `:=` T	assignment, T expression (see C)
`snd-msg(`T`,...)`	send a message (binary, synchronous)
`rec-msg(`T`,...)`	receive a message (binary, synchronous)
`snd-note(`T`)`	send a note (broadcast, asynchronous)
`rec-note(`T`)`	receive a note (asynchronous)
`no-note(`T`)`	no notes available for process
`subscribe(`T`)`	subscribe to notes
`unsubscribe(`T`)`	unsubscribe from notes
`delay(`T`)`	relative time delay of atom
`abs-delay(`T`,...)`	absolute time delay of atom[2]
`timeout(`T`)`	relative timeout of atom
`abs-timeout(`T`,...)`	absolute timeout of atom[2]
`rec-connect(`Tid`?)`	receive a connection request from a tool
`rec-disconnect(`Tid`?)`	receive a disconnection request form a tool
`execute(`Tnm`(`T`,...), `Tid`?)`	execute a tool[1]
`snd-terminate(`Tid`, `T`)`	terminate the execution of a tool
`shutdown(`T`)`	terminate ToolBus
`reconfigure`	reconfigure ToolBus†
`attach-monitor`	attach a monitoring tool to a process†
`detach-monitor`	detach a monitoring tool from a process†
`snd-eval(`Tid`, `T`)`	send evaluation request to tool
`snd-cancel(`Tid`)`	cancel an evaluation request to tool†
`rec-value(`Tid`, `T`)`	receive a value from a tool
`snd-do(`Tid`, `T`)`	send request to tool (no return value)
`rec-event(`Tid`, `T`, ...)`	receive event from tool
`snd-ack-event(`Tid`, `T`)`	acknowledge a previous event from a tool
`printf(`S`, `T`, ...)`	print terms (after variable replacement) according to format S
`read(`T_1`, `T_2`)`	give prompt T_1, read term, should match with T_2
`process` Pnm`(`F`, ...) is` P	process definition[3]
`let` F`, ... in` P `endlet`	declare variables in P
`tool` Tnm`(`F`,...) is {` $Feat$`, ... }`	tool definition[3]
`host =` Str	host feature in tool definition
`command =` Str	command feature in tool definition
`details = <<` $Lines$ `>>`	details feature in tool definition
`toolbus(`Pnm`(`T`,...), ...)`	ToolBus configuration

Notes

1	`(`T`, ...)` is optional
2	Absolute time described by a 6-tuple (*year*, *month*, *day*, *hour*, *minutes*, *seconds*) with $year \geq 95$, $1 \leq month \leq 12$, $1 \leq day \leq 31$, $0 \leq hour \leq 23$, $0 \leq minutes \leq 59$, and $0 \leq seconds \leq 61$ (seconds can be greater than 59 to allow leap seconds). Absolute time may be abbreviated, by omitting, at most, the first three elements of the 6-tuple. Omitted elements default to their current value.
3	`(`F`, ...)` is optional
†	Not yet implemented

Legendum

T	term
T`, ...`	list of terms separated by comma's
V	variable
F	declaration of formal or local variable of the form V:*Type*
P, P_1, P_2	process expression
Tid	tool identifier, a variable of type *Tnm* (with *Tnm* declared as tool name)
Tnm	tool name
Pnm	process name
Pid	process identifier, a variable of type `int`
Str	a string constant
Lines	list of lines

References

[BK95] J.A. Bergstra and P. Klint. The discrete time toolbus. Report p9502, Programming Research Group, Department of Mathematics and Computer Science, University of Amsterdam, FWI WCW, Kruislaan 403, 1098 SJ Amsterdam, 1995.

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

[FH95] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, 1995.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, second edition, 1988.

[RVRT89] R. Van Staveren R. Van Renesse and A.S. Tanenbaum. Performance of the amoeba distributed operating system. *Software – Parctice and Experience 19*, 1989.