The ToolBus 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. Building large, heterogeneous, distributed software systems poses serious problems for the software engineer; achieving interoperability of software systems is still a major challenge. We describe an experiment in designing a generic software architecture for solving these problems. 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 component interconnection architecture resembling a hardware communication bus, and therefore we will call it a "ToolBus".
We describe the coordination of tools in process-oriented "T scripts" featuring, amongst others, (1) sequential composition, choice and iteration of processes; (2) handshaking (synchronous) communication of messages; (3) asynchronous communication of notes to an arbitrary number of processes; (4) note subscription; (5) dynamic process creation. Most notably lacking are built-in datatypes: operations on data can only be performed by tools, giving opportunities for efficient implementation. In three large case studies, the ToolBus architecture has been used to build editor-interfaces with user-defined extensions, to study feature interaction in intelligent networks, and to build a simulator for traffic light control. We give an overview of these case studies and briefly sketch the evolution of the ToolBus design that incorporates the lessons we have learned from them.

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 (see, for instance, [SvdB93]), 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.

Tool integration is just one instance of the more general *component interconnection problem* in which the nature (e.g., hardware *versus* software) and granularity (e.g., natural number *versus* database) of components are left unspecified. As such, solutions to this problem may also increase our understanding of subjects like modularization, parameterization of datatypes, module interconnection languages, and structured system design.

In this paper we will pursue the more specific goal of integrating software components like user-interfaces, editors and compilers. It is generally recognized that the integration of such interactive tools requires three steps:

Data integration: how can tools exchange and share data structures representing application specific information? In its full generality, the data integration problem amounts to exchanging (complicated) data values among tools that have been implemented in different programming languages.

Control integration: how can tools communicate or cooperate with each other? 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.

User-interface integration: how can the user-interfaces of the various tools be integrated in a uniform manner? [4] Two trends in the field of human-computer interaction are relevant here:

- User-interfaces and in particular human-computer dialogues are more and more defined using formal techniques. Techniques being used are transition networks, context-free grammars and events. There is a growing consensus that dialogues should be multi-threaded (i.e., the user may be simultaneously involved in more than one dialogue at a time).
- There is also some evidence that a complete separation between user-interface and application is too restrictive.

We will now first discuss related work (Section 2), and then we give an overview of the ToolBus coordination architecture (Section 3). A description and evaluation of three case studies (Section 4) and a discussion (Section 5) complete the paper.

2 Related work in coordination languages and tool integration

In this section we briefly sketch work in the field of coordination languages and tool integration and relate it to our approach. For a discussion of the design issues in coordination languages we refer to [GC92]. 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 datastructures 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 dataformats 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

[4] Some authors like, e.g., [Dal93], call this step *presentation integration*.

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 reusability, 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*[5] 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 behaviour of the cooperating tools.

The Manifold [AHS93] language uses events and datastreams 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 prcess 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 process specifications in our approach.

[5] Small pieces of interfacing software.

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 multiset transformations.

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 datastructure (tuple/multiset) 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 User-interface integration

Two trends in the field of human-computer interaction are relevant here:

- User-interfaces and in particular human-computer dialogues are more and more defined using formal techniques. Techniques being used are transition networks, context-free grammars and events. There is a growing consensus that dialogues should be multi-threaded (i.e., the user may be simultaneously involved in more than one dialogue at a time) [Gre86].
- There is also some evidence that a complete separation between user-interface and application is too restrictive [Hil86].

We refer to [HH89] for an extensive survey of human-computer interface development and to [Mye92] for a more recent overview of the role of concurrency in languages for developing user-interfaces. Other approaches in this category that have some similarities with our approach are Abstract Interaction Tools [vdB88], Squeak [CP85], and the use of Esterel for control integration [Dis94]. Abstract Interaction Tools uses extended regular expressions to control a hierarchy of interactive tools. Squeak uses CSP to describe the behaviour of input devices like a mouse or keyboard when building user-interfaces. Experience with the Sophtalk approach we already mentioned earlier, has led to experiments to use (and extend) the synchronous parallel language Esterel for describing all control interactions between tools.

We will *not* address user-interface integration as a separate topic, but it turns out that the control integration mechanisms in the ToolBus can be exploited to achieve user-interface integration as well.

2.4 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 behaviour 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, behavioural, 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).

2.5 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 we are proposing in this paper 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] we have applied a number of established techniques (i.e., process algebra [BK84, BW90], algebraic specification using Asf+Sdf [BHK89, HHKR89], 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.

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 fixed number 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.

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 informs 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 (untyped) *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.
- $Id(Term_1, Term_2, \ldots)$ is a term, provided that $Term_1$, $Term_2$, ... are also terms.

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

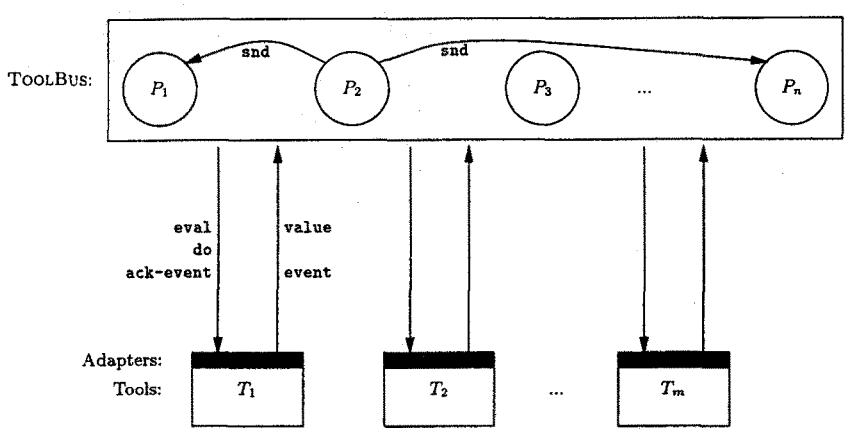

Fig. 1. Global organization of the ToolBus

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:

`define` *Pname Formals* = *P*

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

Fig. 2. Overview of ToolBus primitives.

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*$_1$`, ..., `*Pname*$_n$`)`

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.

3.2 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 external 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.

TOOLBUS *script for the calculator.*

```
define CALC =
    ( rec-msg(calc, Exp) . snd-eval(calc,Exp) . rec-value(calc, Val) .
      snd-msg(calc, Exp, Val) . snd-note(calc, Exp, Val)
    ) * delta

define UI1 =
    ( rec-event(ui(U), button(calc)) . snd-eval(ui(U), dialogGetExpr).
      ( rec-value(ui(U), cancel)
      + rec-value(ui(U), expr(Exp)) . snd-msg(calc, Exp) .
        rec-msg(calc, Exp, Val) . snd-do(ui(U), displayValue(Val))
      ) . snd-ack-event(ui(U))
    + rec-event(ui(U), button(showLog)) . snd-msg(showLog) .
      rec-msg(showLog, Log) . snd-do(ui(U), displayLog(Log)) .
      snd-ack-event(ui(U))
    ) * delta

define UI2 =
    ( rec-event(ui(U), button(showTime)) . snd-msg(showTime) .
      rec-msg(showTime, Time). (snd-do(ui(U), displayTime(Time)) .
      snd-ack-event(ui(U)))
    ) * delta

define BATCH =
    ( snd-eval(batch, fromFile). rec-value(batch, expr(Exp)) .
      snd-msg(calc, Exp). rec-msg(calc, Exp, Val). snd-do(batch, toFile(Val))
    ) * delta

define LOG =
    subscribe(calc) .
    ( rec-note(calc, Exp, Val) . snd-do(log, writelog(Exp, Val))
    + rec-msg(showLog) . snd-eval(log, readLog) .
      rec-value(log, Log) . snd-msg(showLog, Log)
    ) * delta

define CLOCK =
    ( rec-msg(showTime) . snd-eval(clock, readTime) . rec-value(clock, Time) .
      snd-msg(showTime, Time)
    ) * delta
```

Different configurations of calculator. Observe that the above definitions allow us to generate the following five, meaningful, systems:

- `toolbus(UI1, CALC, LOG)`
- `toolbus(UI1, CALC, LOG, BATCH)`
- `toolbus(UI2, CLOCK)`
- `toolbus(UI1, CALC, LOG, UI2, CLOCK)`
- `toolbus(UI1, CALC, LOG, BATCH, UI2, CLOCK)`

3.3 An experimental TOOLBUS interpreter implemented in C

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.

The ToolBus *interpreter.* The ToolBus itself is implemented as a separate Unix process that interprets a given **T** script. The following steps are taken by the interpreter:

- Syntax analysis of the script.
- Typechecking of the script: this amounts to definition/use checking of names and detection of recursive use of named processes.
- Creation of tools: execute the tool as a separate Unix process. We also support the case the execution of a tool is started independently and that it will connect to the ToolBus later on. An input and an output channel are created between the ToolBus and each tool.
- Create the toolbus configuration as defined by the script. The ToolBus interpreter maintains a list of active processes.
- Execute:
  1. wait for an event coming from one of the tools;
  2. compute the effect of the event on the ToolBus state;
  3. perform any internal communication steps;
  4. perform any atomic actions.
  5. repeat.

4 Case studies

We have experimented with the ToolBus approach in several smaller case studies, like a multi-player game, graphics constraints, and a simple syntax-directed editing environment.

However, a better assessment can only be made by applying these concepts to larger, real-life, examples. Therefore we briefly summarize here the experiences gained in three recent case studies.

The first study concerns the translation of user-interface definitions to **T** scripts. The second one explores feature interactions in intelligent networks. In the third study, the ToolBus is used for simulating a real-time, distributed algorithm for traffic regulation.

4.1 An editor-interface with user-defined extensions

In [Koo92], the specification language Seal is described for extending the user-interface of syntax-directed editors with user-defined menus and buttons. A major issue in this language (and its implementation) is how to connect parts of the user-interface with user-defined functionality and how to achieve appropriate enabling and disabling of user-interface elements.

In this first case-study [Oli94], Seal descriptions of a user-interface are translated automatically into **T** scripts. The steps to achieve this are:

- Determine which tools are needed to support the notions provided by the Seal run-time system.
- Determine a translation method from Seal to **T** scripts. Two schemes have been experimented with: (a) a scheme in which each condition is evaluated by a (generated) process expression using the operators . and + to implement the Booleans operators in conditions; (b) a scheme in which the complete condition is sent to a condition tool for evaluation.
- Compile the "user-defined functions" (specified in Asf+Sdf) to C using the techniques described in [KW93].

4.2 Feature interaction in intelligent networks

Intelligent Networks are telephone networks that provide a variety of services for extending the basic call process. An example of such a service is *call forwarding* that permits the redirection of incoming calls to another phone number. Another example is the rejection of calls coming from certain selected phone numbers. Such extra services can enhance or even modify the basic call process. A problem encountered in these networks is so-called *feature interaction*: the undesired interaction of services that have been activated during a call. The two examples just given illustrate this point: when combining them the result becomes unpredictable. With the increasing number of extra services being provided it becomes more and more difficult to predict interactions of a new service with existing ones.

To explore feature interactions in an experimental fashion, a telephone exchange based on the intelligent network model has been built using the ToolBus [Bul94]. Each telephone as well as each service are modeled as a tool.

4.3 A simulator for traffic light control

The starting point for this third case study [BKK] was a proprietary dynamic traffic regulation algorithm developed by the Dutch company *Nederland Haarlem* and a real-time process algebra specification of it. The problem at hand concerns the control of the traffic lights for a road crossing. Each road consists of three lanes for traffic, respectively, going straight on or turning to the left or to the right. Each lane is controlled by a separate traffic light. A simulation now consists of detecting incoming traffic, calculating a new state and duration for each traffic light based on given scheduling rules, and visualizing the global state of the road crossing, including the current traffic and the state of the traffic lights.

On closer inspection of the last issue, one can draw two conclusions:

- The execution speed of compiled ASF+SDF specifications is good, but the techniques used in this case study for interfacing them to the ToolBus were not yet sufficiently mature to give the desired performance. Interfacing involved several data conversions and Unix level process creations per invocation of the data tool. In the mean time we have solved this problem by considerably improving the efficiency of these interfacing techniques.
- From a more fundamental perspective, it is clear that one should not use *any* interprocess communication mechanism or data conversion to obtain, for instance, the addition of two integers as is done in the data tool in the third case study. Clearly, there is then a mismatch between the granularity of the desired operation and the overhead caused by the implementation techniques used.
  A more interesting conclusion, therefore, is that we should make a distinction between the *logical design* of a system (as embodied by a T script and the tools it requires) and its *realization*. In the current ToolBus implementation these two coincide: the logical design is mapped onto an implementation with exactly the same structure. It is therefore interesting to explore the potentials of a ToolBus *compiler* that transforms a logical design into an implementation from which as much interprocess communication has been eliminated as possible. In the most extreme case, the generated implementation could be a single, monolithic, process without any interprocess communication left!

Primitive	Description
`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. 3. *Overview of new ToolBus primitives.*

4.5 Evolution of the ToolBus design

The above observations have encouraged us to extend the ToolBus design into a "Discrete Time ToolBus" [BK95]. In Figure 3 we give a list of the new features. They can be categorized as follows:

The case study was motivated by the desire to reduce the long execution times needed when simulating the traffic regulation algorithm using standard process simulation tools and the need to visualize the large amounts of information produced by such simulations. The given real-time specification consists of a process part and a data part. Both require a different treatment in order to obtain a ToolBus based simulator. The process part is transformed as follows:

- The real-time specification is transformed into an untimed specification by introducing explicit time actions.
- The recursion used in the original specification is transformed into iteration.
- The specification resulting from the previous two steps is manually translated into a **T** script.

The data part consists of an algebraic specification of the data types needed in the process part. Since **T** scripts do not provide operations on data, it is mandatory to implement these data types as a separate tool. Two different approaches have been experimented with to obtain the data tool:

- Automatic compilation of the algebraic specification into C (also done in the first case study).
- Manual implementation of the data types in C.

As a last step, a user-interface has been built using Tcl/Tk.

4.4 Experiences

On the positive side, in all case studies except the second one concerning feature interaction, systems with the desired functionality and performance could be built at acceptable costs. This shows, at least, the potentials of the ToolBus for building complex systems and for integrating different implementation technologies. On the negative side, the following observations can be made (but see Section 4.5 for solutions to all of these problems):

- In the first case study, the lack of a conditional construct in combination with the absence of a built-in Boolean datatype resulted in a considerable increase of complexity of the generated **T** scripts. In the third study, the lack of a conditional construct complicated the translation somewhat.
- The way in which variables are treated in **T** scripts is overly restrictive: in *sending* actions all variables are replaced by their current value in the sending process; in *receiving actions* all variables get a value assigned as a result of the match with the sending action. Sometimes it is natural to combine these two mechanisms, e.g., the action `rec-msg(V,W)` in which we want to replace `V` by its current value *before* matching and assign a value to `W` *as a result of* matching. A solution is to mark all variables that should get a value as a result of matching, e.g., `rec-msg(V,W?)`. In this manner, *two-way* data exchange can be realized during a single communication between two processes.
- In the second case study, the lack of a mechanism for dynamically executing (and naming) tools was a serious obstacle.[6] Without such a mechanism, all possible combinations of telephones and services have to be executed at the moment the ToolBus configuration is created, rather than executing them on a call-by-need basis.
- In the third case study, *time* is an essential aspect of the problem. Since it could not be modeled in the **T** script itself, a separate time tool had to be introduced.
- In the third case study, interfacing of a compiled algebraic specification with the ToolBus turned out to give unacceptable performance.
- *Expressions, assignment and, conditionals.* In response to the lack of built-in datatypes (and the resulting performance problems) we provide simple operations on terms (including operations on Booleans, integers and lists). Expressions can be used in assignments and as test in conditionals. In addition, a flexible type system has been introduced that smoothly integrates static and dynamic typing of variables.
- *Time.* Many of the applications we envisage need a notion of time so we provide features for defining (relative and absolute) delays and timeouts.
- *Dynamic execution, termination, connection and disconnection of tools.* Tools can not only be executed by the ToolBus, but they can also be executed externally (even on another computer) and be connected to the ToolBus later on. Similarly, tools can be terminated or disconnected.
- *Monitoring facilities*: the construction of heterogeneous, distributed, systems is not easy, therefore we provide primitives for the (distributed) debugging of such systems. Other applications of monitoring primitives are transaction monitoring, performance monitoring, and the logging and play back of complete interactive sessions.

[6] Recall that *processes* can be created dynamically, but all *tools* are executed at the moment of start up and cannot be executed dynamically.

The process CALC as shown earlier in Section 3.2, will now be written as follows:

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

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

Apart from a slight difference in concrete syntax, the following observations can be made about this fragment:

- Local variables are now explicitly declared (with their type) in a `let` construct. This is also the case for formal parameters of process and tool declarations.
- The notion of "result occurrence" of variables (denoted by ?) has been introduced in order to mark variable occurrences that lead to an assignment to that variable. This mechanism is fully integrated with term matching. See the discussion in Section 4.4.
- Tools are now explicitly executed by the script. Execution yields a tool identifier that identifies the tool instance just created.
- Tool declarations define an executable command, but may also define additional attributes of tools (e.g., their host machine). Each tool declaration introduces a new type as can be seen in the declaration of the variable `Tid` of type `calc`. Tool declarations can be parameterized.

The Discrete Time ToolBus has been implemented and it has been tested in a variety of new applications ranging from a C development environment for embedded systems, graphic constraints, multi-player games, and adapters for a variety of languages (e.g., ASF+SDF, Perl, Prolog, Python, Tcl).

A major test case our group is currently working on is the complete reengineering of the ASF+SDF Meta-Environment[Kli93]: given a formal description of some language (e.g., programming language, query language, mark up language, or other application language) the Meta-Environment generates an interactive programming environment for it. We use the ToolBus as the interconnection technology for connecting the various components involved (such as parser generators, syntax-directed editors, rewrite rule compilers, compiled specifications, and the like). By doing so, we will be able to assess the real benefits of this approach as stated in the introduction of this paper:

- By decomposing a single monolithic system into a number of cooperating components, the modularity and flexibility of the systems' implementation can be improved:
  - For each component we can select the optimal reengineering strategy (e.g., adapt the old component, re-implement it using the best tools available, buy it, etc.)
  - The components can be combined in new ways that were inconceivable in the old situation; rather than using a fixed skeleton for each programming environment generated, it becomes possible to tailor it towards specific needs.
- By connecting existing tools we can reuse their implementation and build new systems with lower costs. Typical examples we have encountered so far are the integration of existing tools for constructing user-interfaces and for constraint solving, as well as implementations of various programming languages.

5 Discussion

The ToolBus coordination architecture itself is a promising solution to the component interconnection problem, while the method used to arrive at its design has some merits of its own: the orchestrated use of process theory for design, algebraic specification for rapid prototyping, and C for experimental implementation, leads to a versatile framework for experimentation and implementation at affordable costs.

Case studies done so far, make clear that the expressive power of T scripts is acceptable: even sophisticated systems can be expressed concisely. We expect that the recently developed Discrete Time ToolBus is expressive enough to easily solve an even wider range of component interconnection problems. The simple approach we take compares favourably with other much more complex solutions like,

e.g., the *Object Request Broker Architecture* as proposed by the Object Management Group [ORB93] or IBM's *Common Blue Print* [IBM93].

Regarding performance, the major lesson we have learned is that a distinction should be made between a *logical design* of a system and its subsequent implementation. Currently, the structure of the logical design is closely followed at the implementation level. More sophisticated implementation techniques involving the full compilation of **T** scripts can probably eliminate all overhead due to interprocess communication and data conversion.

Acknowledgements

We like to thank Doeco Bosscher, Thomas Bullens, Steven Klusener, Wilco Koorn, and Pieter Olivier for providing us with valuable feedback on our first ToolBus design.

References

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

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5:23–70, 1993.

[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.

[BKK] D. Bosscher, A. S. Klusener, and J.W.C. Koorn. A simulator for process algebra specifications with the ToolBus. to appear.

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

[BM90] J. Banatre and D.L. 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.

[Bul94] T. Bullens. private communication, 1994.

[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.

[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.

[CP85] L. Cardelli and R. Pike. Squeak: a language for communication with mice. *Computer Graphics*, 19(3):199–204, 1985.

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

[Dis94] S. Dissoubray. Using Esterel for control integration. In *GIPE II: ESPRIT project 2177, Sixth review report.* january 1994.

[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.

[Gre86] M. Green. A survey of three dialogue models. *ACM Transactions on Graphics*, 5(3):244–275, 1986.

[HH89] H. R. Hartson and D. Hix. Human-computer interface development: concepts and systems for its management. *ACM Computing Surveys*, 21(1):5–92, 1989.

[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.

[Hil86] R. D. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction—the Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210, 1986.

[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.

[Koo92] J. W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. Technical Report P9222, Programming Research Group, University of Amsterdam, 1992.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-R93-30, CWI, 1993.

[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.

[Mye92] B.A. Myers, editor. *Languages for developing user interfaces*. Jones and Bartlett Publishers, 1992.

[Oli94] P. Olivier. SEAL versus the TOOLBUS. Master's thesis, Programming Research Group, University of Amsterdam, 1994.

[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.

[vdB88] J. van den Bos. Abstract interaction tools: a language for user interface management systems. *ACM Transactions on Programming Languages and Systems*, 14(2):215–247, 1988.

[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.