The Discrete Time ToolBus

J.A. Bergstra[1,2] and P. Klint[3,1]

[1] Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands
[2] Department of Philosophy, Utrecht University
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
[3] Department of Software Technology
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract. The notion of "time" plays an important role when coordinating large, heterogeneous, distributed software systems. We present a generic coordination architecture that supports relative and absolute, discrete time. First, we briefly sketch the ToolBus coordination architecture. Next, we give a major example of its use: a distributed auction. Finally, we present the theory underlying our notion of discrete time.

1 Introduction

1.1 Motivation

Building large, heterogeneous, distributed software systems poses serious problems for the software engineer. Systems grow *larger* because the complexity of the tasks we want to automate increases. They become *heterogeneous* because large systems may be constructed by re-using existing software as components. It is more than likely that these components have been developed using different implementation languages and run on different hardware platforms. Systems become *distributed* because they have to operate in the context of local area networks.

It is fair to say that the *interoperability* of software components is essential to solve these problems. The question how to connect a number of independent, interactive, tools and integrate them into a well-defined, cooperating whole has already received substantial attention in the literature and it is easy to understand why:

- by connecting existing tools we can reuse their implementation and build new systems with lower costs;
- by decomposing a single monolithic system into a number of cooperating components, the modularity and flexibility of the systems' implementation can be improved.

We refer to our paper [BK96] for further motivation and and a survey of related work.

1.2 Our approach

Requirements and points of departure. Before explaining our approach to component interconnection in more detail, it is useful to make a list of our requirements and state our points of departure.

To get control over the possible interactions between software components ("tools") we forbid direct inter-tool communication. Instead, all interactions are controlled by a "script" that formalizes all the desired interactions among tools. This leads to a communication architecture resembling a hardware communication bus, and therefore we will call it a "ToolBus". Ideally speaking, each individual tool can be replaced by another one, provided that it implements the same protocol as expected by other tools. The resulting software architecture should thus lead to a situation in which tools can be combined with each other in many fashions. We replace the classical procedure interface (a named procedure with typed arguments and a typed result) by a more general *behaviour description.*

A "**T** script" should satisfy a number of requirements:

- It has a formal basis and can be formally analyzed.
- It is simple, i.e., it only contains information directly related to the objective of tool integration.
- It exploits a number of predefined communication primitives, tailored towards our specific needs. These primitives are such, that the common cases of deadlock can be avoided by adhering to certain styles of writing specifications.
- The manipulation of *data* should be completely transparent, i.e., data can only be received from and send to tools, but inside the ToolBus there are no operations on them.
- There should be no bias towards any implementation language for the tools to be connected. We are at least interested in the use of C, Lisp, Tcl, and Asf+Sdf for constructing tools.
- It can be mapped onto an efficient implementation.

The ToolBus. The ToolBus coordination architecture can integrate and coordinate a fixed number of existing tools. We approach the problem of tool integration as follows:

Data integration: Instead of providing a general mechanism for representing the data in arbitrary applications, we will use a single, uniform, data representation based on term structures.

Control integration: the control integration between tools is achieved by using process-oriented "**T** scripts" that model the possible interactions between tools.

User-interface integration: we will *not* address user-interface integration as a separate topic, but we want to investigate whether our control integration mechanism can be exploited to achieve user-interface integration as well.

A consequence of this approach is that *existing* tools will have to be encapsulated by a small layer of software that acts as an "adapter" between the tool's internal dataformats and conventions and those of the TOOLBUS.

Compared with other approaches, the most distinguishing features of the TOOLBUS approach are:

- The prominent role of primitives for process control in the setting of tool integration. The major advantage being that complete control over tool communication can be achieved.
- The absence of built-in datatypes. Compare this with the abstract datatypes in, for instance, LOTOS [Bri87], PSF [MV90, MV93], and μCRL [GP90]. We only depend on a free algebra of terms and use matching to manipulate data. Transformations on data can only be performed by tools, giving opportunities for efficient implementation.

In [BK94] (see [BK96] for an extended abstract) we have applied a number of established techniques (i.e., process algebra [BK84, BW90], algebraic specification using ASF+SDF [BHK89, HHKR89, vDHK96], and C implementation) to approach the design of the TOOLBUS at various levels of abstraction. This has given rise to—even mutual—feedback between different levels. Experiences with this first design were reported in [BK96]. A redesign of the TOOLBUS is fully described in [BK95]. In this contribution (an extended abstract of [BK95]), we concentrate on the time-related features of the new TOOLBUS design. A large application of the TOOLBUS is described in [Oli96a, Oli96b]. A guide to TOOLBUS programming can be found in [Kli96].

1.3 Plan of the paper

We will now first give an overview of the TOOLBUS coordination architecture including an annotated example not involving time features (Section 2). Next, we discuss a major example that makes essential use of time: a distributed auction (Section 3). Then we present a description of the discrete time process algebra needed to define the notion of time in the TOOLBUS (Section 4). A discussion (Section 5) completes the paper.

2 Overview of the TOOLBUS coordination architecture

The global architecture of the TOOLBUS is shown in figure 1. The TOOLBUS serves the purpose of defining the cooperation of a variablenumber of *tools* T_i ($i = 1, ..., m$) that are to be combined into a complete system. The internal behaviour or implementation of each tool is irrelevant: they may be implemented in different programming languages, be generated from specifications, etc. Tools may, or may not, maintain their own internal state. Here we concentrate on the external behaviour of each tool. In general an *adapter* will be needed for each tool to adapt it to the common data representation and message protocols imposed by the TOOLBUS.

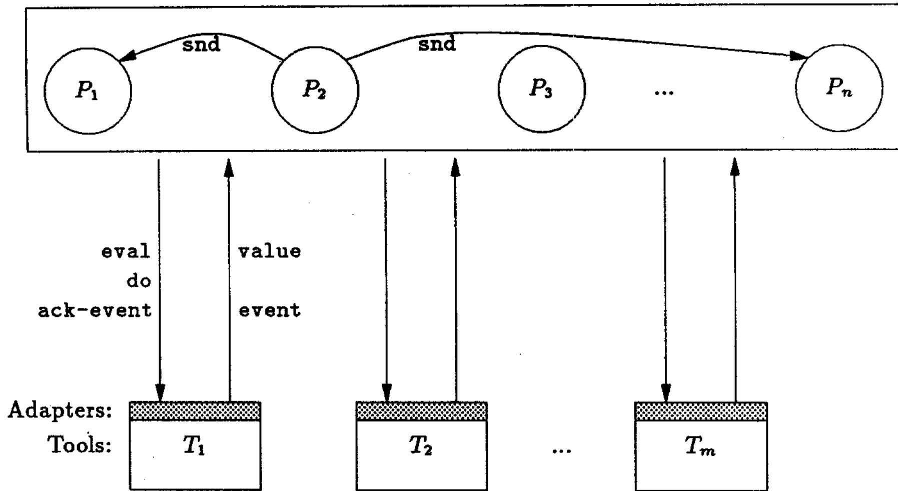

Fig. 1. Global organization of the TOOLBUS

The TOOLBUS itself consists of a variable number of processes P_i ($i = 1, ..., n$). The parallel composition of the processes P_i represents the intended behaviour of the whole system. Although a one-to-one correspondence between tools and processes seems simple and desirable, we do not enforce this and permit tools that are being controlled by more than one process as well as clusters of tools being controlled by a single process.

Inside the TOOLBUS, there are two communication mechanisms available. First, a process can send a *message* (using snd-msg) which should be received, synchronously, by one other process (using rec-msg). Messages are intended to request a service from another process. When the receiving process has completed the desired service it may inform the sender, synchronously, by means of another message (using snd-msg). The original sender can receive the reply using rec-msg. By convention, the original message is contained in the reply.

Second, a process can send a *note* (using snd-note) which is broadcasted to other, interested, processes. The sending process does not expect an answer while the receiving processes read notes asynchronously (using rec-note) at a low priority. Notes are intended to notify others of state changes in the sending process. Sending notes amounts to *asynchronous selective broadcasting*. Processes will only receive notes to which they have *subscribed*.

The communication between TOOLBUS and tools is based on handshaking communication between a TOOLBUS process and a tool. A process may send messages in several formats to a tool (snd-eval, snd-do, and snd-ack-event) while a tool may send the messages event and value to a TOOLBUS process.

Primitive	Description
`delta`	inaction ("deadlock")
`+`	choice between two alternatives (P_1 or P_2)
`.`	sequential composition (P_1 followed by P_2)
`*`	iteration (zero or more times P_1 followed by P_2)
`create`	process creation
`snd-msg`	send a message (binary, synchronous)
`rec-msg`	receive a message (binary, synchronous)
`snd-note`	send a note (broadcast, asynchronous)
`rec-note`	receive a note (asynchronous)
`no-note`	no notes available for process
`subscribe`	subscribe to notes
`unsubscribe`	unsubscribe from notes
`snd-eval`	send evaluation request to tool
`rec-value`	receive a value from a tool
`snd-do`	send request to tool (no return value)
`rec-event`	receive event from tool
`snd-ack-event`	acknowledge a previous event from a tool
`if ... then ... fi`	guarded command
`if ... then ... else ... fi`	conditional
	expressions
`||`	communication-free merge (parallel composition)
`let ... in ... endlet`	local variables
`:=`	assignment
`delay`	relative time delay
`abs-delay`	absolute time delay
`timeout`	relative timeout
`abs-timeout`	absolute timeout
`rec-connect`	receive a connection request from a tool
`rec-disconnect`	receive a disconnection request from a tool
`execute`	execute a tool
`snd-terminate`	terminate the execution of a tool
`shutdown`	terminate ToolBus
`attach-monitor`	attach a monitoring tool to a process
`detach-monitor`	detach a monitoring tool from a process

Fig. 2. Overview of ToolBus primitives.

There is no direct communication possible between tools.

2.1 Overview of T scripts

First, we address the data integration problem by introducing a notion of *terms* as follows:

- An integer *Int* is a term.

- A string *String* is a term.
- A variable *Var* is a term.
- A single identifier *Id* is a term.
- *Id*(*Term*$_1$, *Term*$_2$, ...) is a term, provided that *Term*$_1$, *Term*$_2$, ... are also terms.
- A list [*Term*$_1$, *Term*$_2$, ...] is a term, provided that *Term*$_1$, *Term*$_2$, ... are also terms.

Examples of terms are: `747` and `departure(flight(123), "12:35")`. It is important to stress that terms provide a simple, but versatile, mechanism for representing arbitrary data.

We distinguish two kinds of *occurrences of variables*:

- *Value occurrences* of the form *V* whose value is obtained from the context in which they are used.
- *Result occurrences* of the form *V*? which get a value assigned depending on the context in which they occur; this may be either as a result of a successful match with another term, or as a result of an assignment.

For instance, in a context where variable `X` has value 3, the term `f(X)` is equivalent to `f(3)`. When, on the other hand, the terms `f(X?)` and `f(3)` are matched, the value 3 will be assigned to variable `X` as a result of this successful match.

A "**T** script" describes the complete behaviour of a system and consists of a number of definitions (for processes and tools) followed by one "ToolBus configuration".

A process definition is a named processes expression (see Figure 2 for an overview of the primitives used in process expressions). It has the form:

`process` *Pname*(*Formals*) `is` *P*

Formals are optional and contain a list of formal parameter names. *P* is a process expression.

A ToolBus *configuration* is an parallel composition of processes and has the form:

`toolbus(`*Pname*$_1$(*Formals*$_1$)`, ..., `*Pname*$_n$(*Formals*$_n$)`)`

It describes the initial configuration of processes in the ToolBus. During execution, new processes can be created using the `create` primitive. Each process is identified by a unique, dynamically generated, process identifier.

We will explain many of the primitives in Figure 2 while presenting examples later on. In particular, we will explain the relation between time primitives and other primitives in Section 3.

2.2 Example: a calculator

Informal description. Consider a calculator capable of evaluating expressions, showing a log of all previous computations, and displaying the current time.

Concurrent with the interactions of the user with the calculator, a batch process is reading expressions from file, requests their computation, and writes the resulting value back to file.

The calculator is defined as the cooperation of six processes:

- The user-interface process `UI1` can receive the external events `button(calc)` and `button(showLog)`.
  After receiving the "calc" button, the UI process is requested to provide an expression (probably via a dialogue window). This may have two outcomes: `cancel` to abort the requested calculation or the expression to be evaluated. After receiving the "showLog" button all previous calculations are displayed.
- The user-interface process `UI2` can receive the event `button(showTime)` which displays the current time. The user-interface has the property that the "showTime" button can be pushed at any time, i.e. even while a calculation is in progress. That is why the control over the user-interface is split in the two parallel processes `UI1` and `UI2`.
- The actual calculation process `CALC`.
- A process `BATCH` that reads expressions from file, calculates their value, and writes the result back on file.
- A process `LOG` that maintains a log of all calculations performed. Observe that `LOG` explicitly subscribes to "`calc`" notes.
- A process `CLOCK` that can provide the current time on request.

Note on the use of time primitives. The calculator example does *not* use the time-related primitives of the Discrete Time ToolBus. However, the example does contain a tool `clock` that deals with time. It turns out that for the detailed control of the timing aspects of distributed applications built-in primitives at the level of the T scripts are mandatory. Their use is shown in Section 3.

ToolBus *script for the calculator.*

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

We take a closer look at the definition of the `CALC` process. First, three typed variables are introduced: `Tid` (of type `calc`, a tool identifier representing the `calc`-tool, see below), `E` (a string variable representing the expression whose value is to be computed), and `V` (an integer variable representing the computed value of expressions). The first atom,

```
execute(calc, Tid?)
```

executes the calc-tool using the command (and optionally also the desired host computer) as defined in calc's tool definition. The result variable Tid gets as value a descriptor of this particular execution of the calc-tool. All subsequent atoms (e.g., snd-eval, rec-event) that communicate with this tool instance will use this descriptor as first argument. Next, we encounter a construct of the form

(rec-msg(compute, E?) ...) * delta

describing an infinite repetition of all steps inside the parentheses. Note that inaction (delta) will be avoided as long as there are other steps possible. Next, we see the atom

```
rec-msg(compute, E?)
```

for receiving a computation request from another process. Here, compute is a constant, and the variable E will get as value a string representing the expression to be computed. Next, an evaluation request goes to the calc-tool as a result of

```
snd-eval(Tid, expr(E))
```

The resulting value is received by

```
rec-value(Tid, V?)
```

Observe the combination of an ordinary variable Tid and a result variable V. Clearly, this atom should *only* match with a value event coming from the calc-tool that was executed at the beginning of the CALC process. It is also clear that V should get a value as a result of the match. A reply to the original request rec-msg(compute, E?) is then given by

```
snd-msg(compute, E, V)
```

and this is followed by the notification

```
snd-note(compute(E, V))
```

that will be used by the LOG process.
The definition for the calc tool is:

```
tool calc is {command = "./calc"}
```

The string value given for command is the operating system level command needed to execute the tool. It may contain additional arguments as can be seen in the definition of the ui-tool below.

The user-interface is defined by the process UI. First, it executes the ui-tool and then it handles three kinds of buttons. Note that the buttons "calc" and "log" exclude each other: either the "calc" button or the "log" button may be pushed but not both at the same time. The "time" button is independent of the other two buttons: it remains enabled while any of the other two buttons has been pushed.

```
process UI is
  let Tid : ui
  in
     execute(ui, Tid?) .
     ( ( CALC-BUTTON(Tid) + LOG-BUTTON(Tid) ) * delta
     ||
       TIME-BUTTON(Tid) * delta
     )
  endlet

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

The treatment of each button is defined in a separate, auxiliary, process definition. They have a common structure:

- Receive an event from the user-interface.
- Handle the event (either by doing a local computation or by communicating with other ToolBus processes that may communicate with other tools).
- Send an acknowledgement to the user-interface that the handling of the event is complete.

```
process CALC-BUTTON(Tid : ui) is
  let  N : int, E : str, V : int
  in
     rec-event(Tid, N?, button(calc)) .
     snd-eval(Tid, get-expr-dialog).
     ( rec-value(Tid, cancel)
     + rec-value(Tid, expr(E?)) .
       snd-msg(compute, E) . rec-msg(compute, E, V?) .
       snd-do(Tid, display-value(V))
     ) . snd-ack-event(Tid, N)
  endlet

process LOG-BUTTON(Tid : ui) is
  let N : int, L : term
  in
      rec-event(Tid, N?, button(showLog)) .
      snd-msg(showLog) .  rec-msg(showLog, L?) .
      snd-do(Tid, display-log(L)) .
      snd-ack-event(Tid, N)
  endlet

process TIME-BUTTON(Tid : ui) is
  let N : int, T : str
  in
      rec-event(Tid, N?, button(showTime)) .
      snd-msg(showTime) . rec-msg(showTime, T?) .
      snd-do(Tid, display-time(T)) .
      snd-ack-event(Tid, N)
  endlet
```

The BATCH process executes the batch tool, reads expressions from file, computes their value by exchanging messages with process CALC and writes an (expression, value) pair back to a file.

```
process BATCH is
  let Tid : batch, E : str, V : int
  in
     execute(batch, Tid?) .
     ( snd-eval(Tid, fromFile). rec-value(Tid, expr(E?)) .
       snd-msg(compute, E). rec-msg(compute, E, V?).
       snd-do(Tid, toFile(E, V))
     ) * delta
  endlet

tool batch is {command = "./batch"}
```

The LOG process subscribes to notes of the form compute(<str>,<int>), i.e., a function compute with a string and an integer as arguments.

```
process LOG is
  let Tid : log, E : str, V : int, L : term
  in
     subscribe(compute(<str>,<int>)) .
     execute(log, Tid?) .
     ( rec-note(compute(E?, V?)) . snd-do(Tid, writeLog(E, V))
     + rec-msg(showLog) . snd-eval(Tid, readLog) .
       rec-value(Tid, L?) . snd-msg(showLog, L)
     ) * delta
  endlet

tool log is {command = "./log"}
```

There are alternatives for the way in which the process definitions in this example can be defined. The LOG process can, for instance, be defined without resorting to a tool in the following manner:

```
process LOG1 is
  let TheLog : list, E : str, V : int
  in
     subscribe(compute(<str>,<int>)) .
     TheLog := [] .
     ( rec-note(compute(E?, V?)) . TheLog := join(TheLog, [[E, V]])
     + rec-msg(showLog) .  snd-msg(showLog, TheLog)
     ) * delta
  endlet
```

Instead of storing the log in a tool we can use a variable (TheLog) for this purpose in which we maintain a list of pairs. We use the function "join" (list concatenation) to append a new pair to the list. Note that join operates on lists, hence we concatenate a singleton list consisting of the pair as single element. The process CLOCK executes the clock tool and answers requests for the current time.

```
process CLOCK is
  let Tid : clock, T : str
  in
     execute(clock, Tid?) .
     ( rec-msg(showTime) .
       snd-eval(Tid, readTime) .
       rec-value(Tid, T?) .
       snd-msg(showTime, T)
     ) * delta
  endlet

tool clock is {command = "./clock"}
```

Finally, we define one of the possible TOOLBUS configurations that can be defined using the above definitions:

```
toolbus(UI, CALC, LOG1, CLOCK, BATCH)
```

3 A distributed auction

3.1 Preliminaries

Before turning our attention to an example where the use of time at the level of the **T** script is essential, we need to explain how the time primitives interact with other primitives.

The following attributes can be attached to atomic processes, in order to define their behaviour in time:

- `delay`: relative execution delay.
- `abs-delay`: absolute execution delay.
- `timeout`: relative timeout for execution.
- `abs-timeout`: absolute timeout for execution.

We only permit the following combinations of these attributes:

- relative time: `delay`, `delay/timeout`, `timeout`.
- absolute time: `abs-delay`, `abs-delay/abs-timeout`, `abs-timeout`.

Other combinations, e.g., mixtures of relative and absolute time are forbidden. Note that time is determined by the actual clock time of the TOOLBUS and not by the clocks of the tools, since these may be executing on different computers and their clocks are likely to be in conflict with each other.

A typical example is

```
rec-msg(compute, E?) delay(sec(10))
```

which becomes enabled after 10 seconds and is then identical to

```
rec-msg(compute, E?)
```

if b then X else Y fi	=	if b then X fi + if not(b) then Y fi
if true then X fi	=	X
if false then X fi	=	delta

Fig. 3. Axioms for conditionals in **T** scripts.

More complex behaviour can be defined by combining time primitives and conditionals. The behaviour of the conditional constructs in **T** scripts is defined in Figure 3. Now consider the fragment

```
if not(or(Final, Sold)) then
   snd-note(any-higher-bid) delay(sec(10))
fi
```

In this example, the `snd-note` can only become enabled when the test of the conditional yields `true` *and* at least 10 seconds have passed. In the auction example, we will see the use of several choices between conditionals of the above form, each with its own Boolean and timing constraints.

3.2 Informal description

Consider a completely distributed auction in which the auction master (auctioneer) and the bidders are cooperating via a workstation in their own office. The problem is how to synchronize bids, how to inform bidders about higher bids, and how to decide when the bidding is over. In addition, bidders may connect and disconnect from the auction whenever they want.[4]

The auction is defined by the following processes:

- The auction is initiated by the process `Auction` which executes the "master" tool (the user-interface used by the auction master) and then handles connections and disconnections of new bidders, introduction of a new item for sale to the auction, and the actual bidding process. A `delay` is used to determine the end of the bidding activity per item.
- A `Bidder` process is created for each new bidder that connects to the auction; it describes the possible behaviour of the bidder.

This example illustrates the dynamic connection/disconnection of tools and the use of time.

[4] This example is an extension of the example given in [Yel94], where it was used in the context of protocol conversion and the generation of protocol adapters. We have added certain features, e.g., dynamic connection and disconnection of bidders and time considerations, to approximate the behaviour of a "real" auction.

3.3 T script for auction

The overall steps performed during an auction are described by the process Auction.

```
process Auction is
  let Mid : master, Bid : bidder
  in
      execute(master, Mid?) .        %% execute the master tool
      ( ConnectBidder(Mid, Bid?)     %% repeat:  add new bidder
                                     %%          between sales,
      +                              %%        or
        OneSale(Mid)                 %%          perform one sale
      ) *
      rec-event(Mid, quit) .         %% until auction master quits
      shutdown("Auction is closed") %% close the auction
  endlet

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

The auxiliary process ConnectBidder handles the connection of a new bidder to the auction. It takes the following steps:

- Receive a connection request from some bidder. This may occur when someone executes a bidder tool outside the ToolBus (may be even on another computer). As part of its initialization, the bidder tool will attempt to make a connection with some ToolBus (the particular ToolBus is given as a parameter when executing the bidder tool).
- Create an instance of the process Bidder that defines the behaviour of this particular bidder.
- Ask the bidder for its name and send that to the auction master.

```
process ConnectBidder(Mid : master, Bid : bidder?) is
  let Pid : int, Name : str
  in
     rec-connect(Bid?) .             %% receive connection request from
                                     %% new bidder
     create(Bidder(Bid), Pid?) .     %% create a new Bidder process
     snd-eval(Bid, get-name) .       %% ask bidder for its name, and send
     rec-value(Bid, Name?) .         %% it to the master tool
     snd-do(Mid, new-bidder(Bid, Name))
  endlet
```

The auxiliary process OneSale handles all steps needed for the sale of one item:

- Receive an event from the master tool announcing a new item for sale.
- Broadcast this event to all connected bidders and perform one of the following steps as long as the item is not sold:
  - receive a new bid;
  - connect a new bidder;

- ask for a final bid if no bids were received during the last 10 seconds;
- declare the item sold if no new bids arrive within 10 seconds after asking for a final bid.

The process definition is:

```
process OneSale(Mid : master) is
  let Descr : str,       %% Description of item for sale
      InAmount : int,    %% Initial amount for item
      Amount : int,      %% Current amount
      HighestBid : int,  %% Highest bid so far
      Final : bool,      %% Did we already issue a final call for bids?
      Sold : bool,       %% Is the item sold?
      Bid : bidder       %% New bidder tool connected during sale
  in
      rec-event(Mid, new-item(Descr?, InAmount?)) .
      HighestBid := InAmount .
      snd-note(new-item(Descr, InAmount)) .
      Final := false . Sold := false .
      ( if not(Sold) then
            rec-msg(bid(Bid?, Amount?)) .
            snd-do(Mid, new-bid(Bid, Amount)) .
            if less-equal(Amount, HighestBid) then
               snd-msg(Bid, rejected)
            else
               HighestBid := Amount .
               snd-msg(Bid, accepted) .
               snd-note(update-bid(Amount)) .
               snd-do(Mid, update-highest-bid(Bid, Amount)) .
               Final := false
            fi
        fi
      +
        if not(or(Final, Sold)) then
           snd-note(any-higher-bid) delay(sec(10)) .
           Final := true
        fi
      +
        if and(Final, not(Sold)) then
           snd-note(sold(HighestBid)) delay(sec(10)) .
           Sold := true
        fi
      +
        ConnectBidder(Mid, Bid?) .  %% add new bidder during a sale
        snd-msg(Bid, new-item(Descr, HighestBid)) .
        Final := false
      ) *
      if Sold then snd-ack-event(Mid, new-item(Descr, InAmount)) fi
  endlet
```

The Bidder process defines the behaviour of one bidder.

```
process Bidder(Bid : bidder) is
  let Descr : str,       %% Description of current item for sale
      Amount : int,      %% Current amount
      Acceptance : term %% Acceptance/rejection of our last bid
  in
     subscribe(new-item(<str>, <int>)) . subscribe(update-bid(<int>)) .
     subscribe(sold(<int>)) . subscribe(any-higher-bid) .
     ( ( rec-msg(Bid, new-item(Descr?, Amount?))
       +
         rec-note(new-item(Descr?, Amount?))
       +
         rec-disconnect(Bid) . delta
       ) .
       snd-do(Bid, new-item(Descr, Amount)) .
       ( rec-event(Bid, bid(Amount?)) .
         snd-msg(bid(Bid, Amount)) . rec-msg(Bid, Acceptance?) .
         snd-do(Bid, accept(Acceptance)) .
         snd-ack-event(Bid, bid(Amount))
       +
         rec-note(update-bid(Amount?)) . snd-do(Bid, update-bid(Amount))
       +
         rec-note(any-higher-bid) . snd-do(Bid, any-higher-bid)
       +
         rec-disconnect(Bid) . delta
       ) *
       rec-note(sold(Amount?)) . snd-do(Bid, sold(Amount))
     )
     * delta
  endlet

tool bidder(Name : str) is
  { command = "wish-adapter -script bidder.tcl -script-args -name Name" }
```

The complete auction is, finally, defined by the ToolBus configuration:

```
toolbus(Auction)
```

4 Discrete time process algebra for the ToolBus

4.1 Preliminaries

Following [BB96] we assume that time is divided in a countably infinite number of discrete slices. In the case of the ToolBus, the unit is a second and the first slice ranges from 0 to 1. With $a^{\vee}(n+1)$ we denote a process that may perform the action a during the course of time slice number $n+1$ or, alternatively, it may idle indefinitely. The $n+1$-th time slice takes time from $time = n$ to $time = n+1$. Within a slice actions take place in interleaved fashion.

We follow discrete time process algebra in [BB96] in our explanation. In the syntax used there, we have $a^\vee(n+1) = \underline{a}(n+1) + \delta$ where $\underline{a}(n+1)$ is a process that *must* perform a in time slice $n+1$ and δ is a process that idles forever.

To model **T** scripts, it suffices to work in a subalgebra of the parametric time process algebra of [BB96]. This subalgebra is the *time-stop-free parametric time process algebra* and is is generated by time-stop-free actions $a^\vee(n)$ rather than $\underline{a}(n)$. Parametric discrete time processes allow initialisation at any time $t \in N$. With $n \gg P$ we denote the process that P develops into after initialization at n. $n \gg P$ itself is a so called absolute discrete time process. Its actions are all timed with reference to the same initial time 0

Using time spectrum abstraction we introduce parametric time processes: $P = \surd_d x.F$. When initialised at n this P behaves as $P[n/x]$ (or, more precisely, as $n \gg P[n/x]$). Time spectrum abstraction in the real time case was introduced in [BB93]. The discrete time case occurs in [BB92, BB96].

Two parametric time processes are equal (in absolute time) if they are equal after all possible initializations. In this way equality is reduced to the simple notion of strong bisimulation for absolute discrete time transition systems. The Extensionality for Parametric Discrete Time rule

$$\frac{\text{for all } n : n \gg X = n \gg Y}{X = Y} \qquad \text{(EPDT)}$$

embodies this version of process equality in parametric time. Notice that relative time notation is easily obtained on the basis of time spectrum abstraction: $a^\vee[n] = \surd_d x.a^\vee(n+x)$. Hence, $a^\vee[n]$ can perform a in slice "n after initialization" or idle.

Using $a^\vee(x)$, time spectrum abstraction, and infinite sums $\sum_{i \in N} P_i$ we can define the meaning of the actions $script(a)$ (excluding the time primitives) that occur in the TOOLBUS by

$$script(a) = \sum_{i \in N} a^\vee(i+1).$$

This means that to explain a as an action in a **T** script in the discrete time setting, we replace it by an infinite sum of $a^\vee$'s. We will write a for $script(a)$ when no confusion arizes.

a absdelay(e)	$= \surd_d n. \sum_{i \in N}(i \geq e[n/time]) :\rightarrow a^\vee(i)$
a abstimeout(e)	$= \surd_d n. \sum_{i \in N}(i < e[n/time]) :\rightarrow a^\vee(i)$
a absinterval(e_1, e_2)	$= \surd_d n. \sum_{i \in N}(i \geq e_1[n/time]) \wedge (i < e_2[n/time]) :\rightarrow a^\vee(i)$
a delay(e)	$= \surd_d n. \sum_{i \in N}(i \geq e[n/time] + n) :\rightarrow a^\vee(i)$
a timeout(e)	$= \surd_d n. \sum_{i \in N}(i < e[n/time] + n) :\rightarrow a^\vee(i)$
a interval(e_1, e_2)	$= \surd_d n. \sum_{i \in N}(i \geq e_1[n/time] + n) \wedge (i < e_2[n/time] + n) :\rightarrow a^\vee(i)$

Fig. 4. Timed atoms

Basic Process Algebra (BPA)

Axiom	
$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y).z = x.z + y.z$	A4
$(x.y).z = x.(y.z)$	A5

Deadlock (BPA_δ)

Axiom	
$x + \delta = x$	A6
$\delta.x = \delta$	A7

Free merge operator

Axiom	
$x \parallel y = x \mathbin{\|\!\!\lfloor} y + y \mathbin{\|\!\!\lfloor} x$	M1
$a \mathbin{\|\!\!\lfloor} (n \gg x) = a.(n \gg x)$	M2
$a.x \mathbin{\|\!\!\lfloor} (n \gg y) = a.(x \parallel (n \gg y))$	M3
$(x + y) \mathbin{\|\!\!\lfloor} z = x \mathbin{\|\!\!\lfloor} z + y \mathbin{\|\!\!\lfloor} z$	M4

Merge operator

Axiom	
$a \mid b = \gamma(a, b)$, if γ defined	CF1
$a \mid b = \delta$, otherwise	CF2
$x \parallel y = x \mathbin{\|\!\!\lfloor} y + y \mathbin{\|\!\!\lfloor} x + x \mid y$	CM1
$a \mathbin{\|\!\!\lfloor} (n \gg x) = a.(n \gg x)$	CM2
$a.x \mathbin{\|\!\!\lfloor} (n \gg y) = a.(x \parallel (n \gg y))$	CM3
$(x + y) \mathbin{\|\!\!\lfloor} z = x \mathbin{\|\!\!\lfloor} z + y \mathbin{\|\!\!\lfloor} z$	CM4
$a.x \mid b = (a \mid b).x$	CM5
$a \mid b.x = (a \mid b).x$	CM6
$a.x \mid b.y = (a \mid b).(x \parallel y)$	CM7
$(x + y) \mid z = x \mid z + y \mid z$	CM8
$x \mid (y + z) = x \mid y + x \mid z$	CM9

Encapsulation operator

Axiom	
$\partial_H(a) = a$, if $a \notin H$	D1
$\partial_H(a) = \delta$, if $a \in H$	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\partial_H(x.y) = \partial_H(x).\partial_H(y)$	D4

Renaming operator

Axiom	
$\rho_f(\delta) = \delta$	RN0
$\rho_f(a) = f(a)$	RN1
$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$	RN2
$\rho_f(x.y) = \rho_f(x).\rho_f(y)$	RN3
$\rho_{id}(x) = x$	RR1
$\rho_f \circ \rho_g(x) = \rho_{f \circ g}(x)$	RR2

Process creation operator

Axiom	
$E_\phi(a) = a$, if $a \notin cr(D)$	CR1
$E_\phi(cr(d)) = \overline{cr}(d).E_\phi(\phi(d))$ for $d \in D$	CR2
$E_\phi(a.x) = a.E_\phi(x)$, if $a \notin cr(D)$	CR3
$E_\phi(cr(d).x) = \overline{cr}(d).E_\phi(\phi(d) \parallel x)$ for $d \in D$	CR4
$E_\phi(x + y) = E_\phi(x) + E_\phi(y)$	CR5

State operator

Axiom	
$\lambda_S(\delta) = \delta$	SO1
$\lambda_S(a) = a(S)$	SO2
$\lambda_S(a.x) = a(S).\lambda_{S(a)}(x)$	SO4
$\lambda_S(x + y) = \lambda_S(x) + \lambda_S(y)$	SO5

Iteration operator

Axiom	
$x^* y = x.(x^* y) + y$	I

Conditional control

Axiom	
$T :\rightarrow x = x$	C1
$F :\rightarrow x = \delta$	C2

Fig. 5. Untimed Process Algebra axioms

$a^\vee(0)$	$= \delta$
$\delta^\vee(k)$	$= \delta$
$a^\vee(k+1).x$	$= a^\vee(k+1).k \gg x$

Fig. 6. Timed atoms (primitives)

$k \gg a$	$= a^{\vee}(k+1) + (k+1) \gg a$
$k \gg a^{\vee}(l)$	$= a^{\vee}(l)$ if $k < l$, otherwise δ
$k \gg (x+y)$	$= k \gg x + k \gg y$
$k \gg (x.y)$	$= (k \gg x).y$
$k \gg (x \parallel y)$	$= (k \gg x) \parallel (k \gg y)$
$k \gg (x \lfloor\!\lfloor y)$	$= (k \gg x) \lfloor\!\lfloor (k \gg y)$
$k \gg (x \mid y)$	$= (k \gg x) \mid (k \gg y)$
$k \gg \partial_H(x)$	$= \partial_H(k \gg y)$
$k \gg \rho_f(x)$	$= \rho_f(k \gg x)$
$k \gg \lambda_S(x)$	$= \lambda_S(k \gg x)$
$k \gg E_\phi(x)$	$= E_\phi(k \gg x)$
$k \gg l \gg x$	$= l \gg x$ if $k \leq l$
$k \gg l \gg m \gg x$	$= max(k,l) \gg m \gg x$
$k \gg a\ absdelay(e)$	$= max(k, e[k/time]) \gg a$
$k \gg a\ abstimeout(e)$	$= e[k/time] > k :\rightarrow$ $(a^{\vee}(k+1) + (k+1) \gg a\ abstimeout(e[k/time]))$
$k \gg a\ absinterval(ae_1, e_2)$	$= max(k, e_1[k/time]) \gg a\ abstimeout(e_2[k/time])$
$k \gg a\ delay(e)$	$= (k + e[k/time]) \gg a$
$k \gg a\ timeout(e)$	$= e[k/time] > 0 :\rightarrow$ $(a^{\vee}(k+1) + (k+1) \gg a\ timeout(e[k/time] - 1))$
$k \gg a\ interval(e_1, e_2)$	$= (k + e_1[k/time]) \gg a\ timeout(e_2[k/time])$

Fig. 7. Initialization

Timed atoms are described in table 4. Note that e is an expression with free variable *time*, and the empty sum equals δ.

In the axiomatization below we will avoid time spectrum abstraction and infinite sums, thus obtaining equations that are closer to an implementation though (perhaps) less intuitive.

4.2 Untimed process algebra axioms

For reference purposes, we include here first in Figure 5 the standard (untimed) process algebra axioms as given in [BK94]. Note that in this table:

- a ranges over all possible time-stop-free atomic actions (both $script(a)$ and $a^{\vee}(n)$).
- BPA consists of A1 ... A5.
- PA consists of BPA plus free merge operator.
- ACP consists of BPA_δ plus merge and encapsulation.
- Axioms M1, M2, (CM2), M3 (CM3) have been modified using $n \gg x$ instead of x. Here, $n \gg x$ denotes the initialization of x in n. In the absence of time bound actions, $n \gg x$ is just x. This modification guarantees consistency with the setting involving time bounds.

In Figures 6, 7 and 8 we extend the above table with additional axioms for discrete time process algebra. In the definition of the state operator, $a(S)^{\vee}(k)$ is

$a^{\vee}(k) \mid b^{\vee}(l)$	$= \delta$ if $k \neq l$
$a^{\vee}(k) \mid b^{\vee}(k)$	$= (a \mid b)^{\vee}(k)$
$\partial_H(a^{\vee}(k))$	$= a^{\vee}(k)$ if $a \notin H$
$\partial_H(a^{\vee}(k))$	$= \delta$ if $a \in H$
$\rho_f(a^{\vee}(k))$	$= (f(a))^{\vee}(k)$
$\lambda_S(a^{\vee}(k))$	$= a(S)^{\vee}(k)$
$\lambda_S(a^{\vee}(k).x)$	$= a(S)^{\vee}(k).\lambda_{S(a)}(x)$
$E_\phi(a^{\vee}(k))$	$= a^{\vee}(k)$ if $a \notin \mathrm{cr}(D)$
$E_\phi(\mathrm{cr}(d)^{\vee}(k))$	$= \overline{\mathrm{cr}}(d)^{\vee}(k).E_\phi(\phi(d))$
$E_\phi(\mathrm{cr}(d)^{\vee}(k).x)$	$= a^{\vee}(k).E_\phi(x)$
$E_\phi(\mathrm{cr}(d)^{\vee}(k).x)$	$= \overline{\mathrm{cr}}(d)^{\vee}(k).E_\phi(x \parallel \phi(d))$

Fig. 8. Axioms for other operators that modify atoms

the action that takes place if within the scope of λ_S a takes place in time slice k. $S(a)$ is the new state after performing a in state S during time slice k. Observe that all axioms where some operator distributes over + can also be applied to infinite sums, e.g.,

$$(\sum_i X_i).Y = \sum_i (X_i.Y)$$

This is helpful to rewrite the expressions that emerge if **T** scripts are assigned a meaning by means of the definitions given in Section 4.1.

5 Discussion

The Discrete Time ToolBus is a satisfactory extension of the original, untimed, ToolBus, but many further extensions can be imagined, such as:

- Capturing the influence of monitoring and debugging on the time behaviour of a system.
- Describing time behaviour with arbitrary precision ("Real Time ToolBus"): this is both conceptually and technically an open problem and constitutes an interesting research area.
- Which security concepts are needed in a coordination architecture?
- How can transaction monitoring and crash recovery be incorporated in a coordination architecture?

References

[BB92] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra (extended abstract). In *Proceedings of CONCUR'92*, LNCS 630. Springer Verlag, 1992.

[BB93] J.C.M. Baeten and J.A. Bergstra. Real space process algebra. *Formals Aspects of Computing*, 5(6):481–529, 1993.

[BB96] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 1996. To appear.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information & Control*, 60:82–95, 1984.

[BK94] J.A. Bergstra and P. Klint. The ToolBus—a component interconnection architecture. Technical Report P9408, Programming Research Group, University of Amsterdam, 1994.

[BK95] J.A. Bergstra and P. Klint. The Discrete Time ToolBus. Technical Report P9502, Programming Research Group, University of Amsterdam, 1995.

[BK96] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.

[Bri87] E. Brinksma, editor. *Information processing systems–open systems interconnection–LOTOS–a formal description technique based on the temporal ordering of observational behaviour.* 1987. ISO/TC97/SC21.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, 1990.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Kli96] P. Klint. A guide to ToolBus programming. Technical report, Programming Research Group, University of Amsterdam, 1996. to appear.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, pages 85–139, 1990.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic specification of communication protocols*, volume 36 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993.

[Oli96a] P. Olivier. Embedded system simulation – testdriving the ToolBus. Technical Report P9601, Programming Research Group, University of Amsterdam, 1996.

[Oli96b] P. Olivier. A simulator framework for embedded systems. In *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 436–439, 1996.

[vDHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping, An Algebraic Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Co., 1996. To appear.

[Yel94] D.M. Yellin. Interfaces, protocols and the semi-automatic construction of software adaptors. Technical Report RC19460, IBM T.J. Watson Research Center, 1994.

Note. The citations [BK94, BK95, BK96] contain an extensive list of references to related work. This paper is an extended abstract of [BK95].