The Discrete Time ToolBus — a software coordination architecture —

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 minor and a major example of its use: a calculator and a distributed auction. Finally, we sketch a framework for describing the operational behavior of the ToolBus, and conclude with a survey of implementation aspects and applications.

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 will now first discuss related work and briefly sketch our approach (Section 2). Next we give an overview of the ToolBus coordination architecture (Section 3) and an annotated example not involving time features (Section 4). After this introduction follows a major example that makes essential use of time: a distributed auction (Section 5). Next we sketch a framework for the description and interpretation of ToolBus-based applications (Section 6) and we also discuss implementation aspects. A discussion (Section 7) concludes the paper.

2 Related work in coordination languages and tool integration

First we will briefly sketch work in the field of coordination languages and tool integration and relate it to the ToolBus approach. For a discussion of the design issues in coordination languages we refer to [GC92]. For recent collections of research papers on this topic we refer to [AHM96, CH96]. A survey of interdisciplinary aspects of coordination can be found in [MC94].

2.1 Data integration

In its full generality, the data integration problem amounts to exchanging (complicated) data values among tools that have been implemented in different programming languages. The common approach to this problem is to introduce an *intermediate data description language*, like ASN-1 [ASN87] or IDDL [Sno89], and define a bi-directional conversion between data structures in the respective implementation languages and a common, language-independent, data format.

Instead of providing a general mechanism for representing the data in arbitrary applications, we will use a single, fixed, data representation based on term structures. We do not allow the exchange of arbitrary data structures, but insist that all data are represented in the same term format before they can be exchanged between tools. 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 data formats and conventions and those of the ToolBus.

2.2 Control integration

The integration of the control of different tools can vary from loosely coupled to tightly coupled systems. A loose coupling is, for instance, achieved in systems based on broadcasting or object-orientation: tools can notify other tools of certain changes in their internal state, but they have no further means to interact. A tighter coupling can be achieved using remote procedure calls. The tightest coupling is possible in systems based on general message passing.

Broadcasting. The Field environment developed by Reiss [Rei90] has been the starting point of work on several software architectures for tool integration. In these broadcast-based environments tools are independent agents, that interact with each other by sending messages. The distinguishing feature of Field is a centralized message server, called *Msg*, which routes messages between tools. Each tool in the environment registers with *Msg* a set of message patterns that indicate the kinds of messages it should receive. Tools send messages to *Msg* to announce changes that other tools might be interested in. *Msg* selectively broadcasts those messages to tools whose patterns match those messages. Variations on this approach can be found in [Ger88, GI90]. In [Clé90] an approach based on signals and tool networks is described which has been further developed into the Sophtalk system [BJ93]. In [Boa93] the SPLICE system is described, a network-based approach in which each component is controlled by an "agent" and agents communicate with each other through global broadcasting. These and similar approaches lead to a new, modular, software structure and make it possible to add new tools dynamically without the need to adjust existing ones. A major disadvantage of most of these approaches is that the tools still contain control information and this makes it difficult to understand and debug such event-driven networks. In other words, there is *insufficient global control* over the flow of control in these networks. An approach closely related to broadcasting is *blackboarding*: tools communicate with each other via a common global database [EM88].

Object-orientation. Similar in spirit are object-oriented frameworks like the *Object Request Broker Architecture* proposed by the Object Management Group [ORB93] or IBM's *Common Blue Print* [IBM93]. They are based on a common, transparent, architecture for exchanging and sharing data objects among software components, and provide primitives for transaction processing and message passing. The current proposals are very ambitious but not yet very detailed. In particular, issues concerning process cooperation and concurrency control have not yet been addressed in detail. These efforts reflect, however, the commercial interest in re-usability, portability and interoperability.

Remote procedure calls. In systems based on remote procedure calls, like [Gib87, BCL+87], the general mode of operation is that a tool executes a remote procedure call and waits for the answer to be provided by a server process or another tool. This approach is well suited for implementing *client/server* architectures. The major advantage of this approach is that flow of control between tools stays simple and that deadlock can easily be avoided. The major disadvantage, however, is that the model is too simple to accommodate more sophisticated tool interactions requiring, for instance, nested remote procedure calls. See, for instance, [TvR85, BJ94] for an overview of these and related issues in the context of distributed operating systems.

General message passing. The most advanced tool integration can be achieved in systems based on general message passing. In SunMicrosystems'

ToolTalk [TOO92], data integration as well as generic message passing are available. For each tool the names and types of the incoming and outgoing messages are declared. However, a description of the message interactions between between tools is not possible.

Another system in this category is Polygen, described in [WP92], where a separate description is used of the permitted interactions between tools. From this description, *stubs*[4] are generated to perform the actual communication. The major advantage of this approach is that the tool interactions can be described independently from the actual, underlying, communication mechanisms. The major disadvantage of this particular approach is that the interactions are defined in an ad hoc manner, that precludes further analysis of the interaction patterns like, for instance, the study of the dead lock behavior of the cooperating tools.

The Manifold [Arb96] language uses events and data streams through named ports as communication mechanisms. Coordination is described by means of transition-diagrams. The ToolBus has many objectives in common with Manifold, although the technical details are largely different: process descriptions based on process algebra versus transition diagrams, a different data model (terms versus bit strings), and different implementation techniques (direct interpretation of **T** scripts versus compilation and linkage-edit time configuration of modules).

The hardware metaphor. Although the analogy between methods for the interconnection of hardware components and those for connecting software components has been used by various authors, it turns out that more often than not approaches using the same analogy are radically different in their technical contents. For instance, in the Eureka Software Factory (ESF) a "software bus" is proposed that distinguishes the roles of tools connected to the bus, like, e.g., user-interface components and service components. As such, this approach puts more emphasis on the structural decomposition of a system then on the communication patterns between components. See [SvdB93] for a more extensive discussion of these aspects of ESF. A similar approach is Atherton's Software Backplane described in [Bla93], which takes a purely object-oriented approach towards integration.

In [Pur94], Purtillo proposes a software interconnection technology based on the "Polylith software bus". This research shares many goals with the work we present in this paper, but the perspectives are different. Purtillo takes the static description of a system's structure as starting point and extends it to also cover the system's runtime structure. This leads to a module interconnection language that describes the logical structure of a system and provides mappings to essentially different physical realizations of it. One application is the transparent transportation of software systems from one parallel computer architecture to another one with different characteristics. We take the communication patterns between components as starting point and therefore primarily focus on a system's run-time structure. Another difference is the prominent role of formal

[4] Small pieces of interfacing software.

process specifications in our approach.

The notion of "Software IC's" is proposed by several authors. For instance, [Cox86] uses it in a purely object-oriented context, while [Clé90] describes a communication model based on broadcasting (see above).

Other paradigms. Various other solutions have been proposed. For instance, Linda [CG89] uses a shared tuple space as general mechanism for communication and synchronization. The language is based on two important principles: (a) computation and communication are orthogonal aspects of programming and should be treated independently; (b) flexibility through uncoupling of components. Various Linda implementations are available that extend existing programming languages with Linda's tuple operations. A related language is Gamma [BM90]; it is based on multi-set transformations. We refer to [AHM96, CH96] for various papers related to the use of Linda and Gamma for coordination.

We see as a general disadvantage of these two particular approaches that the problem to be solved has to be moulded to fit the given data structure (tuple/multi-set) of the underlying coordination language. In our work we prefer to explore a process-oriented view on coordination.

Control integration in the ToolBus. The control integration between tools is achieved by using process-oriented "**T** scripts" that model the possible interactions between tools. The major difference with other approaches is that we use one, formal, description of *all* tool interactions. Coordination and computation are strictly separated: inside the ToolBus a varying number of parallel processes takes care of the coordination while all actual computation is performed in tools (and not in the ToolBus itself). We uncouple the coordination activities inside the ToolBus by using pattern matching to establish communication between processes rather than using explicitly named communication ports. We support heterogeneity, since tools implemented in different languages running on different machines can be coordinated by way of a single ToolBus.

2.3 The relation with Module Interconnection Languages

Module Interconnection Languages [PDN86] and modules in programming languages are the classical solution to the problem of decomposing large software systems into smaller components. Modules can *provide* certain operations to be used by other modules and they can *require* operations from other modules. It is the task of the Module Interconnection Language (or the module mechanism) to establish a type-safe connection between provided and required operations. The dynamic behavior of modules is usually not taken into account, e.g., the fact that the proper use of a "stack" module implies that first a "push" operation has to be executed before a "pop" operation is allowed.

The approach to component interconnection to be presented in this paper, *concentrates* on these dynamic, behavioral, aspects of modules. It shares many of the objectives of the work on "formal connectors" [AG94], where (untimed) CSP is used to describe software architectures. Their work is more ambitious than

ours, since it aims at describing *arbitrary* software architectures, while we use a fixed (bus-oriented) architecture. The mechanisms we use to configure our bus architecture are, however, more powerful than the ones described in [AG94] (i.e., dynamic process creation, dynamic connection and disconnection of components, time).

2.4 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 *behavior 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 sent 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, C++, Lisp, Java, Tcl, Python, 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.

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 data formats 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 data types. Compare this with the abstract data types 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] 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 [BK96a]. A redesign of the ToolBus is fully described in [BK95]. In this paper, we concentrate on giving an overview of the new ToolBus design by way of examples. Larger applications of the ToolBus are described in [DG95, Oli96a, Oli96b]. A guide to ToolBus programming can be found in [Kli96].

3 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 variable number of *tools* T_i ($i = 1, ..., m$) that are to be combined into a complete system. The internal behavior 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 behavior 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.

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 behavior 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.

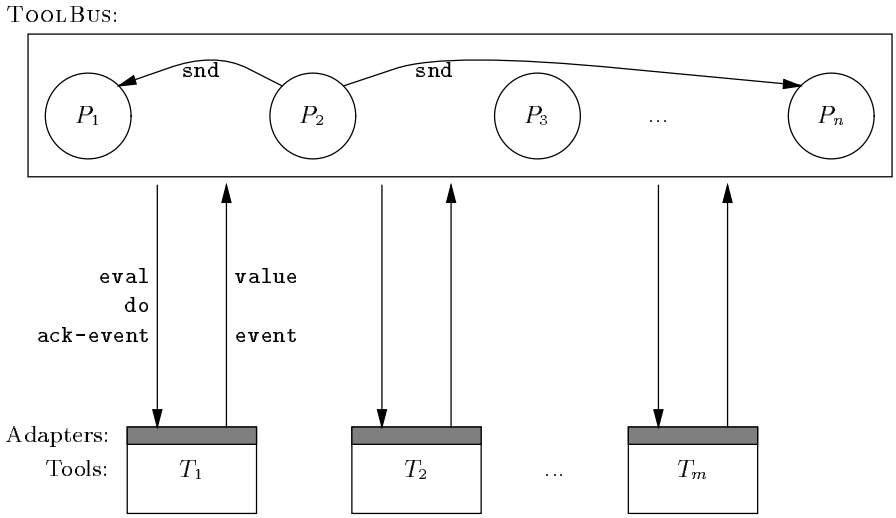

Fig. 1. Global organization of the ToolBus

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. There is no direct communication possible between tools.

3.1 Overview of T scripts

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

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.

- 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 behavior 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 5.

4 An introductory example: a calculator

4.1 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:

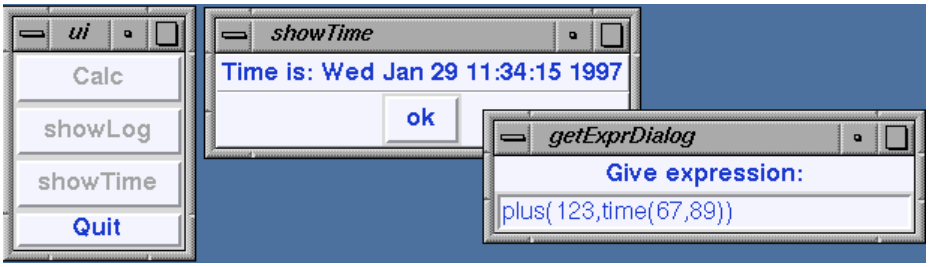

Fig. 3. The calculator application.

- 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 dialog 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.

In Figure 3 we see a snapshot of the calculator application. On the left, the main menu of the application is shown; it has the form of a list of buttons. The user is at this moment engaged in two simultaneous dialogs: Pushing the `showTime` button has resulted in a message showing the current time; this dialog could be completed by pushing the `ok` button. Instead, the user has pushed the `Calc` button and is now typing an expression in the dialog window. Note that `Quit` is the only button that is still available to the user (shown in boldface, the unavailable buttons are shown in grey).

4.2 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 5.

4.3 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)
```

4.4 Concluding remarks

As mentioned earlier, a snapshot of the calculator-in-action can be seen in Figure 3. The user-interface was implemented using Tcl/Tk while the other tools were implemented in C. A further discussion of implementation issues is postponed until Section 6.3.

5 A distributed auction

5.1 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 behavior 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?)
```

More complex behavior can be defined by combining time primitives and conditionals. The behavior of the conditional constructs in **T** scripts is defined in Figure 4. 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.

5.2 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,

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. 4. Axioms for conditionals in **T** scripts.

and how to decide when the bidding is over. In addition, bidders may connect and disconnect from the auction whenever they want.[5]

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 behavior of the bidder.

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

In Figures 5, 6, and 7 we see the auction *in action*. In Figure 5 the auction master (running on machine A) has initiated the sale of a bicycle for an initial price of $100. Bidder Paul has been connected to the auction (his bidder tool is running on machine B) and he has made a bid of $110 that was accepted (Figure 6). Bidder Jan has been connected (his bidder tool is running on machine C) and he is observing the progress of the auction (Figure 7). The auction master has just called for any higher bids ("Last chance to bid") and has started a time out procedure of 10 seconds.

5.3 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
```

[5] 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 behavior of a "real" auction.

Fig. 5. The master tool (executing on machine A).

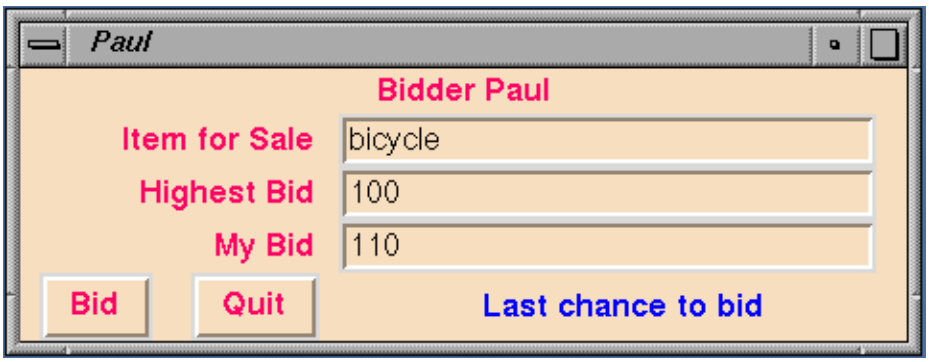

Fig. 6. The bidder tool for Paul (executing on machine B).

```
      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.

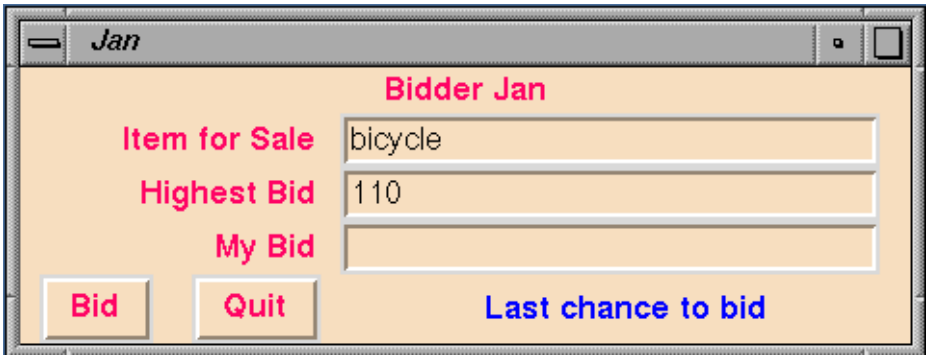

Fig. 7. The bidder tool for Jan (executing on machine C).

– 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)
```

5.4 Concluding remarks

As mentioned earlier, a snapshot of the auction-in-action can be seen in Figures 5, 6, and 7. All tools were implemented using Tcl/Tk.

A 2-page **T** script in combination with two 2-page Tcl scripts are sufficient to construct a distributed application with a functionality that would require much more effort using traditional implementation techniques. A further discussion of implementation issues is postponed until Section 6.3.

6 A framework for design and interpretation of T scripts

Our approach to the design and interpretation of **T** scripts consists of three steps: formal semantics (Section 6.1), operational behaviour (Section 6.2), and efficient implementation (Section 6.3).

6.1 Formal semantics

The formal semantics of **T** scripts has been described in [BK94] using Process Algebra with extensions for discrete time. Process Algebra (ACP) is an algebraic approach to the description of parallel, communicating, processes originally proposed in [BK84]. We refer to [BW90] for an elaborate description of Process Algebra and to [BK96b] for a description of the discrete time extensions we have used in the ToolBus.

6.2 Prototyping the operational behaviour of a ToolBus

The operational behaviour of **T** scripts has been described in [BK95] by means of an algebraically specified interpreter written in Asf+Sdf: a specification formalism for describing all syntactic and semantic aspects of (formal) languages. Support for writing Asf+Sdf specifications is given in the Asf+Sdf Meta-environment described in [Kli93]. The use of Asf+Sdf for language prototyping is documented in [vDHK96].

How to describe the operational behaviour of **T** scripts? The Process Algebra semantics given in [BK94], describes *all* possible execution paths of a given script. A usual approach to prototyping and verification would be to build a *simulator* that allows the exploration of all these possible execution paths.

Here, we take a different approach since our goal is to obtain a real implementation of the system as characterized by the script. This can only be achieved by interpreting the script in such a way that *specific* execution paths are selected. We will therefore develop an *interpreter* for **T** scripts that includes scheduling rules for selecting execution paths.

Representing a ToolBus. Our overall strategy is as follows. At any moment during interpretation each process, say process k, is represented as

$$\rho_k(\lambda_{Env}(<AP_1 + ... + AP_n>)).$$

Each AP_i is an *action-prefix form*, i.e., a process expression starting with an action, and represents a possible choice in the process. AP_i does not itself contain any +-operators. The operator λ_{Env} represents the local state of AP_k where Env is a mapping from variables to their respective values. The operator ρ represents a *renaming* that identifies all atoms as belonging to process k. All other information related to a process is maintained in a global *bus state* to be described in a moment.

The behaviour of the ToolBus can be characterized completely by the following parallel composition of all processes in the ToolBus:

$$\begin{array}{l}\lambda_{BS}(E_{Script}(\\ \quad\{\rho_1(\lambda_{Env_1}(<AP_{11} + ... + AP_{1n_1}>))||\\ \quad ...||\\ \quad \rho_m(\lambda_{Env_m}(<AP_{m1} + ... + AP_{mn_m}>))\\ \}\,)).\end{array}$$

The operator E_{Script} represents process creation where *Script* is the **T** script being executed. The operator λ_{BS} represents, finally, the global state of the ToolBus. It consists of a variable number of "bus assignments" of the form F `:=` V where F is an identifier optionally indexed with a process index, e.g., `time` or `name(`k`)`.

One interpretation step consists of selecting one alternative AP_{ij} in each process—according to certain fixed scheduling rules defined by the interpreter—and computing a new bus.

For descriptive purposes, we also model the *tools* connected to the ToolBus as processes. The interpreter as a whole thus captures the input/output

behaviour of the system described by the script: given a **T** script and events coming from the tools connected to the ToolBus, it computes responses modeled by messages to the connected tools.

When defining operations on process expressions, the standard approach is to *normalize* them, i.e., replace all operators by simpler ones thus obtaining a normal form containing a limited set of operators. The major advantage of this approach is simplicity, since more complex operators can be defined axiomatically in terms of simpler ones. Operationally, however, this approach is less suitable, since the resulting normal forms may become very large and their computation may be very expensive. In this specification we use a fixed format of process expressions (as described above) and manipulate them directly, without normalisation. This approach can be characterized as "lazy" as opposed to "eager" normalisation before interpretation.

6.3 Implementation

Implementation options. There are many methods for implementing the interpretation of **T** scripts, ranging from purely interpretative methods to fully compilational methods that first transform the **T** script into a transition table. The former are easier to implement, the latter are more efficient. For ease of experimentation we have opted for the former approach.

Another global implementation decision to be made is the way process communication is implemented. There are at least two options. First, one can use Unix "pipes" for this purpose, but this requires that all tools are child processes of the ToolBus interpreter and that interpreter and tools run on the same machine. Second, one can use general "socket" communication between processes. We have opted for this second approach to allow experiments with a client/server architecture where tools run on different machines and can be started at any moment *after* the ToolBus interpreter has been started. This requires that tools can take the initiative to make a connection with the ToolBus interpreter.

A final choice, is the way data are exchanged between ToolBus and tools. The data to be exchanged are *terms* and our approach will be to linearize a term (i.e., print it in prefix form) at the sending side and parsing it at the receiving side. In this way there is a completely standard way of sending and receiving terms which is independent of any implementation language.

T scripts are executed using *randomized execution.* This means that execution is performed in such a way that if, according to the process algebra semantics, execution can go into different directions a "non-deterministic" choice is made (involving the use of a random number generator). Using randomized execution we guarantee that process algebra equations are correctness preserving transformations on **T** scripts. This will prevent writing **T** scripts that make use of implementation dependent run-time properties of execution that may turn out to be different in new implementations.

The ToolBus *interpreter.* The ToolBus itself is implemented as a separate Unix process that interprets a given **T** script. First, syntax analysis and type-

checking of the script are done. Next, the initial ToolBus configuration is created as defined by the script the application as a whole starts executing according to the script. Any tools that have to be created are executed as a separate Unix process. We also support the case that the execution of a tool is started independently and that it connects to the ToolBus later on. An input and an output channel are created between the ToolBus and each tool. The ToolBus interpreter maintains a list of active processes and its actions are determined by internal computation steps, external events coming from one of the tools or the expiration of an internal timer (used for the implementation of the delay/timeout primitives).

Implementing tools Tools have generally the following structure: first all ToolBus related initializations are performed, a connection is made with the ToolBus interpreter, and an event handler is established for dealing with incoming events. This handler gets the incoming event in the form of a term (a predefined datatype), analyzes it, performs arbitrary application-specific computations and returns a new term that will be send back to the ToolBus. Tools can also take the initiative to generate events. A standard library is available for common operations on terms such as matching and constructing, reading and writing, and the like.

For both examples presented in this paper, the user-interfaces have been implemented using Tcl/Tk, while the other tools have been written in C.

7 Discussion

So far, the design of the ToolBus has gone through two major iterations. Improvements and changes have been based on small case studies as well as on several larger applications (see [BK96a]). To date, more than 40 applications have been built using the ToolBus in areas like multi-user, distributed, programming environments (e.g., a C development environment for embedded systems, a distributed debugging system, a new version of Asf+Sdf Meta-Environment, a constraint-based graphical editor), multi-user games, various tool interconnections (e.g., symbolic mathematics tools, proof tools), and traffic simulations.

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?

Last but not least, many challenging implementations problems have to be addressed such as the use of shared memory between tools, dynamic loading of tool

executables in the **T** script interpreter, and the use of multi-threading in the interpreter. The general spirit is to maintain the *logical* system architecture of ToolBus-based applications as presented in this paper, but to develop efficient implementation techniques to further optimize their run-time efficiency.

Acknowledgements

The first (untimed) version of the ToolBus was implemented by the second author. Pieter Olivier made significant contributions to the implementation of the second version (the Discrete Time ToolBus). We also want to thank the *users* of the ToolBus for their comments and suggestions.

References

[AG94] R. Allen and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, 1994.

[AHM96] J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors. *Coordination Programming: Mechanisms, Models and Semantics.* Imperial College Press, 1996.

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activities. In *[CH96]*, pages 34–56, 1996.

[ASN87] *Specification of Abstract Syntax Notation One (ASN-1).* 1987. ISO 8824.

[BCL+87] B. Bershad, D. Ching, E. Lazowsky, J. Sanislo, and M. Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13:880–894, 1987.

[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.

[BJ93] J. Bertot and I. Jacobs. Sophtalk tutorials. Technical Report 149, INRIA, 1993.

[BJ94] F.M.T. Brazier and D. Johansen. *Distributed open systems.* IEEE Computer Society Press, 1994.

[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.

[BK96a] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *[CH96]*, pages 75–88, 1996.

[BK96b] J.A. Bergstra and P. Klint. The discrete time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, 1996.

[Bla93] E. Black. The Atherton software backplane. In *[Dal93]*, pages 85–96, 1993.

[BM90] J. Banatre and D. Le Metayer. Programming by multiset transformations. *Science of Computer Programming*, 15:55–77, 1990.

[Boa93] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, 1993.

[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.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CH96] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, 1996.

[Clé90] D. Clément. A distributed architecture for programming environments. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 11–21, 1990. Sofware Engineering Notes, Volume 15.

[Cox86] B. Cox. *Object-oriented programming: an evolutionary approach*. Addison-Wesley, 1986.

[Dal93] R. Daley, editor. *Integration technology for CASE*. Avebury Technical, Ashgate Publishing Company, 1993.

[DG95] D. Dams and J.F. Groote. Specification and implementation of components of a μCRL toolbox. Technical Report 152, Department of Philosophy, University of Utrecht, 1995.

[EM88] R. Engelmore and T. Morgan, editors. *Blackboard systems*. Addison-Wesley, 1988.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[Ger88] C. Geretty. HP softbench: a new generation of software development tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, 1988.

[GI90] D. Garlan and E. Ilias. Low-cost, adaptable tool integration policies for integrated environments. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Develpment Environments*, pages 1–10, 1990. Sofware Engineering Notes, Volume 15.

[Gib87] P. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13:77–87, 1987.

[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.

[IBM93] Open Blueprint Introduction. Technical report, IBM Corporation, December 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

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

[MC94] T.W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[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 *[CH96]*, pages 436–439, 1996.

[ORB93] Object request broker architecture. Technical Report OMG TC Document 93.7.2, Object Management Group, 1993.

[PDN86] R. Prieto-Diaz and J.M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, 1986.

[Pur94] J.M. Purtillo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[Rei90] S. P. Reiss. Connecting tools using message passing in the Field programming environment. *IEEE Software*, 7(4), July 1990.

[Sno89] R. Snodgrass. *The Interface Description Language*. Computer Science Press, 1989.

[SvdB93] D. Schefström and G. van den Broek, editors. *Tool Integration*. Wiley, 1993.

[TOO92] Designing and writing a ToolTalk procedural protocol. Technical report, SunSoft, june 1992.

[TvR85] A.S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, 1985.

[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.

[WP92] E. L. White and J. M. Purtilo. Integrating the heterogeneous control properties of software modules. In *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, pages 99–108, 1992. Software Engineering Notes, Volume 17.

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

This article was processed using the LaTeX macro package with LLNCS style