The ToolBus
—a component interconnection architecture—

J.A. Bergstra[1,2] 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 tool interactions in process-oriented "ToolBus 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.

We present the ToolBus architecture at five different levels of abstraction: (1) motivation and informal overview; (2) description of the intended meaning of ToolBus scripts using Process Algebra; (3) algebraic specification (in Asf+Sdf) of a prototype interpreter for ToolBus scripts; (4) examples of ToolBus scripts; (5) experimental C implementation of a ToolBus interpreter.

The results of this experiment can be viewed from two different angles. The ToolBus architecture itself seems to be one feasible approach to the component interconnection problem, while the method used to arrive at this 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.

1991 CR Categories: C.2.4 [**Computer-communication networks**]: Distributed systems; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Tools and techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors; D.4.1 [**Operating Systems**]: Process management; D.4.4 [**Operating Systems**]: Communications management; H.5.2 [**Information interfaces and presentation**]: User interfaces.
1991 Mathematics Subject Classification: 68N15 [*Software*]: Programming Languages; 68N20 [*Software*]: Compilers and generators; 68Q65 [*Theory of computing*]: Abstract data types; algebraic specification.
Key Words & Phrases: interoperability of software tools, modularity, control integration, client/server architectures, distributed systems, process interpreter, prototyping, software development methodology.
Note: The systematic use of the iteration operator has been inspired by simultaneous work [BBP94] in the context of the ESPRIT Basic Research Action CONFER no. 6454. It may serve as an illustration of the practical value of an iteration operator in Process Algebra.

Contents

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?

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

User-interface integration: how can the user-interfaces of the various tools be integrated in a uniform manner? [1]

We will now discuss these topics in more detail (Section 1.2) and then sketch our own approach (Section 1.3). A global outline of this paper is given in Section 1.4.

1.2 Current research in tool integration

1.2.1 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.

1.2.2 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.

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

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]. These 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 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] 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*[2] 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 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

[2] Small pieces of interfacing software.

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.

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).

1.2.3 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 recent overview of the role of concurrency in languages for developing user-interfaces. 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.

1.3 Our approach

1.3.1 Requirements and points of departure

Before starting a more detailed analysis of control integration, 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 "TOOLBUS script" should satisfy a number of requirements:

- It has a formal basis and can be formally analysed.
- 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.

1.3.2 The ToolBus

The ToolBus 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 "ToolBus 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.

The following list of features supported in ToolBus scripts is one way of characterizing our approach:

- Parallel, composition of a variable number of processes.
- Primitives for the sequential composition, choice and iteration of processes.
- A primitive inactive process.
- Private global variables per process.
- Handshaking (synchronous) communication between processes.
- Asynchronous distribution of notes to an arbitrary number of processes.
- Note subscription: only processes that have subscribed to certain notes will receive them.
- Dynamic process creation.
- Synchronous communication between process and tool.
- Data represented as terms on which only syntactic equality is defined; no other support for datatypes.

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.

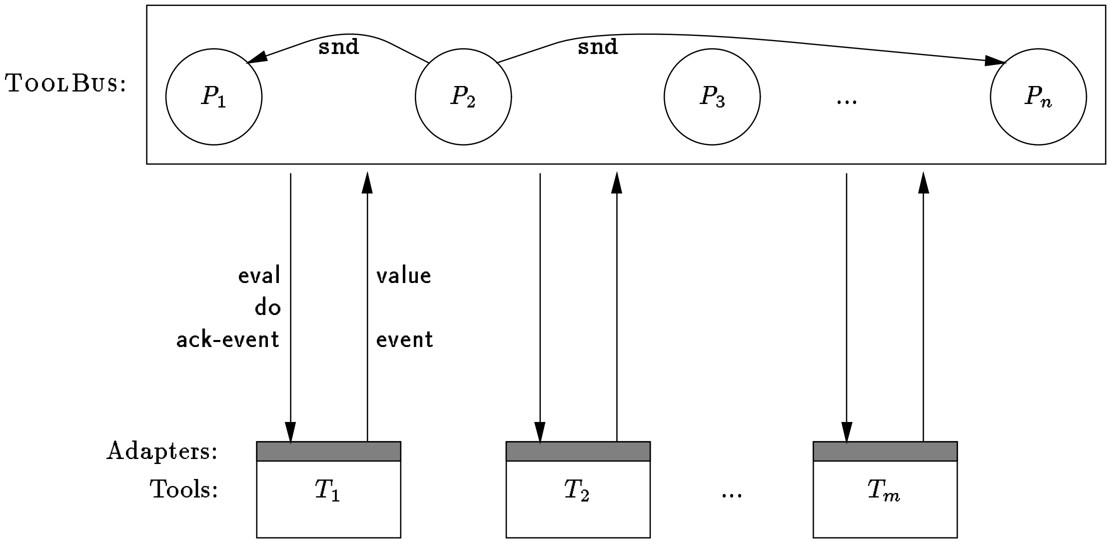

Figure 1: Global organization of the TOOLBUS

1.4 Aims and plan of this paper

We apply a number of established techniques (i.e., process algebra, algebraic specification, C implementation) to approach the design of the TOOLBUS at various levels of abstraction. This can give—and *has* given—rise to even mutual feedback between different levels.

In this paper we present the TOOLBUS approach at five different levels:

Informal introduction. An overview of the TOOLBUS architecture is given in Section 2 and the details of TOOLBUS scripts are explained in Section 3.

Process Algebra description. In Section 4 we describe the intended meaning of TOOLBUS scripts using operations from Process Algebra [BK84, BB88, Ber90, BBP94].

Operational Semantics. In Section 5 we present a formal specification of an *interpreter* for TOOLBUS scripts using ASF+SDF [BHK89a, HHKR89].

Examples. Four examples are presented in Sections 6, 7, 8, and 9, respectively.

Implementation in C. An experimental implementation in C of the prototype interpreter from Section 5 is given in Section 10.

Some concluding remarks are made in Section 11. There are four appendices. Appendix A gives defining equations for all operators from Process Algebra used in Section 4. Appendix B gives a detailed trace of events for the example from Section 6. Appendix C contains a complete listing of the TOOLBUS implementation in C. Appendix D contains a complete listing of the tool library.

2 Overview of the TOOLBUS 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 `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 ToolBus scripts

3.1 Overview

A "ToolBus script" describes the complete behaviour of a system. A script consists of the parallel composition of a number of process names, each defined by a process expression.

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*$_2$, ...) is a term, provided that *Term*$_1$, *Term*$_2$, ... are also terms.

One global name space is shared between all processes in the ToolBus. To avoid name conflicts, each name is implicitly prefixed with the name of the process in which it occurs.

The control integration problem is addressed by introducing the notions of atomic processes, process expressions and process definitions.

The following *atomic processes* are available:

- `snd-msg` and `rec-msg`: used for sending and receiving messages between two processes using synchronous communication. A `snd-msg` can communicate with exactly one `rec-msg` that matches the `snd-msg`'s argument list. A `rec-msg` will assign values to variables appearing in its argument list; these can be used later on in the process expression in which the `rec-msg` occurs.
- `subscribe` and `unsubscribe`: subscribe, respectively unsubscribe, to notes of a given form. A process will only receive notes it has subscribed to.

- `snd-note`, `rec-note`, and `no-note`: used for sending and receiving notes via asynchronous, selective, broadcasting. A `snd-note` is used to send to all (i.e., zero or more) processes that have subscribed to notes of that particular form. Each process maintains a queue of notes that have been received but have not yet been read. In this way, notes can never be lost. A `rec-note` will inspect the note queue of the current process, and if the queue contains a note of a given form, it will remove the note and assign values to variables appearing in its argument list; these can be used later on in the process expression in which the `rec-note` occurs. A `no-note` succeeds if the note queue does *not* contain a note of a given form.

- `rec-event`: receive an event from a tool. The first argument of `rec-event` serves as an identification of the source of the event. This identification is generated by the tool and consists of the tool name possibly extended with extra information. The remaining arguments give the details of the event in question.

- `snd-ack-event`: send an acknowledgement to a previous event received from a source. The assumption is made that the next event from that particular source will not be sent before the previous one has been acknowledged. Since one tool can generate events with different sources, a certain internal concurrency in tools can be supported.

- `snd-eval`: request a tool to evaluate a term. The first argument serves as the name of the tool, while the second one is the term to be evaluated.

- `rec-value`: receive from a tool the result of a previous evaluation request.

- `snd-do`: request a tool to evaluate a term and ignore the resulting value. The first argument serves as the name of the tool, while the second one is the term to be evaluated.

- `create`: dynamically create a new process, given the name of its process definition and actual parameter list (which may be empty). Formal parameters are textually replaced by corresponding actual values and thus act as constants in the resulting process expression.

- `delta`: the atomic process corresponding to dead lock, mainly used for representing process termination.

Composite process expressions may have the following form:

- One of the atomic processes mentioned above.

- P_1 + P_2: a *choice* between process P_1 and process P_2.

- P_1 . P_2: the *sequential composition* of process P_1 and process P_2, i.e., P_1 followed by P_2.

- P_1 * P_2: zero or more *repetitions* of process P_1 followed by process P_2 (binary Kleene star).

- A *named process*: will be replaced by its definition. A list of actual parameters may follow the process name; in that case, the formal parameter names are first systematically replaced by their corresponding actual values before the replacement is made. Recursive definitions are not allowed.

A *process definition* can define a process as follows:

`define` *Pname Formals* = *P*

Formals are optional and contain a list of formal parameter names. They are only relevant for processes that are either used directly in the ToolBus configuration (see below) or are created dynamically.

A ToolBus *configuration* is an encapsulated parallel composition of named processes. It has the form:

`toolbus(`$Pname_1$`, ..., `$Pname_n$`)`

A complete ToolBus *script* consists of a list of process definitions followed by a single ToolBus configuration.

3.2 The syntax of TOOLBUS scripts

The precise syntax of TOOLBUS scripts will now be presented in two steps.[3] First, we introduce the notion TERM, which represents term structures to be used as arguments and results of the various atomic processes.

3.2.1 The syntax of terms

Module Terms
exports
  sorts ID VAR GLOBAL INT STRING TERM TERM-LIST
  lexical syntax

```
    [␣\t\n]                → LAYOUT
    "%%"~[\n]*             → LAYOUT
    [a-z][A-Za-z0-9\-]*    → ID
    [0-9]+                 → INT
    "\""~[\"]*"\""         → STRING
    [A-Z][A-Za-z0-9]*      → VAR
```

  context-free syntax

```
    VAR "$" PNAME              → GLOBAL
    INT                        → TERM
    STRING                     → TERM
    VAR                        → TERM
    GLOBAL                     → TERM
    ID                         → TERM
    ID "(" TERM-LIST ")"       → TERM
    {TERM ","}+                → TERM-LIST
```

  variables

```
    T [0-9']*        → TERM
    Ts [0-9']*       → {TERM ","}+
    OptTs [0-9']*    → {TERM ","}*
    Tlist [0-9']*    → TERM-LIST
    Var [0-9']*      → VAR
    Vars [0-9']*     → {VAR ","}*
    Id [0-9']*       → ID
    Ids [0-9']*      → {ID ","}*
    Int              → INT
    String           → STRING
```

3.2.2 The syntax of scripts

Next, we define the syntax of atoms (ATOM) and processes (PROC), process definitions (PROC-DEF), and TOOLBUS configurations (TB-CONFIG), and TOOLBUS scripts (TB-SCRIPT).

Module TBscript
imports Terms(3.2.1)
exports
  sorts ATOM ATOMIC-FUN PROC PNAME PROC-DEF FORMALS
    TB-CONFIG TB-SCRIPT
  lexical syntax

```
    [A-Z][A-Za-z0-9\-]* → PNAME
```

  context-free syntax

[3] We postpone a description of the formalism used here until Section 5.1.

```
  snd-msg                              → ATOMIC-FUN
  rec-msg                              → ATOMIC-FUN
  snd-note                             → ATOMIC-FUN
  rec-note                             → ATOMIC-FUN
  no-note                              → ATOMIC-FUN
  rec-event                            → ATOMIC-FUN
  snd-ack-event                        → ATOMIC-FUN
  snd-eval                             → ATOMIC-FUN
  snd-do                               → ATOMIC-FUN
  rec-value                            → ATOMIC-FUN
  subscribe                            → ATOMIC-FUN
  unsubscribe                          → ATOMIC-FUN

  ATOMIC-FUN "(" TERM-LIST ")"         → ATOM
  create PNAME                         → ATOM
  create PNAME "(" TERM-LIST ")"       → ATOM

  ATOM                                 → PROC
  delta                                → PROC
  PROC "+" PROC                        → PROC         {left}
  PROC "." PROC                        → PROC         {right}
  PROC "*" PROC                        → PROC         {left}
  "(" PROC ")"                         → PROC         {bracket}
  PNAME                                → PROC
  PNAME "(" TERM-LIST ")"              → PROC

  "(" {VAR ","}* ")"                   → FORMALS
                                       → FORMALS

  define PNAME FORMALS "=" PROC        → PROC-DEF
  toolbus({PNAME ","}+)                → TB-CONFIG
  PROC-DEF+ TB-CONFIG                  → TB-SCRIPT
priorities
  PROC "*"PROC → PROC > "." > PROC "+"PROC → PROC
variables
  Pnm [0-9']*  → PNAME
  P [0-9']*    → PROC
  Ps [0-9']*   → {PROC ","}+
  OPs [0-9']*  → {PROC ","}*
  Pnms [0-9']* → {PNAME ","}*
```

3.3 Constraints on TOOLBUS scripts

A TOOLBUS script should satisfy the following static constraints:

- All names of defined processes should be different.
- All process names appearing in a TOOLBUS configuration or in any process definition should have been defined in some process definition.
- The number of actual parameters following a process name, should be equal to the number of formal parameters in the corresponding process definition.
- Recursive use of named processes is not allowed.

Checking these constraints is straightforward. In the sequel, we will only consider TOOLBUS scripts that satisfy all these constraints.

3.4 Events and the ToolBus protocol

The syntactic form of the "events" used in the ToolBus is defined as follows:

Module Events
imports TBscript$^{(3.2.2)}$ Booleans$^{(5.1)}$
exports
sorts EVENT EVENT-LIST
context-free syntax

event(TERM-LIST) → EVENT
value(TERM-LIST) → EVENT
eval(TERM-LIST) → EVENT
do(TERM-LIST) → EVENT
ack-event(TERM-LIST) → EVENT
note(TERM-LIST) → EVENT
"[" {EVENT ","}* "]" → EVENT-LIST
EVENT "∈" EVENT-LIST → BOOL

variables

Event → EVENT
Events [0-9′]*→ {EVENT ","}*

equations

$$Event \in [Events_1, Event, Events_2] = \text{true} \qquad \text{[in-1]}$$

$$Event \in [Events] = \text{false} \quad \textbf{otherwise} \qquad \text{[in]}$$

The events event and value come from a tool T_i and communicate with one process P_j, while the events eval, do and ack-event come from a process P_i and go to some tool T_j.

The first argument of all events acts as an addressing mechanism. For event and value it gives a precise description of (a part of) the sending tool. For eval, do and ack-event it gives a precise description of the intended destination tool.

We permit arbitrary terms as address. In this way a tool need not have a single address, e.g., `ui`, but an address may have more structure, e.g. `ui(buttons)`. In this way, tools can be subdivided into logically related parts. This is particularly relevant when the steps in one part have to be strictly sequential, while steps in different parts of the same tool may proceed in parallel.

We will use a fixed protocol between tools and ToolBus which can informally be summarized as follows (S stands for the source, and Ts for a list of terms):

((event(S,Ts_1) . (eval(S,Ts_2) . value(S,Ts_3) + do(S, Ts_4)) * ack-event(S))
+ (eval(S,Ts_1) . value(S,Ts_2))
+ do(S,Ts_1)
) * delta

In other words, a tool has the following three possible behaviours:

- The tool sends an "event" to the ToolBus, then receives zero or more "eval" or "do" messages. The "eval" message is always answered by the tool with a "value" message. Finally, the tool receives an "ack-event" message from the ToolBus which concludes the interactions for this event. During all these interactions the tool cannot generate a new event with source S.
- The tool receives an "eval" message and answers it with a "value" message.
- The tool receives a "do" message.

3.5 A first example: compiler/editor cooperation

Our first motivating example illustrates how an independent compiler and editor can be made to cooperate. The behaviour we want to achieve is as follows. First the user starts the compilation of a file. Next, when there are compilation errors, we want the text editor to display the error location. When the user presses a "next-error" button in the editor, the compiler proceeds until the next error or the end of compilation. We define three processes:

- `COMPILER` receives a compilation request (in this case from `UI`, see below), starts the actual compilation, broadcasts notes describing compilation errors (if any) and sends the result of the compilation back.
- `EDITOR` either receives a note describing a compilation error and stores the file and error location for later visits, or receives a next-error event from the editor and visits a previously stored error location. Observe that the `EDITOR` process explicitly `subscribes` to `compile-error` notes.
- `UI` represents a user-interface containing a "compile" button. When this button is pushed, a compile message is send and the reply is awaited.

The process definitions are:

```
define COMPILER =
    ( rec-msg(compile, Name) . snd-eval(compiler, Name) .
      ( rec-value(compiler, error(Err), loc(Loc)) .
          snd-note(compile-error, Name, error(Err), loc(Loc))
      ) * rec-value(compiler, Name, Res) . snd-msg(compile, Name, Res)
    ) * delta

define EDITOR  =
    subscribe(compile-error) .
    ( rec-note(compile-error, Name, error(Err), loc(Loc)) .
        snd-do(editor, store-error(Name, error(Err), loc(Loc)))
    + rec-event(editor, next-error(Name)) .
        snd-do(editor, visit-next(Name)) . snd-ack-event(editor)
    ) * delta

define UI =
    ( rec-event(ui, button(compile, Name)) .
        snd-msg(compile, Name) . rec-msg(compile, Name, Res) . snd-ack-event(ui)
    ) * delta
```

Using these definitions, we can describe two systems:

- `toolbus(COMPILER, EDITOR, UI)` defines the system described above.
- `toolbus(COMPILER, UI)` defines a simpler system without the facility to visit error locations.

4 ToolBus scripts in Process Algebra

4.1 A Quick overview of Process Algebra (ACP)

Process Algebra (ACP) is an algebraic approach to the description of parallel, communicating, processes originally proposed in [BK84]. It consists of a considerable collection of axiomatically defined operators capturing the meaning of many features one usually encounters in concurrent systems. In addition, Process Algebra provides a meta-theory allowing for the verification of process descriptions.

Although we do not want to give an extensive description of Process Algebra here, we do give a brief summary of the subset of the theory we are using, in order to make this paper self contained. We will present the operators of BPA_δ, PA and ACP as well as operators for renaming, iteration, process creation, and state manipulation. A summary of these axiom systems and operator definitions is given in Appendix A. We refer to [BW90] for a more elaborate description.

Basic Process Algebra (BPA). The theory BPA is build up from the following ingredients:

- a, b, c, ...: atomic actions (from some alphabet A).
- x, y, z, ...: processes.
- $x.y$: first execute x, then y (sequential composition).
- $x + y$: either execute x or y, but not both (choice, alternative composition).

One should observe that the alphabet A of atomic actions forms a parameter of the whole theory. When instantiating the theory—as we will do in Section 4.2—the atomic actions can become arbitrarily complex and may represent, for instance, fragments of imperative programs.

The theory can then be extended with the notion:

- δ: deadlock (inaction),

yielding the axiom system BPA_δ.

Process Algebra (PA). Next, we introduce the notion of *parallel composition* of processes:

- $x \parallel y$: execute x and y in parallel (merge).
- $x \mathbin{|\!\lfloor} y$: same as merge, but first step must come from x (left merge)

This leads to the axioms PA. It is important to observe that the *complete behaviour* of a process is characterized by PA. Using PA, one can, for instance, prove the following equalities:

- $a \parallel b = a.b + b.a$
- $(a.b) \parallel c = a.(b.c + c.b) + c.a.b$

And here we see, that a process expression captures all possible executions of the process (as opposed to one particular execution thread).

An important result of the theory is that the operators $\parallel$ and $\mathbin{|\!\lfloor}$ can always be eliminated. In other words, the behaviour of a process can be expressed only using the operators . and $+$.

Encapsulation and renaming. When defining larger systems, it is necessary to have some means to subdive them in smaller subsystems. In particular, the occurrences of certain atomic processes may have to be restricted to a subsystem. This is achieved by the operator ∂_H which *encapsulates* a process, i.e., renames all atoms in $H \subset A$ into δ.

Encapsulation is a special case of *renaming* atomic actions as introduced in [Vaa90]. Let f be a function from the set A of atomic actions to the set $A \cup \{\delta\}$. Then one can define the unary renaming operator ρ_f, which replaces every occurrence of a constant $a \in A$ by $f(a)$.

Algebra of Communicating Processes (ACP). The next step in the development of the theory is the introduction of the notion of communication:

- $\gamma(a, b) = c$: communication function (partial) on atoms.
- $x \mid y$: execute x and y in parallel, first step must be communication between x and y.

This leads to the Algebra of Communicating Processes [BK84]. Observe that the way communication is defined is, again, very general: the result of communication is an atom which, in its turn, could be engaged in other communications. We will make essential use of this flexibility in section 4.2.

Note that the operators $\parallel$, $\mathbin{|\!\lfloor}$, $|$, ∂_H can always be eliminated from finite process expressions.

The state operator. The next ingredient we will need is the description of the notion of a *global state* [BB88]. In particular, we want to describe the effect of an atomic action a on a global state S. This leads to equations of the form

$$\lambda_S(a.x) = a'.\lambda_{S'}(x)$$

where a' appears as the result of executing action a in state S (the *action*, denoted by $a(S)$) and S' is the state resulting from the execution of a in state S (the *effect*, denoted by $S(a)$).

Iteration. Next, we introduce iteration [BBP94] by means of the binary Kleene's star operation: $x * y$ is the process that chooses between x and y, and upon termination has this choice again.

Process creation. Finally, we introduce the process creation operator E_ϕ, described in [Ber90], which allows the use of atoms of the form create(x) to generate process instances $\phi(x)$ as follows

$$E\phi(\text{create}(x)\ .\ P) = \overline{\text{create}}(x)\ .\ E_\phi(\phi(x) \parallel P)$$

4.2 Semantics of TOOLBUS scripts

Preliminaries. Let $\vec{S}$ denote a vector of terms $S_1, ..., S_k$. The *length* $|\vec{S}|$ of a vector equals the number of elements in it, e.g., $|\vec{S}| = k$. A vector $\vec{R}$ *matches* a vector $\vec{S}$ resulting in a substitution σ if the following conditions hold:

1. $|\vec{R}| = |\vec{S}|$.
2. For each element R_i in $\vec{R}$ and corresponding element S_i in $\vec{S}$ we have $\sigma(R_i) = S_i$ (notation: $\sigma(\vec{R}) = \vec{S}$).

If such a substitution σ does not exist, we say that $\vec{R}$ *does not match* vector $\vec{S}$.

We can use this notion of term matching, since all data terms appearing in our process expressions are *purely syntactic* in nature, i.e., they are generated by a free term algebra and are thus not constrained by any equations.

ToolBus semantics. We define the meaning of toolbus$(P_1, ..., P_n)$ by:

$$\lambda_G \circ \partial_H \circ E_\phi(\rho_{tag_1}(P_1) \ \parallel\ ...\ \parallel\ \rho_{tag_n}(P_n))$$

In the following paragraphs, we will further define and explain the constituents of this definition.

Renamings and encapsulation. Let $tag_i : A \to A$ be a collection of functions for tagging atoms with the label i. Then ρ_{tag_i} renames the actions in a process expression in such a way that, whenever needed, the tag i is attached to appropriate components of the action, like, e.g.,

$$\rho_{tag_i}(\mathsf{rec\text{-}msg}(\mathsf{calc}, E)\ .\ P) = \mathsf{rec\text{-}msg}(\mathsf{calc},\ E_i)\ .\ \rho_{tag_i}(P).$$

In this way we can ensure that the same name, appearing in different processes, is correctly disambiguated. In addition, ρ_{tag_i} renames all rec-note, subscribe, and unsubscribe atoms as follows:

$$\rho_{tag_i}(\mathsf{rec\text{-}note}(...)) = \mathsf{rec\text{-}note}_i(...)$$
$$\rho_{tag_i}(\mathsf{no\text{-}note}(...)) = \mathsf{no\text{-}note}_i(...)$$
$$\rho_{tag_i}(\mathsf{subscribe}(...)) = \mathsf{subscribe}_i(...)$$
$$\rho_{tag_i}(\mathsf{unsubscribe}(...)) = \mathsf{unsubscribe}_i(...)$$

All other atoms, i.e., snd-msg, rec-msg, snd-note, rec-event, rec-value, snd-eval, snd-do, snd-ack-event, and create, remain unaffected. As a result, rec-note, subscribe, and unsubscribe will have effect in the appropriate context.

Finally, we define ∂_H with $H = \{\mathsf{snd\text{-}msg}, \mathsf{rec\text{-}msg}\}$, such that all communication atoms are encapsulated.

Communication among atoms is defined by:

$$\mathsf{snd\text{-}msg}(\vec{S}) \mid \mathsf{rec\text{-}msg}(\vec{R}) = \mathsf{comm\text{-}msg}(\vec{S}; \vec{R})$$

The global state $\mathcal{G}$ contains the following entities:

- a single global name space;
- a counter $c \in \mathbf{N}$ used for process creation, initially $c = n$;
- for each process i a *subscription list* $\mathcal{S}_i$, i.e., a multi-set of names of notes to which process i has subscribed;
- for each process i a *note queue* $\mathcal{Q}_i$, i.e., a queue of notes received but not yet read by process i.

On *subscription lists*, we define the operations:

- $\mathcal{G}' = \text{subscribe}(N, i, \mathcal{G})$ for adding the name N to i's subscription list $\mathcal{S}_i$;
- $\mathcal{G}' = \text{unsubscribe}(N, i, \mathcal{G})$ for deleting the name N from i's subscription list $\mathcal{S}_i$;

where addition and deletion are *multi-set* operations, i.e., they increase or decrease the occurrence count of name N in the multi-set. Both operations yield a new state $\mathcal{G}'$.

For *note queues*, several operations exist. A note $N(\vec{S})$ is placed in note queue $\mathcal{Q}_i$ by $\mathcal{G}' = \text{insert}(N(\vec{S}), i, \mathcal{G})$. The operation $\text{dist}(N(\vec{S}), \mathcal{G})$ distributes a note over all relevant note queues and is defined as follows:

1. Replace all variables appearing in $\vec{S}$ by their value in $\mathcal{G}$, yielding $\vec{S}'$.
2. Define $\mathcal{G}_0 = \mathcal{G}$.
3. For $i = 1, ..., c$ do the following:
   (a) If $N \in \mathcal{S}_i$ define $\mathcal{G}_i = \text{insert}(N(\vec{S}'), i, \mathcal{G}_{i-1})$.
   (b) Otherwise, define $\mathcal{G}_i = \mathcal{G}_{i-1}$.
4. Define $\text{dist}(N(\vec{S}), \mathcal{G}) = \mathcal{G}_c$.

Notes can be retrieved from the note queue by $\text{has-note}(N(\vec{R}), i, \mathcal{G})$ yielding:

- *none*, if $\mathcal{Q}_i$ *does not* contain a note matching $N(\vec{R})$.
- $\mathcal{G}'$, if $\mathcal{Q}_i$ *does* contain a matching note $N(\vec{R})$. In that case there exists a substitution $\sigma(\vec{R}) = \vec{S}$ and $\mathcal{G}'$ is the state obtained from $\mathcal{G}$ by removing the matching note from $\mathcal{Q}_i$ *and* by replacing the values of all variables in $\mathcal{G}$ that are in the domain of σ by their respective values.

Finally, we define the operation $\text{create-process}(\mathcal{G})$ yielding a new global state in which:

- the counter c has been incremented;
- a new, empty, subscription list $\mathcal{S}_c$ has been created;
- a new, empty, note queue $\mathcal{Q}_c$ has been created.

The state operator $\lambda_{\mathcal{G}}$. We can now enumerate all the cases defining the state operator $\lambda_{\mathcal{G}}$. For alternative composition ($+$), we define:

$$\lambda_G(P_1 + P_2) = \lambda_G(P_1) + \lambda_G(P_2)$$

All the cases for sequential composition (.) are described below.

Communicating a *message* may have two possible effects: if the arguments of sender and receiver do not match, the effect is deadlock. If, however, they do match, the effect of the communication is a modification of the values of the variables appearing in the receiver:

$$\lambda_{\mathcal{G}}(\textsf{comm-msg}(\vec{S}; \vec{R}) \ . \ P) = \begin{cases} \delta & \textbf{if } \forall\sigma : \sigma(\vec{R}) \neq \vec{S} \\ t \ . \ \lambda_{\sigma(\mathcal{G})}(P) & \textbf{where } \sigma(\vec{R}) = \vec{S} \end{cases}$$

In the above definition, and in the ones following, t represents an arbitrary internal step without any possible communication behaviour (pre-abstraction in the sense of [BB88]).

Sending a note amounts to distributing the note to all subscribing processes:

$$\lambda_{\mathcal{G}}(\textsf{snd-note}(N(\vec{S})) \ . \ P) = t \ . \ \lambda_{\mathrm{dist}(N(\vec{S}),\mathcal{G})}(P)$$

Receiving a note may have two possible effects: if the argument of rec-note does not match some note in the queue, the effect is deadlock. Otherwise, the values of the variables appearing in the rec-note are updated and the note is removed from the queue:

$$\lambda_{\mathcal{G}}(\textsf{rec-note}_i(N(\vec{R})) \ . \ P) = \begin{cases} \delta & \textbf{if } \mathrm{has\text{-}note}(N(\vec{R}), i, \mathcal{G}) = \mathit{none} \\ t \ . \ \lambda_{\mathcal{G}'}(P) & \textbf{if } \mathrm{has\text{-}note}(N(\vec{R}), i, \mathcal{G}) = \mathcal{G}' \end{cases}$$

Determining the absence of a note may have two possible effects: if the arguments of no-note match the arguments of some note in the queue, the effect is deadlock. Otherwise, no-note is replaced by t:

$$\lambda_{\mathcal{G}}(\textsf{no-note}_i(N(\vec{R})) \ . \ P) = \begin{cases} \delta & \textbf{if } \mathrm{has\text{-}note}(N(\vec{R}), i, \mathcal{G}) = \mathcal{G}' \\ t \ . \ \lambda_{\mathcal{G}}(P) & \textbf{if } \mathrm{has\text{-}note}(N(\vec{R}), i, \mathcal{G}) = \mathit{none} \end{cases}$$

Subscribing to notes has the effect of *adding* a name to the subscription list of the process in which the atom subscribe appears:

$$\lambda_{\mathcal{G}}(\textsf{subscribe}_i(N) \ . \ P) = t \ . \ \lambda_{\mathrm{subscribe}(N,i,\mathcal{G})}(P)$$

Unsubscribing to notes has the effect of *deleting* a name from the subscription list of the process in which the atom unsubscribe appears:

$$\lambda_{\mathcal{G}}(\textsf{unsubscribe}_i(N) \ . \ P) = t \ . \ \lambda_{\mathrm{unsubscribe}(N,i,\mathcal{G})}(P)$$

Receiving an event is an unrestricted input operation which we model by constructing the sum of all possible values that may be assigned to the variables appearing in the argument $\vec{R}$. For simplicity, we assume that the rec-event has only one single variable as argument. (This avoids the additional complexity that variables occur at nested positions in complex terms appearing as arguments.)

$$\lambda_{\mathcal{G}}(\textsf{rec-event}(R) \ . \ P) = \textstyle\sum_{r \in Dom(R)} \textsf{rec-event}(r) \ . \ \lambda_{\mathcal{G}[r/R]}(P)$$

Receiving a value follows the same pattern as rec-event above:

$$\lambda_{\mathcal{G}}(\textsf{rec-value}(R) \ . \ P) = \textstyle\sum_{r \in Dom(R)} \textsf{rec-value}(r) \ . \ \lambda_{\mathcal{G}[r/R]}(P)$$

The next three cases, snd-eval, snd-do, and ack-event are completely straightforward:

$$\lambda_{\mathcal{G}}(\textsf{snd-eval}(\vec{S}) \ . \ P) = \textsf{snd-eval}(\vec{S}) \ . \ \lambda_{\mathcal{G}}(P)$$

$$\lambda_{\mathcal{G}}(\textsf{snd-do}(\vec{S})\ .\ P) = \textsf{snd-do}(\vec{S})\ .\ \lambda_{\mathcal{G}}(P)$$

$$\lambda_{\mathcal{G}}(\textsf{snd-ack-event}(\vec{S})\ .\ P) = \textsf{snd-ack-event}(\vec{S})\ .\ \lambda_{\mathcal{G}}(P)$$

Process creation takes the form of an atomic action $\textsf{create}(X,\ \vec{T})$, where X is a name and $\vec{T}$ is a list of argument terms. Its meaning is defined by the process creation operator E_ϕ, where we define ϕ by:

$$\phi(X, \vec{T}) = \sum_{k \in \mathbf{N}} (k = c) :\rightarrow \rho_{tag_k}(X(\vec{T}))$$

A unique new index is associated with each created process (using create-process (see below). The process expression for the new process is obtained via the name X and instantiated with actual parameters $\vec{T}$. The resulting expression is renamed using the new index c (to avoid any interference with the expressions of other processes), prefixed with a conditional and embedded in a sum over all possible process indices.

Observe that the process creation operator E_ϕ leaves a "trace" of the create atom of the form $\overline{\textsf{create}}$. The definition of the state operator for this case is now as follows:

$$\lambda_{\mathcal{G}}(\overline{\textsf{create}}(X,\vec{T})\ .\ P) = \overline{\textsf{create}}(X,\vec{T})\ .\ \lambda_{\text{create-process}(\mathcal{G})}(P)$$

This completes the definition of the state operator $\lambda_{\mathcal{G}}$.

5 A prototype ToolBus interpreter specified in Asf+Sdf

Our next, more concrete, level of description of the ToolBus concerns the operational behaviour of ToolBus scripts. The Process Algebra semantics given in the previous section, 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 ToolBus scripts that includes scheduling rules for selecting execution paths.

Our overall strategy is as follows. At any moment during interpretation, the state of each process can be expressed as $<P_1,\ \ldots,\ P_n>:E:S:N$. Each P_i represents a possible choice in the process and does not itself contain any +-operators. E is the local environment of the process containing all variables and their values, S is the list of notes the process has subscribed to, and N is a queue of notes awaiting consumption by the process.

The complete state of the interpreter can then be expressed as a collection of parallel processes:

$$\{<P_{11},\ldots,P_{1n_1}>:E_1:S_1:N_1,\ldots,\ <P_{m1},\ldots,P_{mn_m}>:E_m:S_m:N_m\}.$$

One interpretation step consists of selecting one alternative P_{ij} in each process—according to certain fixed scheduling rules defined by the interpreter—and computing a new interpreter state as well as zero or more external communications with tools.

The interpreter as a whole thus captures the input/output behaviour of the system described by the script: given a ToolBus script and a list of input events (modeling the events coming from the tools connected to the ToolBus), it computes a list of responses (modeling messages to the connected tools).

We should—once more—emphasize that interpretation implies scheduling, which excludes certain possible execution paths. However, the path selected is among all possible paths as defined by the semantics given earlier in Section 4.

The presentation consists of five parts. First, we give in Section 5.1 a quick overview of the specification language Asf+Sdf we will use. Next, as a preparation for the definition of the ToolBus interpreter, we need several auxiliary notions:

- *Environments*: for representing the value of the variables occurring in process expressions (Section 5.2).
- *Matching and substitution*: for determining the match (or mismatch) between terms and lists of terms, and for replacing, in terms, variables by their value (Section 5.3).
- ToolBus *representation*: the internal representation of the ToolBus during interpretation (Section 5.4).

Finally, we describe in Section 5.5 the interpreter proper.

5.1 A quick overview of Asf+Sdf

Asf+Sdf is a specification formalism for describing all syntactic and semantic aspects of (formal) languages. It is an amalgamation of the formalisms Sdf [HHKR89, HK89b] for describing syntax, and Asf [BHK89b] for describing semantics.

Asf is a conventional algebraic specification formalism providing notions like first-order signatures, import/export, variables, and conditional equations. The meaning of Asf specifications is based on their initial algebra semantics. If specifications satisfy certain criteria, they can be executed as term rewriting system.

Sdf introduces the idea of a "syntactic front-end" for terms and equations defined over a first-order signature. This creates the possibility to write first-order terms as well as equations in arbitrary concrete syntactic forms: from a given Sdf definition for some context-free grammar, a fixed mapping from strings to terms can be derived. Sdf specifications can be executed using general scanner and parser generation techniques [HKR90, HKR92].

An Asf+Sdf specification consists of a sequence of named modules. Each module may contain:

Imports of other modules.

Sort declarations defining the sorts of a signature.

Lexical syntax defining layout conventions and lexical tokens.

Context-free syntax defining the concrete syntactic forms of the functions in the signature.

Variables to be used in equations. In general, each variable declarations has the form of a regular expression and defines the class of all variables whose name is described by the regular expression.

Equations define the meaning of the functions defined in the contex-free syntax.

Two features, *lists* and *default equations* will be used extensively in this paper, and deserve some further explanation. In the description of context-free grammars one frequently encounters the notion of iteration or list, in order to describe syntactic constructs like statement-list, parameter-list, declaration-list, etc. In Asf+Sdf this notion is provided at the syntactic as well as at the semantic level. At the syntactic level, one can define, for instance, a "list of zero or more identifiers separated by comma's". At the semantic level, variables over such lists may be declared and used in equations. Semantically, lists can always be eliminated. Operationally, the matching of a list structure is achieved by local backtracking during term rewriting [Hen89].

As already pointed out in [HK89a], significant abbreviations of algebraic specifications are possible by permitting negative conditions in equations. In Asf+Sdf we go one step further and also provide *default equations* intended for defining in a single equation all "the remaining cases for defining a certain function". This is typically advantageous when defining equality-like functions, where all true cases are defined by separate equations and all false cases can be captured by a single default equation. Semantically, default equations can always be eliminated provided that the specification is sufficiently complete. Operationally, they can be implemented using priority rewrite rules [BBKW89].

We have already shown fragments of Asf+Sdf specifications in previous sections. In Sections 3.2.1 and 3.2.2 we have seen the syntax of terms, respectively, ToolBus scripts. By interpreting sorts as non-terminals and reading the SDF rules from right to left, the similarity with standard BNF grammars becomes apparent. Both definitions are pure syntax definitions without equations.

In Section 3.4, we have seen a syntax definition *with* equations. The function $\in$ is defined by two equations: equation (in-1) uses list matching to locate in a list the Event we are looking for, while equation (default-in) covers the case that Event does *not* occur in the list. In all specifications in this paper, we will assume the existence of modules Booleans and Integers.

Support for writing ASF+SDF specifications is given in the ASF+SDF Meta-environment described in [Kli93].

5.2 Environments

Environments are needed for representing the values of the variables occurring in process expressions. They are also used for representing variable bindings during the matching of terms.

Module Environments
imports TBscript$^{(3.2.2)}$
exports
  sorts ENV ENTRY
  context-free syntax
    "(" GLOBAL ":" TERM ")" → ENTRY
    "[" {ENTRY ","}* "]" → ENV
    assign(GLOBAL, TERM, ENV) → ENV
    undefined → TERM
    value(GLOBAL, ENV) → TERM
    update(ENV, ENV) → ENV
    append(ENV, ENV) → ENV
    assign-list "(" {VAR ","}* "," PNAME "," {TERM ","}* "," ENV ")" → ENV
    create-env "(" {VAR ","}* "," PNAME "," {TERM ","}* ")" → ENV
  variables
    Entry [0-9']* → ENTRY
    Entries [0-9']* → {ENTRY ","}*
    E [0-9']* → ENV
    Global [0-9']* → GLOBAL
equations

Variables are global and they are made unique inside each process expression, by suffixing them with a process name, e.g., variable V in process P is represented by V \$ P (see Section 5.4). Environments (ENV) consist of zero or more (GLOBAL, TERM) tuples, on which the operations assign and value are defined.

$$\mathrm{assign}(\mathit{Global},\ T,\ [(\mathit{Global} : T'),\ \mathit{Entries}]) = [(\mathit{Global} : T),\ \mathit{Entries}] \qquad \text{[assign-1]}$$

$$\frac{\mathit{Global} \neq \mathit{Global}',\quad \mathrm{assign}(\mathit{Global},\ T,\ [\mathit{Entries}]) = [\mathit{Entries}']}{\mathrm{assign}(\mathit{Global},\ T,\ [(\mathit{Global}' : T'),\ \mathit{Entries}]) = [(\mathit{Global}' : T'),\ \mathit{Entries}']} \qquad \text{[assign-2]}$$

$$\mathrm{assign}(\mathit{Global},\ T,\ []) = [(\mathit{Global} : T)] \qquad \text{[assign-3]}$$

Observe that assignment to a variable that does not occur in the environment leads to the extension of the environment with a new tuple. In a similar way, we define the value function.

$$\mathrm{value}(\mathit{Global},\ [(\mathit{Global} : T),\ \mathit{Entries}]) = T \qquad \text{[value-1]}$$

$$\frac{\mathit{Global} \neq \mathit{Global}'}{\mathrm{value}(\mathit{Global},\ [(\mathit{Global}' : T'),\ \mathit{Entries}]) = \mathrm{value}(\mathit{Global},\ [\mathit{Entries}])} \qquad \text{[value-2]}$$

$$\mathrm{value}(\mathit{Global},\ []) = \mathrm{undefined} \qquad \text{[value-3]}$$

Given two environments, we define an update function that updates the values of the variables in the second environment with those in the first one.

$$\frac{\mathsf{assign}(\mathit{Global},\ T,\ [\mathit{Entries}']) = [\mathit{Entries}'']}{\mathsf{update}([(\mathit{Global} : T),\ \mathit{Entries}],\ [\mathit{Entries}']) = \mathsf{update}([\mathit{Entries}],\ [\mathit{Entries}''])} \quad \textsf{[update-1]}$$

$$\mathsf{update}([],\ [\mathit{Entries}]) = [\mathit{Entries}] \quad \textsf{[update-2]}$$

Next, we define an append operation on environments which is primarily used for the concatenation of variable bindings during matching (see Section 5.3).

$$\mathsf{append}([\mathit{Entries}_1],\ [\mathit{Entry},\ \mathit{Entries}_2]) = \mathsf{append}([\mathit{Entries}_1,\ \mathit{Entry}],\ [\mathit{Entries}_2]) \quad \textsf{[append-1]}$$

$$\mathsf{append}([\mathit{Entries}],\ []) = [\mathit{Entries}] \quad \textsf{[append-2]}$$

Finally, we define the functions assign-list and create-env that are intended for the construction of a new environment during process creation.

$$\frac{\mathsf{assign}(\mathit{Var}\ \$\ \mathit{Pnm},\ T,\ E) = E'}{\mathsf{assign\text{-}list}(\mathit{Var},\ \mathit{Vars},\ \mathit{Pnm},\ T,\ \mathit{Ts},\ E) = \mathsf{assign\text{-}list}(\mathit{Vars},\ \mathit{Pnm},\ \mathit{Ts},\ E')} \quad \textsf{[assign-list-1]}$$

$$\mathsf{assign\text{-}list}(\mathit{Var},\ \mathit{Pnm},\ T,\ E) = \mathsf{assign}(\mathit{Var}\ \$\ \mathit{Pnm},\ T,\ E) \quad \textsf{[assign-list-2]}$$

$$\mathsf{assign\text{-}list}(\mathit{Vars},\ \mathit{Pnm},\ \mathit{Ts},\ E) = [] \qquad \textbf{otherwise} \quad \textsf{[assign-list]}$$

$$\mathsf{create\text{-}env}(\mathit{Vars},\ \mathit{Pnm},\ \mathit{Ts}) = \mathsf{assign\text{-}list}(\mathit{Vars},\ \mathit{Pnm},\ \mathit{Ts},\ []) \quad \textsf{[create-env-1]}$$

5.3 Matching and substitution

Next, we define matching and substitution of terms.

Module Match

imports TBscript$^{(3.2.2)}$ Environments$^{(5.2)}$
exports
context-free syntax

substitute(TERM, ENV)	→ TERM
substitute-list(TERM-LIST, ENV)	→ TERM-LIST
match(TERM, TERM, ENV, ENV)	→ ENV
match-list(TERM-LIST, TERM-LIST, ENV, ENV)	→ ENV
nomatch	→ ENV
"(" TERM-LIST ")"	→ TERM-LIST {**bracket**}

variables
Bind [0-9′]∗→ ENV
equations

Given a term, a process name and an environment, we can define the result of replacing all variable occurrences in the term by their corresponding value in the environment.

$$\frac{\mathsf{value}(\mathit{Global},\ E) = T,\ \ T \neq \mathsf{undefined}}{\mathsf{substitute}(\mathit{Global},\ E) = T} \quad \textsf{[substitute-1]}$$

$$\frac{\mathsf{substitute\text{-}list}(\mathit{Ts},\ E) = \mathit{Ts}'}{\mathsf{substitute}(\mathit{Id}(\mathit{Ts}),\ E) = \mathit{Id}(\mathit{Ts}')} \quad \textsf{[substitute-2]}$$

$$\mathsf{substitute}(T,\ E) = T \qquad \textbf{otherwise} \quad \textsf{[substitute]}$$

Give this substitution operation, we can trivially define a substitution operation on lists of terms.

$$\text{substitute-list}(T,\ E) = \text{substitute}(T,\ E) \qquad \textbf{[substitute-list-1]}$$

$$\frac{\text{substitute}(T,\ E) = T',\quad \text{substitute-list}(Ts,\ E) = Ts'}{\text{substitute-list}(T,\ Ts,\ E) = T',\ Ts'} \qquad \textbf{[substitute-list-2]}$$

Given two terms (the subject and the pattern), an environment, and a list of variable bindings under construction (also represented by an environment), we define a match operation by an inductive definition according to all possible term structures. In the case of multiple occurrences of the same variable in the subject term, we require that all occurrences match the same term. Observe that match and match-list (see below) are defined in a mutually recursive fashion.

$$\frac{\text{value}(Global,\ [Entries]) = \text{undefined}}{\text{match}(Global,\ T,\ E,\ [Entries]) = [(Global : \text{substitute}(T,\ E)),\ Entries]} \qquad \textbf{[match-1]}$$

$$\frac{\text{value}(Global,\ [Entries]) = \text{substitute}(T,\ E)}{\text{match}(Global,\ T,\ E,\ [Entries]) = [Entries]} \qquad \textbf{[match-2]}$$

$$\text{match}(Int,\ Int,\ E,\ Bind) = Bind \qquad \textbf{[match-3]}$$

$$\text{match}(String,\ String,\ E,\ Bind) = Bind \qquad \textbf{[match-4]}$$

$$\text{match}(Id,\ Id,\ E,\ Bind) = Bind \qquad \textbf{[match-5]}$$

$$\frac{\text{match-list}(Ts_1,\ Ts_2,\ E,\ Bind) = Bind'}{\text{match}(Id(Ts_1),\ Id(Ts_2),\ E,\ Bind) = \text{append}(Bind,\ Bind')} \qquad \textbf{[match-6]}$$

$$\text{match}(T_1,\ T_2,\ E,\ Bind) = \text{nomatch} \quad \textbf{otherwise} \qquad \textbf{[match]}$$

Now, match-list can now be defined using induction over term lists.

$$\text{match-list}(T_1,\ T_2,\ E,\ Bind) = \text{match}(T_1,\ T_2,\ E,\ Bind) \qquad \textbf{[match-list-1]}$$

$$\frac{\text{match}(T_1,\ T_2,\ E,\ Bind) = [Entries_1],\quad \text{match-list}(Ts_1,\ Ts_2,\ E,\ Bind) = [Entries_2]}{\text{match-list}(T_1,\ Ts_1,\ T_2,\ Ts_2,\ E,\ Bind) = [Entries_1,\ Entries_2]} \qquad \textbf{[match-list-2]}$$

$$\text{match-list}(Ts_1,\ Ts_2,\ E,\ Bind) = \text{nomatch} \quad \textbf{otherwise} \qquad \textbf{[match-list]}$$

5.4 Representation of a ToolBus configuration

For the purpose of interpretation, we transform and extend a given ToolBus script in several ways defined below.

Module TB-representation
imports Booleans(5.1) TBscript(3.2.2) Environments(5.2) Events(3.4)
exports
  sorts AP-FORM AP-FORM-REPR TB-REPR SUBSCRIPTIONS
  context-free syntax

definition "[" PNAME "]"	→ PROC-DEF
res-names(TERM-LIST, PNAME, ENV)	→ TERM-LIST
res-names(PROC, PNAME, ENV)	→ PROC
is-atom(PROC)	→ BOOL

```
ATOM ";" PROC                                              → AP-FORM
AP-FORM ";" PROC                                           → AP-FORM
delta                                                      → AP-FORM
"(" AP-FORM ")"                                            → AP-FORM        {bracket}

"<" {AP-FORM ","}* ">"                                     → AP-FORM
AP-FORM ":" ENV ":" SUBSCRIPTIONS ":" EVENT-LIST           → AP-FORM-REPR
"{" {AP-FORM-REPR ","}* "}"                                → TB-REPR

convert(TB-CONFIG)                                         → TB-REPR

"[" {ID ","}* "]"                                          → SUBSCRIPTIONS
                                                           → SUBSCRIPTIONS
has-subscription(ID, SUBSCRIPTIONS)                        → BOOL

expand(PROC)                                               → AP-FORM

create-process(PNAME, {TERM ","}*)                         → AP-FORM-REPR
```

variables

```
Atom             → ATOM
AP [0-9']*       → AP-FORM
APs [0-9']*      → {AP-FORM ","}+
OAPs [0-9']*     → {AP-FORM ","}*
APR [0-9']*      → AP-FORM-REPR
APRs [0-9']*     → {AP-FORM-REPR ","}*
Atomic-Fun       → ATOMIC-FUN
Formals          → FORMALS
Subscriptions    → SUBSCRIPTIONS
S [0-9']*        → SUBSCRIPTIONS
N [0-9']*        → EVENT-LIST
```

equations

First, we have to explain and motivate how we represent TOOLBUS scripts in this specification. In a completely standard approach, we would organize our specification in such a way that all relevant functions have an argument of sort TB-SCRIPT representing the TOOLBUS script being interpreted. This approach has, however, as major disadvantage that nearly *all* functions in the specification get such an argument. This is not surprising, since the current script acts as a global entity for the interpreter. In this specification we avoid this problem *by representing process definitions as equations* and introducing the auxiliary function definition that retrieves the process definition associated with a certain name. To be more specific, the process definition

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

will be represented by the equation:

```
definition[CALC] = define CALC = ( rec-msg(calc, Exp) . snd-eval(calc,Exp) .
                                     rec-value(calc, Val) . snd-reply(calc, Exp, Val)
                                 ) * delta
```

For each process definition in the script, an equation of the above form is required. It should be stressed that this approach is only a non-essential technicality. However, we prefer to use it since it makes the specification somewhat simpler.

Next, we *resolve names* in process expressions:

- We suffix all variable names with the name of the process expression in which they occur. This avoids name clashes when a process name appears inside a process expression and is expanded into its corresponding definition.

- We replace all variables that have an associated value in a given environment E by their value. This is used for formal/actual parameter binding both for process instantiation (e.g., the use of a process name inside a process expression), and for process creation (by means of create). In both cases, formals are replaced by their actual values throughout the process expression associated with the given process name.

$$\text{res-names}(Int,\ Pnm,\ E) = Int \qquad \textbf{[res-names-1]}$$
$$\text{res-names}(String,\ Pnm,\ E) = String \qquad \textbf{[res-names-2]}$$

$$\frac{\text{value}(Var\ \$\ Pnm,\ E) = \text{undefined}}{\text{res-names}(Var,\ Pnm,\ E) = Var\ \$\ Pnm} \qquad \textbf{[res-names-3]}$$
$$\frac{\text{value}(Var\ \$\ Pnm,\ E) = T,\quad T \neq \text{undefined}}{\text{res-names}(Var,\ Pnm,\ E) = T} \qquad \textbf{[res-names-4]}$$

$$\text{res-names}(Id,\ Pnm,\ E) = Id \qquad \textbf{[res-names-5]}$$
$$\text{res-names}(Id(Tlist),\ Pnm,\ E) = Id(\text{res-names}(Tlist,\ Pnm,\ E)) \qquad \textbf{[res-names-6]}$$
$$\text{res-names}(Atomic\text{-}Fun(Tlist),\ Pnm,\ E) = Atomic\text{-}Fun(\text{res-names}(Tlist,\ Pnm,\ E)) \qquad \textbf{[res-names-7]}$$
$$\text{res-names}(\text{create } Pnm(Ts),\ Pnm',\ E) = \text{create } Pnm(\text{res-names}(Ts,\ Pnm',\ E)) \qquad \textbf{[res-names-8]}$$
$$\text{res-names}(\text{delta},\ Pnm,\ E) = \text{delta} \qquad \textbf{[res-names-9]}$$

$$\frac{\text{res-names}(T,\ Pnm,\ E) = T',\quad \text{res-names}(Ts,\ Pnm,\ E) = Ts'}{\text{res-names}(T,\ Ts,\ Pnm,\ E) = T',\ Ts'} \qquad \textbf{[res-names-10]}$$

$$\text{res-names}(P_1 + P_2,\ Pnm,\ E) = \text{res-names}(P_1,\ Pnm,\ E) + \text{res-names}(P_2,\ Pnm,\ E) \qquad \textbf{[res-names-11]}$$
$$\text{res-names}(P_1\ .\ P_2,\ Pnm,\ E) = \text{res-names}(P_1,\ Pnm,\ E)\ .\ \text{res-names}(P_2,\ Pnm,\ E) \qquad \textbf{[res-names-12]}$$
$$\text{res-names}(P_1 * P_2,\ Pnm,\ E) = \text{res-names}(P_1,\ Pnm,\ E) * \text{res-names}(P_2,\ Pnm,\ E) \qquad \textbf{[res-names-13]}$$
$$\text{res-names}(Pnm_1,\ Pnm_2,\ E) = Pnm_1 \qquad \textbf{[res-names-14]}$$

Next, we define further preprocessing of process expressions:

- We transform all (possibly nested) occurrences of $+$ into n-ary lists of the form $<\ P_1\ ,\ \ldots\ ,\ P_n\ >$. In this way, we have actually introduced Process Algebra axiom A2 (associativity). This will permit—in the definition of the interpreter—the use of list matching instead of associative matching over a binary operator (which is not available in ASF+SDF).

- We further transform the expressions P_i into the form $A_i; P_i'$, where A_i is an arbitrary atom.

As a result of the above steps, process expressions have been converted to the *action-prefix form* $<\ A_1; P_1\ ,\ \ldots\ ,\ A_n; P_n\ >$, which will simplify the definition of interpretation in the next section. In the specification, we represent this form by the sort AP-FORM and define a function expand which converts from PROC to AP-FORM. The nesting of (expanded) $+$-operators is eliminated by:

$$<\ OAPs_1,\ <\ APs\ >,\ OAPs_2\ >\ =\ <\ OAPs_1,\ APs,\ OAPs_2\ > \qquad \textbf{[flat-1]}$$

We also need versions of the Process Algebra axioms A3, A4, and A5. Observe that axiom A1 (commutativity) is implicitly defined in equation (flat-2), below.

$$< \mathit{OAPs}_1,\ \mathsf{AP},\ \mathit{OAPs}_2,\ \mathsf{AP},\ \mathit{OAPs}_3 > \; = \; < \mathit{OAPs}_1,\ \mathsf{AP},\ \mathit{OAPs}_2,\ \mathit{OAPs}_3 > \qquad \text{[flat-2]}$$
$$< \mathsf{AP} > ;\ P \; = \; < \mathsf{AP};\ P > \qquad \text{[flat-3]}$$
$$< \mathsf{AP},\ \mathit{APs} > ;\ P \; = \; < \mathsf{AP};\ P,\ < \mathit{APs} > ;\ P > \qquad \text{[flat-4]}$$
$$\mathsf{AP};\ P_1;\ P_2 \; = \; \mathsf{AP};\ P_1\ .\ P_2 \qquad \text{[flat-5]}$$
$$\mathit{Atom};\ P_1;\ P_2 \; = \; \mathit{Atom};\ P_1\ .\ P_2 \qquad \text{[flat-6]}$$
$$< \mathit{OAPs}_1,\ \mathsf{delta},\ \mathit{OAPs}_2 > \; = \; < \mathit{OAPs}_1,\ \mathit{OAPs}_2 > \qquad \text{[flat-7]}$$

The function expand also takes care of expansion of process names and expressions containing the * operator. Recall that a process can be defined by an equation of the form:

define $Pname\ Formals = P$

Whenever $Pname$ appears as the left most operand of a process expression it is changed (by the function expand) into the disambiguated and expanded version of the process expression P. In this way, process names are expanded into their definition only when needed and endless recursion due to the leftmost innermost reduction strategy used by the Asf+Sdf system can be avoided. In a similar fashion, expansion of the process expression $P_1 * P_2$ yields $P_1' \ ;\ P_1 * P_2 + P_2'$, where P_1' and P_2' are the expanded version of P_1, respectively, P_2. In the definition of expand we will need the following auxiliary predicate is-atom which yields true if a given process is an atom. Note that in equation (is-atom-1) the type of the argument is ATOM which makes this equation only applicable to atoms. All other cases are handled by the second equation.

$$\mathsf{is\text{-}atom}(\mathit{Atom}) = \mathsf{true} \qquad \text{[is-atom-1]}$$
$$\mathsf{is\text{-}atom}(P) = \mathsf{false} \quad \textbf{otherwise} \qquad \text{[is-atom]}$$

The definition of expand is:

$$\frac{P_1 \neq \mathsf{delta},\ \ P_2 \neq \mathsf{delta}}{\mathsf{expand}(P_1 + P_2) = < \mathsf{expand}(P_1),\ \mathsf{expand}(P_2) >} \qquad \text{[expand-1]}$$

$$\mathsf{expand}(\mathsf{delta} + P) = \mathsf{expand}(P) \qquad \text{[expand-2]}$$
$$\mathsf{expand}(P + \mathsf{delta}) = \mathsf{expand}(P) \qquad \text{[expand-3]}$$
$$\mathsf{expand}(\mathit{Atom}\ .\ P) = \mathit{Atom};\ P \qquad \text{[expand-4]}$$
$$\mathsf{expand}(\mathsf{delta}\ .\ P) = \mathsf{delta} \qquad \text{[expand-5]}$$

$$\frac{\mathsf{is\text{-}atom}(P_1) = \mathsf{false},\ \ P_1 \neq \mathsf{delta}}{\mathsf{expand}(P_1\ .\ P_2) = \mathsf{expand}(P_1);\ P_2} \qquad \text{[expand-6]}$$

$$\mathsf{expand}((P_1\ .\ P_2)\ .\ P_3) = \mathsf{expand}(P_1\ .\ P_2\ .\ P_3) \qquad \text{[expand-7]}$$
$$\mathsf{expand}(P_1 * P_2) = < \mathsf{expand}(P_1);\ P_1 * P_2,\ \mathsf{expand}(P_2) > \qquad \text{[expand-8]}$$
$$\mathsf{expand}(\mathsf{delta}) = \mathsf{delta} \qquad \text{[expand-9]}$$
$$\mathsf{expand}(\mathit{Atom}) = \mathit{Atom};\ \mathsf{delta} \qquad \text{[expand-10]}$$

$$\frac{\mathsf{definition}[\mathit{Pnm}] = \mathsf{define}\ \mathit{Pnm}\ = P}{\mathsf{expand}(\mathit{Pnm}) = \mathsf{expand}(\mathsf{res\text{-}names}(P,\ \mathit{Pnm},\ []))} \qquad \text{[expand-11]}$$

$$\frac{\begin{array}{c}\mathsf{definition}[\mathit{Pnm}] = \mathsf{define}\ \mathit{Pnm}\ (\mathit{Vars}) = P,\\ E = \mathsf{create\text{-}env}(\mathit{Vars},\ \mathit{Pnm},\ \mathit{Ts})\end{array}}{\mathsf{expand}(\mathit{Pnm}(\mathit{Ts})) = \mathsf{expand}(\mathsf{res\text{-}names}(P,\ \mathit{Pnm},\ E))} \qquad \text{[expand-12]}$$

Next, the test whether a given identifier is a member of the subscriptions of some named process is defined as follows:

$$\text{has-subscription}(Id,\ [Id,\ Ids]) \ = \ \text{true} \qquad \textbf{[has-subs-1]}$$

$$Id \neq Id' \ \Rightarrow \ \text{has-subscription}(Id,\ [Id',\ Ids]) \ = \ \text{has-subscription}(Id,\ [Ids]) \qquad \textbf{[has-subs-2]}$$

$$\text{has-subscription}(Id,\ []) \ = \ \text{false} \qquad \textbf{[has-subs-3]}$$

$$\text{has-subscription}(Id,\ ) \ = \ \text{false} \qquad \textbf{[has-subs-4]}$$

Finally, we define how a given configuration (TB-CONFIG) can be transformed into an equivalent toolbus representation(TB-REPR). We associate with each process an environment E, a subscription list S, and an event list N representing all notes that have been sent to the process but have not yet been read by it. Each process is thus represented as: $< AP_1,\ \ldots,\ AP_n >: E : S : N$. As a result of all these transformations, the complete ToolBus configuration will have the form:

$$\{<AP_{11}, \ldots, AP_{1n_1}>: E_1 : S_1 : N_1, \ldots,\ <AP_{m1}, \ldots, AP_{mn_m}>: E_m : S_m : N_m\}.$$

The transformation is defined in two steps. First, create-process defines the construction of a process representation given a process name with or without actual parameters. This function will be used in two manners: for the construction of the initial ToolBus representation and, during interpretation, for the construction of a new process resulting from a create atom.

$$\frac{\begin{array}{c}\text{definition}[Pnm] = \text{define } Pnm\ = P, \\ \text{AP} = \text{expand}(\text{res-names}(P,\ Pnm,\ []))\end{array}}{\text{create-process}(Pnm,\ ) = <\text{AP}> : [] : [] : []} \qquad \textbf{[create-1]}$$

$$\frac{\begin{array}{c}\text{definition}[Pnm] = \text{define } Pnm\ (Vars) = P, \\ E = \text{create-env}(Vars,\ Pnm,\ Ts), \\ \text{AP} = \text{expand}(\text{res-names}(P,\ Pnm,\ E))\end{array}}{\text{create-process}(Pnm,\ Ts) = <\text{AP}> : [] : [] : []} \qquad \textbf{[create-2]}$$

The complete ToolBus representation can now be constructed as follows:

$$\frac{\text{convert}(\text{toolbus}(Pnms)) = \{APRs\}}{\text{convert}(\text{toolbus}(Pnm,\ Pnms)) = \{\text{create-process}(Pnm,\ ),\ APRs\}} \qquad \textbf{[tb-1]}$$

$$\text{convert}(\text{toolbus}(Pnm)) \ = \ \{\text{create-process}(Pnm,\ )\} \qquad \textbf{[tb-2]}$$

5.5 An interpreter for ToolBus scripts

The interpretater we are interested in takes a ToolBus configuration and an event and calculates the effect of the event. Complete system behaviour is described by defining the effect of a sequence of events on a given ToolBus configuration.

The state of the interpreter (ITP-STATE) is represented by a pair: a list of previous events (a history of all events, i.e., both events coming from tools and events generated by the ToolBus), and a ToolBus representation (TB-REPR).

Now we define the complete interpreter, including several auxiliary functions.

Module Interpreter
imports TBscript$^{(3.2.2)}$ Events$^{(3.4)}$ Match$^{(5.3)}$ TB-representation$^{(5.4)}$
exports
sorts ITP-STATE

context-free syntax

EVENT-LIST "#" TB-REPR	$\rightarrow$ ITP-STATE
msg-steps(TB-REPR)	$\rightarrow$ TB-REPR
distr-note(EVENT, TB-REPR)	$\rightarrow$ TB-REPR
note-steps(TB-REPR)	$\rightarrow$ TB-REPR
atomic-steps(TB-REPR)	$\rightarrow$ ITP-STATE
all-internal-steps(ITP-STATE)	$\rightarrow$ ITP-STATE
rec-from-tool-step(EVENT, TB-REPR)	$\rightarrow$ ITP-STATE
interpret1(EVENT, ITP-STATE)	$\rightarrow$ ITP-STATE
interpret(EVENT-LIST, ITP-STATE)	$\rightarrow$ ITP-STATE
interpret(EVENT-LIST, TB-CONFIG)	$\rightarrow$ ITP-STATE

variables

InterpState [0-9]*$\rightarrow$ ITP-STATE
TBconfig $\rightarrow$ TB-CONFIG

equations

Message communication. msg-steps transforms a given ToolBus representation into a new one such that all possible message communication has been performed. This concerns snd-msg and rec-msg.

$$\frac{\begin{array}{c}\text{APR}_1 = <\mathit{OAPs}', \text{snd-msg}(\mathit{Tlist}_1);\ P_1,\ \mathit{OAPs}'' > : E_1 : S_1 : N_1,\\ \{\mathit{APRs}_1,\ \mathit{APRs}_2\} = \{\mathit{APRs}_3,\ \text{APR}_2,\ \mathit{APRs}_4\},\\ \text{APR}_2 = <\mathit{OAPs}''', \text{rec-msg}(\mathit{Tlist}_2);\ P_2,\ \mathit{OAPs}'''' > : E_2 : S_2 : N_2,\\ [\mathit{Entries}] = \text{match-list}(\mathit{Tlist}_2,\ \mathit{Tlist}_1,\ E_1,\ []),\\ \text{APR}_1{}' = <\text{expand}(P_1)> : E_1 : S_1 : N_1,\\ \text{APR}_2{}' = <\text{expand}(P_2)> : \text{update}([\mathit{Entries}],\ E_2) : S_2 : N_2\end{array}}{\text{msg-steps}(\{\mathit{APRs}_1,\ \text{APR}_1,\ \mathit{APRs}_2\}) = \text{msg-steps}(\{\mathit{APRs}_3,\ \mathit{APRs}_4,\ \text{APR}_1{}',\ \text{APR}_2{}'\})} \quad \text{[msg-1]}$$

$$\text{msg-steps}(\{\mathit{APRs}\}) = \{\mathit{APRs}\} \qquad \textbf{otherwise} \quad \text{[msg]}$$

Note communication. note-steps transforms a given ToolBus representation into a new one such that all possible note communication has been performed. First, we define the auxiliary function distr-note for distributing a note to all subscribed ToolBus processes.

$$\frac{\begin{array}{c}\text{APR} = <\mathit{OAPs}> : E : S : [\mathit{Events}],\\ \text{has-subscription}(\mathit{Id},\ S) = \text{true},\\ \{\mathit{APRs}'\} = \text{distr-note}(\text{note}(\mathit{Id},\ \mathit{OptTs}),\ \{\mathit{APRs}\}),\\ \text{APR}' = <\mathit{OAPs}> : E : S : [\mathit{Events},\ \text{note}(\mathit{Id},\ \mathit{OptTs})]\end{array}}{\text{distr-note}(\text{note}(\mathit{Id},\ \mathit{OptTs}),\ \{\text{APR},\ \mathit{APRs}\}) = \{\text{APR}',\ \mathit{APRs}'\}} \quad \text{[dn-1]}$$

$$\frac{\begin{array}{c}\text{APR} = <\mathit{OAPs}> : E : S : [\mathit{Events}],\\ \text{has-subscription}(\mathit{Id},\ S) = \text{false},\\ \{\mathit{APRs}'\} = \text{distr-note}(\text{note}(\mathit{Id},\ \mathit{OptTs}),\ \{\mathit{APRs}\})\end{array}}{\text{distr-note}(\text{note}(\mathit{Id},\ \mathit{OptTs}),\ \{\text{APR},\ \mathit{APRs}\}) = \{\text{APR},\ \mathit{APRs}'\}} \quad \text{[dn-2]}$$

$$\text{distr-note}(\mathit{Event},\ \{\}) = \{\} \quad \text{[dn-3]}$$

The function note-steps can now treat the atoms snd-note, rec-note, and no-note.

$$\frac{\begin{array}{c}\text{APR} = <\mathit{OAPs}', \text{snd-note}(\mathit{Tlist});\ P,\ \mathit{OAPs}'' > : E : S : N,\\ \{\mathit{APRs}'\} = \text{distr-note}(\text{note}(\text{substitute-list}(\mathit{Tlist},\ E)),\ \{\mathit{APRs}_1,\ \mathit{APRs}_2\}),\\ \text{APR}' = <\text{expand}(P)> : E : S : N\end{array}}{\text{note-steps}(\{\mathit{APRs}_1,\ \text{APR},\ \mathit{APRs}_2\}) = \text{note-steps}(\{\mathit{APRs}',\ \text{APR}'\})} \quad \text{[note-1]}$$

$$\frac{\begin{array}{c}\mathsf{APR} = <\mathit{OAPs}', \text{rec-note}(\mathit{Tlist});\ P,\ \mathit{OAPs}''>\ : E : S : [\mathit{Events}, \text{note}(\mathit{Tlist}_1), \mathit{Events}'],\\ [\mathit{Entries}] = \text{match-list}(\mathit{Tlist}, \mathit{Tlist}_1, E, []),\\ \mathsf{APR}' = <\text{expand}(P)>\ : \text{update}([\mathit{Entries}], E) : S : [\mathit{Events}, \mathit{Events}']\end{array}}{\text{note-steps}(\{\mathit{APRs}_1, \mathsf{APR}, \mathit{APRs}_2\}) = \text{note-steps}(\{\mathit{APRs}_1, \mathsf{APR}', \mathit{APRs}_2\})} \quad \textbf{[note-2]}$$

$$\frac{\begin{array}{c}\mathsf{APR} = <\mathit{OAPs}', \text{no-note}(\mathit{Tlist});\ P,\ \mathit{OAPs}''>\ : E : S : N,\\ \text{note}(\mathit{Tlist}) \in N = \text{false},\\ \mathsf{APR}' = <\text{expand}(P)>\ : E : S : N\end{array}}{\text{note-steps}(\{\mathit{APRs}_1, \mathsf{APR}, \mathit{APRs}_2\}) = \text{note-steps}(\{\mathit{APRs}_1, \mathit{APRs}_2, \mathsf{APR}'\})} \quad \textbf{[note-3]}$$

$$\text{note-steps}(\{\}) = \{\} \quad \textbf{[note-4]}$$

$$\frac{\text{note-steps}(\{\mathit{APRs}\}) = \{\mathit{APRs}'\}}{\text{note-steps}(\{\mathsf{APR}, \mathit{APRs}\}) = \{\mathsf{APR}, \mathit{APRs}'\}} \quad \textbf{otherwise} \quad \textbf{[note]}$$

Other atomic steps inside the ToolBus. All other internal steps of the ToolBus are determined by atomic-steps which takes a ToolBus representation as argument and returns an interpreter state, i.e., a list of events resulting from the internal steps and a new ToolBus representation. The atoms snd-eval, snd-do, snd-ack-event, subscribe, unsubscribe, and create are treated here. In addition, we take care of processes without any further choices.

$$\frac{\begin{array}{c}\mathsf{APR} = <\mathit{OAPs}', \text{snd-eval}(\mathit{Ts});\ P,\ \mathit{OAPs}''>\ : E : S : N,\\ \mathsf{APR}' = <\text{expand}(P)>\ : E : S : N,\\ \text{atomic-steps}(\{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}\end{array}}{\text{atomic-steps}(\{\mathsf{APR}, \mathit{APRs}\}) = [\text{eval}(\text{substitute-list}(\mathit{Ts}, E)), \mathit{Events}]\ \#\ \{\mathsf{APR}', \mathit{APRs}'\}} \quad \textbf{[as-1]}$$

$$\frac{\begin{array}{c}\mathsf{APR} = <\mathit{OAPs}', \text{snd-do}(\mathit{Ts});\ P,\ \mathit{OAPs}''>\ : E : S : N,\\ \mathsf{APR}' = <\text{expand}(P)>\ : E : S : N,\\ \text{atomic-steps}(\{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}\end{array}}{\text{atomic-steps}(\{\mathsf{APR}, \mathit{APRs}\}) = [\text{do}(\text{substitute-list}(\mathit{Ts}, E)), \mathit{Events}]\ \#\ \{\mathsf{APR}', \mathit{APRs}'\}} \quad \textbf{[as-2]}$$

$$\frac{\begin{array}{c}\mathsf{APR} = <\mathit{OAPs}', \text{snd-ack-event}(T);\ P,\ \mathit{OAPs}''>\ : E : S : N,\\ \mathsf{APR}' = <\text{expand}(P)>\ : E : S : N,\\ \text{atomic-steps}(\{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}\end{array}}{\text{atomic-steps}(\{\mathsf{APR}, \mathit{APRs}\}) = [\text{ack-event}(\text{substitute}(T, E)), \mathit{Events}]\ \#\ \{\mathsf{APR}', \mathit{APRs}'\}} \quad \textbf{[as-3]}$$

The next two cases, subscribe and unsubscribe manipulate the subscription list of the process in which they appear. Observe that the subscription list is a multi-set of note names.

$$\frac{\begin{array}{c}\mathsf{APR} = <\mathit{OAPs}', \text{subscribe}(\mathit{Id});\ P,\ \mathit{OAPs}''>\ : E : [\mathit{Ids}] : N,\\ \mathsf{APR}' = <\text{expand}(P)>\ : E : [\mathit{Ids}, \mathit{Id}] : N,\\ \text{atomic-steps}(\{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}\end{array}}{\text{atomic-steps}(\{\mathsf{APR}, \mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathsf{APR}', \mathit{APRs}'\}} \quad \textbf{[as-4]}$$

$$\frac{\begin{array}{c}\mathsf{APR} = <\mathit{OAPs}', \text{unsubscribe}(\mathit{Id});\ P,\ \mathit{OAPs}''>\ : E : [\mathit{Ids}, \mathit{Id}, \mathit{Ids}'] : N,\\ \mathsf{APR}' = <\text{expand}(P)>\ : E : [\mathit{Ids}, \mathit{Ids}'] : N,\\ \text{atomic-steps}(\{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}\end{array}}{\text{atomic-steps}(\{\mathsf{APR}, \mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathsf{APR}', \mathit{APRs}'\}} \quad \textbf{[as-5]}$$

Finally, we consider process creation and destruction. On the one hand, process creation (create) *increases* the number of processes:

$$\frac{\begin{array}{c}\mathsf{APR}_1 = < \mathit{OAPs}', \mathsf{create}\ \mathit{Pnm}(\mathit{Ts});\ P,\ \mathit{OAPs}'' > : E : S : N,\\ \mathsf{APR}_1{}' = < \mathsf{expand}(P) > : E : S : N,\\ \mathsf{APR}_2 = \mathsf{create\text{-}process}(\mathit{Pnm}, \mathit{Ts}),\\ \mathsf{atomic\text{-}steps}(\{\mathit{APRs}, \mathsf{APR}_2\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}\end{array}}{\mathsf{atomic\text{-}steps}(\{\mathsf{APR}_1, \mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathsf{APR}_1{}', \mathit{APRs}'\}} \quad \text{[as-6]}$$

On the other hand, a process that has no choices left can be removed and thus *decreases* the number of processes.

$$\frac{\begin{array}{c}\mathsf{APR} = <\ > : E : S : N,\\ \mathsf{atomic\text{-}steps}(\{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}\end{array}}{\mathsf{atomic\text{-}steps}(\{\mathsf{APR}, \mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}} \quad \text{[as-7]}$$

$$\mathsf{atomic\text{-}steps}(\{\}) \ = \ []\ \#\ \{\} \quad \text{[as-8]}$$

$$\frac{\mathsf{atomic\text{-}steps}(\{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}'\}}{\mathsf{atomic\text{-}steps}(\{\mathsf{APR}, \mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathsf{APR}, \mathit{APRs}'\}} \quad \textbf{otherwise} \quad \text{[as]}$$

Internal steps. The function all-internal-steps determines all steps inside the ToolBus: it transforms a given ToolBus representation into a new one such that all possible internal steps have been performed.

$$\frac{\begin{array}{c}\mathsf{atomic\text{-}steps}(\mathsf{note\text{-}steps}(\mathsf{msg\text{-}steps}(\{\mathit{APRs}\}))) = [\mathit{Events}']\ \#\ \{\mathit{APRs}'\},\\ \{\mathit{APRs}\} \neq \{\mathit{APRs}'\}\end{array}}{\mathsf{all\text{-}internal\text{-}steps}([\mathit{Events}]\ \#\ \{\mathit{APRs}\}) = \mathsf{all\text{-}internal\text{-}steps}([\mathit{Events}, \mathit{Events}']\ \#\ \{\mathit{APRs}'\})} \quad \text{[atb-1]}$$

$$\mathsf{all\text{-}internal\text{-}steps}([\mathit{Events}]\ \#\ \{\mathit{APRs}\}) = [\mathit{Events}]\ \#\ \{\mathit{APRs}\} \quad \textbf{otherwise} \quad \text{[atb]}$$

Communication from tools. The function rec-from-tool-step interprets the atoms rec-value and rec-event which represent events from tools: it takes an event and a ToolBus representation as arguments and returns an interpreter state.

$$\frac{\begin{array}{c}\{\mathit{APRs}\} = \{\mathit{APRs}_1, \mathsf{APR}, \mathit{APRs}_2\},\\ \mathsf{APR} = < \mathit{OAPs}', \mathsf{rec\text{-}value}(\mathit{Tlist});\ P,\ \mathit{OAPs}'' > : E : S : N,\\ [\mathit{Entries}] = \mathsf{match\text{-}list}(\mathit{Tlist}, \mathit{Tlist}_1, E, []),\\ \mathsf{APR}' = < \mathsf{expand}(P) > : \mathsf{update}([\mathit{Entries}], E) : S : N\end{array}}{\mathsf{rec\text{-}from\text{-}tool\text{-}step}(\mathsf{value}(\mathit{Tlist}_1), \{\mathit{APRs}\}) = []\ \#\ \{\mathit{APRs}_1, \mathsf{APR}', \mathit{APRs}_2\}} \quad \text{[rt-1]}$$

$$\frac{\begin{array}{c}\{\mathit{APRs}\} = \{\mathit{APRs}_1, \mathsf{APR}, \mathit{APRs}_2\},\\ \mathsf{APR} = < \mathit{OAPs}, \mathsf{rec\text{-}event}(\mathit{Tlist});\ P,\ \mathit{OAPs}' > : E : S : N,\\ [\mathit{Entries}] = \mathsf{match\text{-}list}(\mathit{Tlist}, \mathit{Tlist}_1, E, []),\\ \mathsf{APR}' = < \mathsf{expand}(P) > : \mathsf{update}([\mathit{Entries}], E) : S : N\end{array}}{\mathsf{rec\text{-}from\text{-}tool\text{-}step}(\mathsf{event}(\mathit{Tlist}_1), \{\mathit{APRs}\}) = []\ \#\ \{\mathit{APRs}_1, \mathsf{APR}', \mathit{APRs}_2\}} \quad \text{[rt-2]}$$

Interpretation. The ultimate interpretation function is interpret, which takes a list of events and a ToolBus configuration as input, and produces the resulting interpreter state as result. It is defined using two auxiliary functions:

$$\frac{\mathsf{rec\text{-}from\text{-}tool\text{-}step}(\mathit{Event}, \{\mathit{APRs}\}) = [\mathit{Events}']\ \#\ \{\mathit{APRs}'\}}{\mathsf{interpret1}(\mathit{Event}, [\mathit{Events}]\ \#\ \{\mathit{APRs}\}) = \mathsf{all\text{-}internal\text{-}steps}([\mathit{Events}, \mathit{Event}, \mathit{Events}']\ \#\ \{\mathit{APRs}'\})} \quad \text{[itp1-1]}$$

$$\text{interpret}([\mathit{Event}, \mathit{Events}], \mathit{InterpState}) = \text{interpret}([\mathit{Events}], \text{interpret1}(\mathit{Event}, \mathit{InterpState})) \quad \textbf{[itp-1]}$$

$$\text{interpret}([], \mathit{InterpState}) = \mathit{InterpState} \quad \textbf{[itp-2]}$$

$$\frac{\{\mathit{APRs}\} = \text{convert}(\text{toolbus}(\mathit{Pnms}))}{\begin{array}{l}\text{interpret}([\mathit{Events}], \text{toolbus}(\mathit{Pnms})) = \\ \text{interpret}([\mathit{Events}], \text{all-internal-steps}([] \ \# \ \{\mathit{APRs}\}))\end{array}} \quad \textbf{[itp-3]}$$

6 Example: a calculator

6.1 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:

- 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.

6.2 ToolBus script

```
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
```

6.3 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)`

7 Example: a text editor with window resize

7.1 Informal description

Now we consider a simplified version of a situation that we have encountered in the implementation of the ASF+SDF Meta-environment: a text editor with three commands `parse` (parse the current text in the editor and return its parse tree) `change-text` (make a modification to the text), `resize` (resize the window in which the text is being displayed). A resize command may be given while a parse command is being executed. Four processes are used to model this situation:

- The user-interface process `UI1` for handling the commands `parse` and `change-text`.
- The user-interface process `UI2` for handling the command `resize`.
- The process `PARSE` that controls the parsing tool.
- The process `TE` that controls the text editing tool.

7.2 ToolBus script

```
define UI1 =
    ( rec-event(ui(U), parse, Str) . snd-msg(parse, Str) .
        rec-msg(parse, Str, Tree) . snd-ack-event(ui(U))
    + rec-event(ui(U), change-text, Delta) .
        snd-msg(change-text, Delta) . snd-ack-event(ui(U))
    ) * delta

define UI2 =
    ( rec-event(ui(U), resize, Win, X, Y) . snd-msg(resize, Win, X, Y) .
        snd-ack-event(ui(U))
    ) * delta

define PARSE =
    ( rec-msg(parse, Str) . snd-eval(parse, Str) . rec-value(parse, Tree) .
        snd-msg(parse, Str, Tree)
    ) * delta

define TE =
    ( rec-msg(change-text, Delta) . snd-do(te, change-text(Delta))
    + rec-msg(resize, Win, X, Y) . snd-do(te,resize(Win, X, Y))
    ) * delta
```

8 Example: an environment for syntax-directed editing

8.1 Informal description

Now we present a slightly more ambitious example that aims at describing an environment for syntax directed editing. We want to be able to open and close an arbitrary number of syntax-directed editors on different files.

Imagine a toplevel user interface containing three push buttons: `Edit`, `Close`, and `Quit`:

- Pushing `Edit` results in a dialogue asking for a file name. Once the file name has been provided by the user, a syntax-directed editor for the file is created.
- Pushing `Close`, also results in a dialogue asking for a file name in order to identify the editor instance to be closed.
- Push `Quit` closes all editors and terminates the execution of the whole environment.

The next question is how to model syntax-directed editing and how to coordinate text editing and syntax-directed editing. We assume the existence of two tools: a user-interface (`ui`) providing text editors and a separate tool (`syn-edit`) providing structure information. The idea is now that a text editor generates events whenever information is needed from a syntax-directed editor. For instance, an "`up`" event from the text editor will lead to a request to the syntax-directed editor to calculate a new focus, representing begin and end point of the text area representing the parent of the current focus. This focus information can then be used to adjust the focus area in the text editor.

Several features are illustrated in this example:

- For each editor a new process is created using `create`.
- The quit operation is achieved by broadcasting a `quit` note to all editor processes.
- `ED-STARTUP`, `ED-COMMAND`, and `ED-SHUTDOWN` are examples of auxiliary, parameterized process definitions.

8.2 ToolBus script

```
define TOPLEVEL =
    snd-do(ui, toplevel) .
    ( rec-event(ui(top), edit(Filename)) .
        create(EDITOR,Filename) . snd-ack-event(ui(top))
    + rec-event(ui(top), close(Filename)) .
        snd-msg(editor(Filename), close) . snd-ack-event(ui(top))
    ) * rec-event(ui(top), quit) . snd-note(quit) . snd-ack-event(ui(top)) .
    snd-do(ui, shutdown)

define ED-STARTUP (Filename) =
    snd-eval(syn-edit, edit(Filename)) .
    ( rec-value(syn-edit(editor(Filename)), error(Msg)) .
        snd-do(ui, displayError(Msg)) . delta
    + rec-value(syn-edit(editor(Filename)), ok)
    ) .
    snd-eval(ui, txt-edit(Filename)) .
    ( rec-value(ui(editor(Filename)), error(Msg)) .
        snd-do(ui, displayError(Msg)) . delta
    + rec-value(ui(editor(Filename)), ok)
    )

define ED-COMMAND (Filename) =
    ( rec-event(ui(editor(Filename)), up) .
                snd-eval(syn-edit(editor(Filename)), up)
    + rec-event(ui(editor(Filename)), down) .
                snd-eval(syn-edit(editor(Filename)), down)
    + rec-event(ui(editor(Filename)), next) .
                snd-eval(syn-edit(editor(Filename)), next)
    + rec-event(ui(editor(Filename)), mouse(X,Y)) .
                snd-eval(syn-edit(editor(Filename)), mouse(X,Y))
    ) .
    rec-value(syn-edit(editor(Filename)), Focus) .
    snd-do(ui(editor(Filename)), setFocus(Focus)) .
    snd-ack-event(ui(editor(Filename)))

define ED-SHUTDOWN (Filename) =
    ( rec-msg(editor(Filename), close)
    + rec-note(quit)
    ) .
    snd-do(syn-edit(editor(Filename)), close) .
    snd-do(ui(editor(Filename)), close)

define EDITOR (Filename) =
    subscribe(quit) .
    ED-STARTUP(Filename) . ED-COMMAND(Filename) * ED-SHUTDOWN(Filename)

toolbus(TOPLEVEL)
```

9 Example: functional versus non-functional tools

The ToolBus architecture imposes only constraints on tools regarding their communication behaviour. No assumptions are made about the internal state of each tool and, therefore, both functional ("stateless") and non-functional ("state-full") tools can be connected. The purpose of the example in this section is to show how state information at the level of the ToolBus can be used in combination with completely functional tools.

Consider the problem of generating unique symbols as, for instance, needed during code generation in a compiler when generating unique labels. A typical "gensym" tool will internally maintain a global counter and will increase the counter each time a new symbol is requested. A script controlling such a non-functional gensym tool is now simply:

```
define UNIQUE-NAMES1 =
    ( rec-msg(unique-name) . snd-eval(gensym, new-symbol) .
      rec-value(gensym, Symbol) . snd-msg(unique-name, Symbol)
    ) * delta
```

Note that this script is incomplete since we do not give definitions of processes communicating with `UNIQUE-NAMES1`. Now suppose that we are interested in connecting a completely functional tool "fgensym" for generating unique names. Two issues have to be addressed in this case: initialization of the global counter and saving and restoring the counter between successive calls. This can be done as follows:

```
define UNIQUE-NAMES2 =
    snd-eval(fgensym, initial-counter) . rec-value(fgensym, Counter) .
    ( rec-msg(unique-name) . snd-eval(fgensym, new-symbol(Counter)) .
      rec-value(gensym, result(Symbol, Counter)) . snd-msg(unique-name, Symbol)
    ) * delta
```

The local variable `Counter` of the process `UNIQUE-NAMES2` now preserves the global state of the tool. Here follows one possible definition of the functional tool "fgensym" in ASF+SDF (we leave the function make-symbol, that converts integers into symbols, unspecified):

Module FGensym
imports Integers$^{(5.1)}$
exports
  sorts SYMBOL RESULT
  context-free syntax
    make-symbol(INT) → SYMBOL

    result(SYMBOL, INT) → RESULT
    initial-counter → INT
    new-symbol(INT) → RESULT
equations

$$\text{initial-counter} = 0 \qquad \text{[initial-counter-1]}$$

$$\text{new-symbol}(\mathit{Int}) = \text{result}(\text{make-symbol}(\mathit{Int} + 1),\ \mathit{Int} + 1) \qquad \text{[new-symbol-1]}$$

Although this example is very simple, it can be scaled up to more complex situations. Typically, during the traversal of a recursive datastructure, a tool needs a service from another tool, for instance, to interact with the user. If the traversal tool is required to be functional, it has to be written in a special style. Instead of making use of a recursive traversal function, the progress of the traversal should be represented explicitly and the traversal function should be written as a transformation from one traversal state to the next one. This global state can then be preserved at the ToolBus level.

10 An experimental ToolBus interpreter implemented in C

10.1 Introduction

Our final—and most concrete—level of description of the ToolBus takes the form of an implementation in C. We will describe an interpreter for ToolBus scripts that reads a script, creates all tools mentioned in it, sets up the ToolBus configuration and then starts interpreting it. Before describing the implementation we will now first make some extension to ToolBus scripts and motivate our implementation choices.

The notion of a "tool". Until now a "tool" has only been characterized by its name as appearing as first argument of the atomic actions `snd-eval`, `rec-value`, `snd-do`, `rec-event`, or `snd-ack-event`. At the implementation level, we have to become more specific, i.e., which command is needed to execute the tool, on which machine should it be executed, etc. In order to capture these details, we extend ToolBus scripts with tool definitions in the following manner:

Module ExtendedTBscript
imports TBscript$^{(3.2.2)}$
exports
  sorts TOOL-FEATURE TOOL-DEF EXT-TB-SCRIPT
  context-free syntax
    host "=" STRING → TOOL-FEATURE
    command "=" STRING → TOOL-FEATURE
    tool ID "{" TOOL-FEATURE* "}" → TOOL-DEF
    PROC-DEF+ TOOL-DEF+ TB-CONFIG → EXT-TB-SCRIPT

Implementation options. There are many methods for implementing the interpretation of ToolBus scripts, ranging from purely interpretative methods that closely follow the Asf+Sdf specification given in Section 5, to fully compilational methods that first transform the ToolBus 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.

10.2 Overview of implementation

The ToolBus interpreter. The ToolBus itself is implemented as a separate Unix process that interprets a given ToolBus 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: for all tools with a non-empty command in their definition execute the tool as a separate Unix process. All other tools are not yet executed, but it is assumed that their execution will be started independently and that they 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 very much resembling the TB-REPR defined in Section 5.4.
- Execute:
  1. wait for an event coming from one of the tools;
  2. compute the effect of the event on the ToolBus state;

```
#include "TB.h"
term *handle_input_from_toolbus(term *e)
{
  ...
}

main(int argc, char *argv[])
{
  TBinit("toolname", argc, argv, handle_input_from_toolbus);
  TBeventloop();
}
```

Figure 2: Global skeleton of a tool.

3. perform any internal communication steps;
4. perform any atomic actions.
5. repeat.

The tool/ToolBus interconnection protocol. Since ToolBus and tools execute in a completely distributed environment, a full client/server protocol has to be used to establish a connection between tool (client) and ToolBus (server). We use a protocol that resembles the one used in the X-window system. The ToolBus has two "well-known sockets", i.e., sockets whose names are known to all tools, to initiate connections. At the *tool* side the steps are:

1. Connect to ToolBus input port.
2. Send tool name and host name to ToolBus.
3. Connect to ToolBus output port.
4. Get port identification from ToolBus.
5. Create input and output ports for tool using port identification.
6. Synchronize with ToolBus.
7. Start execution of tool.

At the ToolBus side the steps are:

1. Wait for connection with ToolBus input port.
2. Get tool name and host name from tool.
3. Wait for connection with ToolBus output port.
4. Compute a unique port identification for the tool and send it to tool.
5. Create ToolBus side of input and output ports for the tool using the port identification.
6. Synchronize with tool.
7. Continue execution of ToolBus.

```
#include "TB.h"

term *handle_input_from_toolbus(term *e)
{
  term *tool, *expr;

  if(TBmatch(e, "eval(%t,%t)", &tool, &expr)){
    return TBmakeTerm("value(%t,%d)", tool, calculate(expr));
  } else {
    TBmsg("wrong event received: %t\n", e);
    return NULL;
  }
}

int calculate(term *t)
{ int n;
  char *s;
  term *t1, *t2;

  if(TBmatch(t, "%d", &n))
     return n;
  else if(TBmatch(t, "%s", &s))
          return atoi(s);
  else if(TBmatch(t, "plus(%t,%t)", &t1, &t2))
          return calculate(t1) + calculate(t2);
  else if(TBmatch(t, "times(%t,%t)", &t1, &t2))
          return calculate(t1) * calculate(t2);
  else {
        TBmsg("panic in calculate: %t\n", t);
        return 0;
      }
}

main(int argc, char *argv[])
{
  TBinit("calc", argc, argv, handle_input_from_toolbus);
  TBeventloop();
}
```

Figure 3: Complete listing of the `calc` tool.

Tools. Each tool has the global structure sketched in Figure 2. In the main program, first all ToolBus related initialization is performed (`TBinit`). In particular, the connection with the ToolBus interpreter is made and the event handler for incoming ToolBus events is established (**`handle_input_from_toolbus`**). This event handler gets the incoming event in the form of a **`term`** (a predefined data type), analyzes it, performs arbitrary computations and returns a new term that will be send back to the ToolBus. The complete listing of the calculator tool (see Section 6) is shown in Figure 3. For the prototyping of the user-interface tools used in the examples in this paper we have made extensive use of Tcl/Tk [Ous94].

The tool library. For tool writers there exists a small library of interfacing functions for matching and writing terms. The tool library contains the following functions:

`TBinit(ToolName, argc, argv, event_handler)` initialization of the tool, connect to ToolBus, and establish event handler.

`TBeventloop()` the tool event loop: wait for incoming event from ToolBus and call the tool's event handler.

`TBaddInPort(Fd, event_handler)` establish another event handler for input port `Fd`.

	equations	ratio	lines	ratio
Process Algebra	$63^{(1)}$	1	$110^{(3)}$	1
Asf+Sdf	$99^{(2)}$	1.6	$397^{(4)}$	3.6
C	–	–	$3401^{(5)}$	30.1

Notes

(1) The 44 equations from Appendix A plus 19 equations given in Section 4.

(2) The 99 equations from Section 5.

(3) The 110 lines in Section 4 (i.e., equations plus comments).

(4) The 397 lines of the specification in Section 5, including comments.

(5) The 3401 lines (`.h` and `.c` files) from Appendix C.

Figure 4: Relative sizes of abstraction levels.

`TBmatch(Trm, Pattern, Vars, ...)` match `Trm` against `Pattern` and make assignments to `Vars` when a submatch is found. `Pattern` should be a well-formed, textual representation of a term which may contain the following:

- `%d` : matches an integer;
- `%s` : matches a string;
- `%t` : matches a term.

`TBmakeTerm(Pattern, Vars)` constructs a term according to `Pattern`, where occurrences of `%d`, `%s`, `%t` (see above) are replaced by the values appearing in `Vars`.

`TBwriteTerm(File, Trm)` writes `Trm` to `File`.

`TBmsg(Pattern, Vars, ...)` writes a message on the standard error output stream.

11 Discussion

The resuls of this experiment can be viewed from two different angles. The ToolBus architecture itself seems to be one possible solution to the component interconnection problem, while the method used to arrive at this design has some merits of its own.

The examples given in the paper show that even sophisticated systems can be expressed concisely in a ToolBus script. Preliminary measurements show that the ToolBus interpreter can handle in the order of 100 events per second on a standard workstation. This is clearly sufficient for constructing systems where the events are generated by users (as opposed to programs). As already indicated in Section 10, our experimental C implementation leaves many options for optimization unused.

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. Some quantitative aspects of our approach are given in Figure 4.

In Section 1.1, we already made the observation that tool integration is just one instance of the more general component interconnection problem. Further research is needed to study the effects of changing the nature or granularity of components on the interconnection architecture we have proposed here. This might shed some new light on subjects like modularization, parameterization of datatypes, module interconnection languages, and structured system design.

Acknowledgements

The problems addressed in this paper originate from work done by Huub Bakker, Wilco Koorn and Paul Vriend on a more distributed implementation of the ASF+SDF Meta-environment [Kli93]. This work is described in [KB93] and aimed at

- replacement of the existing text-editing facilities by an external text-editor (Emacs);
- isolation of all user-interface management in a separate process.

Although a first implementation along these lines was completed, it also became apparent that a more rigorous approach was needed to control the communication between the components. A first step in that direction was made by Bas van Vlijmen and Arjen van Waveren, who wrote—in cooperation with the trio already mentioned—a PSF specification describing this communication [vVVvW94]. This specification uncovered several communication problems and potential deadlocks in the implementation. Dissatisfaction with the size and complexity of this PSF specification finally led to the idea of a ToolBus that provides a built-in communication protocol, thus permitting shorter system specifications.

During the preparation of this paper, we had many interesting discussions about tool integration and distributed computing with, amongst others, Mark van den Brand, Dinesh, Jan Heering, Jasper Kamperman, Wilco Koorn, Robert van Liere, Bas van Vlijmen, Pum Walters, Jos van Wamel, and Arjen van Waveren.

With respect to the C implementation, Robert van Liere was a continuous source of help and suggestions. The implementation of socket communication in Sections C.9, C.9.3, and D.3 is closely patterned after code he has made available to us. Eelco Visser's "ToLaTeX" package for typesetting ASF+SDF specifications was of great help during the preparation of this paper.

References

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

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge University Press, 1990.

[BB88] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78:205–245, 1988.

[BBKW89] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, and W.P. Weijland. Term rewriting systems with rule priorities. *Theoretical Computer Science*, 67:283–301, 1989.

[BBP94] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. Technical Report P9314b, Programming Research Group, University of Amsterdam, 1994.

[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.

[Ber90] J.A. Bergstra. A process creation mechanism in process algebra. In *[Bae90]*, pages 81–88, 1990.

[BHK89a] 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.

[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In *[BHK89a]*, pages 1–66, 1989.

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

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

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

[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.

[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.

[Hen89] P.R.H. Hendriks. Lists and associative functions in algebraic specifications - semantics and implementation. Report CS-R8908, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1989.

[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.

[HK89a] J. Heering and P. Klint. PICO revisited. In *[BHK89a]*, pages 359–379, 1989.

[HK89b] J. Heering and P. Klint. The syntax definition formalism SDF. In *[BHK89a]*, pages 283–297, 1989.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in: *SIGPLAN Notices*, 24(7):179-191, 1989.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.

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

[KB93] J.W.C. Koorn and H.C.N. Bakker. Building an editor from existing components: an exercise in software re-use. Technical Report P9312, Programming Research Group, Univeristy of Amsterdam, 1993.

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

[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.

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

[Ous94] J. K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[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.

[Vaa90] F.W. Vaandrager. Process algebra semantics of POOL. In *[Bae90]*, pages 173–236, 1990.

[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.

[vVVvW94] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. Control and data transfer in the distributed editor of the ASF+SDF meta-environment. Technical report, Programming Research Group, Univeristy of Amsterdam, 1994. (in preparation).

[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.

A Relevant Process Algebra Axioms

Basic Process Algebra (BPA)

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

Deadlock (BPA$_\delta$)

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

Free merge operator

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

Merge operator

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

Encapsulation operator

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

Renaming operator

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

Process creation operator

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

State operator

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

Iteration operator

$x * y$	$= x.(x * y) + y$

Conditional control

$T :\rightarrow x$	$= x$
$F :\rightarrow x$	$= \delta$

Notes:

- PA consists of BPA plus free merge operator.
- ACP consists of BPA plus deadlock, merge and encapsulation.

B Interpretation of the calculator script

We illustrate the effect of executing the ToolBus interpreter as specified in Section 5 for the calculator script described in Section 6.

Initial term

```
interpret(
[    event(ui(1), button(showLog))
,    value(log,history(pair(plus(2,3), 5), pair(times(3,7),21)))
,    event(ui(1), button(calc))
,    event(ui(2), button(showTime))
,    value(batch, expr(times(3,4)))
,    value(clock, time("12h34m"))
,    value(calc,12)
,    value(ui(1), expr(plus(1,2)))
,    value(calc,3)
],
toolbus(CALC, UI1, UI2, LOG, BATCH, CLOCK)
)
```

Trace of events and final state of the ToolBus

```
[ eval(batch,fromFile),
  event(ui ( 1 ),
        button ( showLog )),
  eval(log,readLog),
  value(log,
        history ( pair ( plus ( 2,3 ),
                         5 ),
                  pair ( times ( 3,7 ),
                         21 ) )),
  do(ui ( 1 ),
     displayLog ( history ( pair ( plus ( 2,3 ),
                                   5 ),
                            pair ( times ( 3,7 ),
                                   21 ) ) )),
  ack-event(ui ( 1 )),
  event(ui ( 1 ),
        button ( calc )),
  eval(ui ( 1 ),
       dialogGetExpr),
  event(ui ( 2 ),
        button ( showTime )),
  eval(clock,readTime),
  value(batch,
        expr ( times ( 3,4 ) )),
  eval(calc,
       times ( 3,4 )),
  value(clock,
        time ( "12h34m" )),
  do(ui ( 2 ),
     displayTime ( time ( "12h34m" ) )),
  ack-event(ui ( 2 )),
  value(calc,12),
  do(log,
     writelog ( times ( 3,4 ),
                12 )),
  do(batch,
     toFile ( V $ BATCH )),
  eval(batch,fromFile),
  value(ui ( 1 ),
        expr ( plus ( 1,2 ) )),
  eval(calc,
       plus ( 1,2 )),
  value(calc,3),
  do(log,
```

```
   writelog ( plus ( 1,2 ),
              3 )),
do(ui ( 1 ),
   displayValue ( 3 )),
ack-event(ui ( 1 )) ]
#
{
  <
    rec-note ( calc,
               Exp $ LOG,
               Val $ LOG )
      ;
      snd-do ( log,
               writelog ( Exp $ LOG,
                          Val $ LOG ) )
        .
        (
          rec-note ( calc,
                     Exp $ LOG,
                     Val $ LOG ) . snd-do ( log,
                                            writelog ( Exp $ LOG,
                                                       Val $ LOG ) )
            +
            rec-msg ( showLog )
              .
              snd-eval ( log,readLog )
                .
                rec-value ( log,
                            Log $ LOG ) . snd-msg ( showLog,
                                                    Log $ LOG )
          ) * delta,
    rec-msg ( showLog )
      ;
      (
        snd-eval ( log,readLog )
          .
          rec-value ( log,
                       Log $ LOG ) . snd-msg ( showLog,
                                               Log $ LOG )
        )
        .
        (
          rec-note ( calc,
                     Exp $ LOG,
                     Val $ LOG ) . snd-do ( log,
                                            writelog ( Exp $ LOG,
                                                       Val $ LOG ) )
            +
            rec-msg ( showLog )
              .
              snd-eval ( log,readLog )
                .
                rec-value ( log,
                            Log $ LOG ) . snd-msg ( showLog,
                                                    Log $ LOG )
          ) * delta
    > : [ ( Log $ LOG : history ( pair ( plus ( 2,3 ),
                                         5 ),
                                  pair ( times ( 3,7 ),
                                         21 ) ) ),
          ( Exp $ LOG : plus ( 1,2 ) ),
          ( Val $ LOG : 3 ) ] : [ calc ] : [  ],
  <
    rec-msg ( showTime )
      ;
      (
        snd-eval ( clock,readTime )
```

```
        .
        rec-value ( clock,
                    Time $ CLOCK ) . snd-msg ( showTime,
                                              Time $ CLOCK )
      )
      .
      (
        rec-msg ( showTime )
          .
          snd-eval ( clock,readTime )
            .
            rec-value ( clock,
                        Time $ CLOCK ) . snd-msg ( showTime,
                                                  Time $ CLOCK )
        ) * delta
  > : [ ( Time $ CLOCK : time ( "12h34m" ) ) ] : [  ] : [  ],
<
  rec-event ( ui ( U $ UI2 ),
              button ( showTime ) )
    ;
    (
      snd-msg ( showTime )
        .
        rec-msg ( showTime,
                  Time $ UI2 )
          .
          snd-do ( ui ( U $ UI2 ),
                   displayTime ( Time $ UI2 ) )
            .
            snd-ack-event ( ui ( U $ UI2 ) )
      )
      .
      (
        rec-event ( ui ( U $ UI2 ),
                    button ( showTime ) )
          .
          snd-msg ( showTime )
            .
            rec-msg ( showTime,
                      Time $ UI2 )
              .
              snd-do ( ui ( U $ UI2 ),
                       displayTime ( Time $ UI2 ) )
                .
                snd-ack-event ( ui ( U $ UI2 ) )
        ) * delta
  > : [ ( U $ UI2 : 2 ),
        ( Time $ UI2 : time ( "12h34m" ) ) ] : [  ] : [  ],
<
  rec-value ( batch,
              expr ( Exp $ BATCH ) )
    ;
    (
      snd-msg ( calc,
                Exp $ BATCH )
        .
        rec-msg ( calc,
                  Exp $ BATCH,
                  Val $ BATCH ) . snd-do ( batch,
                                           toFile ( V $ BATCH ) )
      )
      .
      (
        snd-eval ( batch,fromFile )
          .
          rec-value ( batch,
                      expr ( Exp $ BATCH ) )
```

```
            .
            snd-msg ( calc,
                      Exp $ BATCH )
              .
              rec-msg ( calc,
                        Exp $ BATCH,
                        Val $ BATCH ) . snd-do ( batch,
                                                 toFile ( V $ BATCH ) )
      ) * delta
  > : [ ( Exp $ BATCH : times ( 3,4 ) ),
      ( Val $ BATCH : 12 ) ] : [  ] : [  ],
<
  rec-event ( ui ( U $ UI1 ),
              button ( calc ) )
    ;
    (
      snd-eval ( ui ( U $ UI1 ),
                 dialogGetExpr )
        .
        (
          rec-value ( ui ( U $ UI1 ),
                      cancel )
            +
            rec-value ( ui ( U $ UI1 ),
                        expr ( Exp $ UI1 ) )
              .
              snd-msg ( calc,
                        Exp $ UI1 )
                .
                rec-msg ( calc,
                          Exp $ UI1,
                          Val $ UI1 )
                  .
                  snd-do ( ui ( U $ UI1 ),
                           displayValue ( Val $ UI1 ) )
          ) . snd-ack-event ( ui ( U $ UI1 ) )
      )
      .
      (
        rec-event ( ui ( U $ UI1 ),
                    button ( calc ) )
          .
          snd-eval ( ui ( U $ UI1 ),
                     dialogGetExpr )
            .
            (
              rec-value ( ui ( U $ UI1 ),
                          cancel )
                +
                rec-value ( ui ( U $ UI1 ),
                            expr ( Exp $ UI1 ) )
                  .
                  snd-msg ( calc,
                            Exp $ UI1 )
                    .
                    rec-msg ( calc,
                              Exp $ UI1,
                              Val $ UI1 )
                      .
                      snd-do ( ui ( U $ UI1 ),
                               displayValue ( Val $ UI1 ) )
              ) . snd-ack-event ( ui ( U $ UI1 ) )
          +
          rec-event ( ui ( U $ UI1 ),
                      button ( showLog ) )
            .
            snd-msg ( showLog )
```

```
              .
              rec-msg ( showLog,
                        Log $ UI1 )
                .
                snd-do ( ui ( U $ UI1 ),
                         displayLog ( Log $ UI1 ) )
                  .
                  snd-ack-event ( ui ( U $ UI1 ) )
      ) * delta,
rec-event ( ui ( U $ UI1 ),
            button ( showLog ) )
  ;
  (
    snd-msg ( showLog )
      .
      rec-msg ( showLog,
                Log $ UI1 )
        .
        snd-do ( ui ( U $ UI1 ),
                 displayLog ( Log $ UI1 ) )
          .
          snd-ack-event ( ui ( U $ UI1 ) )
    )
    .
    (
      rec-event ( ui ( U $ UI1 ),
                  button ( calc ) )
        .
        snd-eval ( ui ( U $ UI1 ),
                   dialogGetExpr )
          .
          (
            rec-value ( ui ( U $ UI1 ),
                        cancel )
              +
              rec-value ( ui ( U $ UI1 ),
                          expr ( Exp $ UI1 ) )
                .
                snd-msg ( calc,
                          Exp $ UI1 )
                  .
                  rec-msg ( calc,
                            Exp $ UI1,
                            Val $ UI1 )
                    .
                    snd-do ( ui ( U $ UI1 ),
                             displayValue ( Val $ UI1 ) )
            ) . snd-ack-event ( ui ( U $ UI1 ) )
        +
        rec-event ( ui ( U $ UI1 ),
                    button ( showLog ) )
          .
          snd-msg ( showLog )
            .
            rec-msg ( showLog,
                      Log $ UI1 )
              .
              snd-do ( ui ( U $ UI1 ),
                       displayLog ( Log $ UI1 ) )
                .
                snd-ack-event ( ui ( U $ UI1 ) )
      ) * delta
> : [ ( U $ UI1 : 1 ),
      ( Log $ UI1 : history ( pair ( plus ( 2,3 ),
                                     5 ),
                              pair ( times ( 3,7 ),
                                     21 ) ) ),
```

```
        ( Exp $ UI1 : plus ( 1,2 ) ),
        ( Val $ UI1 : 3 ) ] : [  ] : [  ],
<
  rec-msg ( calc,
            Exp $ CALC )
    ;
    (
      snd-eval ( calc,
                 Exp $ CALC )
        .
        rec-value ( calc,
                    Val $ CALC )
          .
          snd-msg ( calc,
                    Exp $ CALC,
                    Val $ CALC ) . snd-note ( calc,
                                             Exp $ CALC,
                                             Val $ CALC )
      )
      .
      (
        rec-msg ( calc,
                  Exp $ CALC )
          .
          snd-eval ( calc,
                     Exp $ CALC )
            .
            rec-value ( calc,
                        Val $ CALC )
              .
              snd-msg ( calc,
                        Exp $ CALC,
                        Val $ CALC ) . snd-note ( calc,
                                                 Exp $ CALC,
                                                 Val $ CALC )
        ) * delta
  > : [ ( Exp $ CALC : plus ( 1,2 ) ),
        ( Val $ CALC : 3 ) ] : [  ] : [  ]
}
```

C The ToolBus in C

C.1 Overview

In this appendix we give a complete listing of the ToolBus implementation. Its consists of the following parts:

`toolbus.h` **(C.3)**: global definitions.

`terms.h`, `terms.c` **(C.4)**: the datatype of terms.

`env.h`, `env.c` **(C.5)**: environments.

`match.h`, `match.c` **(C.6)**: matching.

`procdef.h`, `procdef.c` **(C.7)**: the datatype of process expressions.

`interpreter.h`, `interpreter.c` **(C.8)**: interpreter for ToolBus scripts.

`sockets.h`, `sockets.c` `server.c` **(C.9)**: functions for creating and connecting sockets used for implementing a client/server communication structure. The server (i.e., ToolBus) part is given here. The client (i.e., tool) part is given in Section D.

`utils.h`, `utils.c` **(C.10)**: utilities for reading, parsing, printing, and matching.

`typecheck.h`, `typecheck.c` **(C.11)**: typechecking of ToolBus scripts.

`toolbus.c` **(C.12)**: The ToolBus proper: the representation of processes and tools, main program.

`script.l`, `script.y` **(C.13)**: the lexical and concrete syntax of ToolBus scripts.

C.2 Limitations

For completeness, we list here the current limitations of this implementation:

- Simplicity has been preferred over efficiency. As a consequence, there are ample opportunities for optimization and improvement:
  - the pure interpretation of process expressions can be replaced by a more compilational approach (e.g., generate some form of transition tables).
  - there are many linear searches that are candidate for optimization.
- There is no garbage collection of terms and process expressions in the interpreter.
- There is no garbage collection of terms in tools.
- There are fixed maxima on the number of processes, the number of tools, the number of symbols in a script, the number of alternative input channels per tool, and the length of terms for sending and receiving.
- There is no error detection or recovery in the case of tool failure, i.e., an infinite loop or a complete crash of the tool.
- No distinction is made between "Big Endians" and "Little Endians". As a consequence, this implementation will only work correctly if all host machines involved use the same byte order.

C.3 General Header

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <assert.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/param.h>

/* allocate object of type "tp" */
#define NEW(tp) ((tp *) malloc(sizeof(tp)))

typedef enum bool {false, true} bool;

extern bool ToolBus;

/* debugging */
extern bool debug;
#define DEBUG(stats) if(debug){stats}

extern void panic(const char *);
```

C.4 Terms

A specification of the datatype of terms has been given in Section 3.2.1.

C.4.1 terms.h

```
typedef enum tkind
{
  t_integer, t_string, t_var, t_form, t_appl
} tkind;

typedef struct term
{
```

```
  enum tkind        kind;
  char             *symbol;
  struct term_list *args;
} term;

typedef struct appl
{
  char             *id;
  struct term_list *args;
} appl;

typedef struct term_list
{
  term             *first;
  struct term_list *next;
} term_list;

term      *mk_term(const tkind, char *,
                   term_list *);
char      *resolve(const char *, const char *);
void       pr_term(const term *);

term_list *mk_term_list(term *, term_list *);
void       pr_term_list(const term_list *);
term_list *append_term_list(term_list *, term *);
int        length_term_list(term_list *);
term_list *cp_term_list(term_list *);

bool       streq(const char *, const char *);

typedef struct id_list
{
  char *first;
  struct id_list *next;
} id_list;

id_list   *mk_id_list(char *, id_list *);
void       pr_id_list(id_list *, const char *);
bool       elem(const char *, id_list *);
id_list   *delete(const char *, id_list *);
int        length_id_list(id_list *);
bool       subset(id_list *, id_list *);

extern void print(const char *);
```

C.4.2 terms.c

```
#include "toolbus.h"
#include "terms.h"

bool streq(const char *s1, const char *s2)
{
  return s1 && s2 && (strcmp(s1, s2) == 0);
}

term *mk_term(const tkind k, char *s,
              term_list *args)
{
  term *t = NEW(term);

  t->kind = k;
  t->symbol = s;
  t->args = args;
  switch(k){
  case t_string: case t_integer:  case t_var:
  case t_form:
    assert(args == NULL);
    return t;
  case t_appl:
    return t;
  default:
    panic("mk_term");
  }
  return NULL;
}

char *resolve(const char *id, const char *pname)
{
  char cbuf[256];
  sprintf(cbuf, "%s$%s", id, pname);
  return strdup(cbuf);
}

void pr_term(const term *t)
{
  switch(t->kind){
  case t_integer:
  case t_var:  case t_form:
    print(t->symbol); break;
  case t_string:
    { char cbuf[64];

      sprintf(cbuf, "%d", strlen(t->symbol));
      print("\""); print(cbuf); print(":");
      print(t->symbol); print("\"");
      break;
    }
  case t_appl:
    print(t->symbol);
    if(t->args != NULL){
      print("(");
      pr_term_list(t->args);
      print(")");
    }
    break;
  default: panic("pr_term");
  }
}

/*--- term_list --------------------------------*/

term_list *mk_term_list(term *first,
                        term_list *next)
{
  term_list *tl = NEW(term_list);

  tl->first = first;
  tl->next = next;
  return tl;
}

void pr_term_list(const term_list *tl)
{
  bool first = true;

  for( ; tl; tl = tl->next){
    if(first)
      first = false;
    else
      print(",");
    pr_term(tl->first);
  }
}
```

```
int length_term_list(term_list *tl)
{
  int n = 0;
  term_list *first;

  for(first = tl; first; first = first->next)
    n++;
  return n;
}

term_list *append_term_list(term_list *tl,
                            term *t)
{ term_list *last = tl, *tmp = tl;

  if(!tl)
    return mk_term_list(t, NULL);
  for( ; tmp; tmp = tmp->next){
    if(tmp->next != NULL)
      last = tmp->next;
  }
  last->next = mk_term_list(t, NULL);
  return tl;
}

term_list *cp_term_list(term_list *tl)
{
  term_list head, *prev = &head;

  head.next = NULL;
  for( ; tl ; tl = tl->next){
    prev->next = NEW(term_list);
    prev = prev->next;
    prev->next = NULL;
    prev->first = tl->first;
  }
  return head.next;
}

/*--- id_list ----------------------------------*/

id_list *mk_id_list(char *first, id_list *next)
{
  id_list *idl = NEW(id_list);

  idl->first = first;
  idl->next = next;
  return idl;
}

void pr_id_list(id_list *idl, const char *sep)
{
  bool first = true;

  fprintf(stderr, "[ ");
  for( ; idl; idl = idl->next){
    if(first)
      first = false;
    else
      fprintf(stderr, "%s", sep);
    fprintf(stderr, "%s", idl->first);
  }
  fprintf(stderr," ]");
}

bool elem(const char *id, id_list *ids)
{
  for( ; ids; ids = ids->next){
    if(streq(id, ids->first))
      return true;
  }
  return false;
}

id_list *delete(const char *id, id_list *ids)
{
  id_list head, *prev;

  prev = &head;
  for(head.next = ids; ids; ids = ids->next){
    if(streq(id, ids->first)){
      prev->next = ids->next;
      return head.next;
    }
    prev = ids;
  }
  return head.next;
}

int length_id_list(id_list *ids)
{
  int n = 0;

  for( ; ids; ids = ids->next)
    n++;
  return n;
}

bool subset(id_list *sub, id_list *sup)
{
  for( ; sub; sub = sub->next)
    if(!elem(sub->first, sup))
      return false;
  return true;
}
```

C.5 Environments

A specification of the datatype of environments has been given in Section 5.2.

C.5.1 env.h

```
typedef struct env
{
  char       *id;      /* the name */
  term       *value;   /* its value */
  struct env *next;    /* next entry */
} env;

env  *mk_env(char *, term *, env *);
void  pr_env(env *);
env  *assign(char *, term *, env *);
env  *update(env *, env *);
term *value(const char *, env *);
env  *create_env(id_list *, const char *,
                 term_list *, env *);

extern env *empty_env;
```

C.5.2 env.c

```
#include "toolbus.h"
#include "terms.h"
```

```
#include "env.h"

env * empty_env = NULL;

env *mk_env(char *id, term *t, env *e)
{
  env *nwb = NEW(env);
  nwb->id = id;
  nwb->value = t;
  nwb->next = e;
  return nwb;
}

void pr_env(env *e)
{ fprintf(stderr, "{");
  for( ; e; e = e->next){
    fprintf(stderr, "(%s,", e->id);
    pr_term(e->value);
    fprintf(stderr, ")");
  }
  fprintf(stderr, "}");
}

env *assign(char *id, term *t, env *e)
{
  env *e1;

  for(e1 = e; e1; e1 = e1->next){
    if(streq(id, e1->id)){
      e1->value = t;
      return e;
    }
  }
  return mk_env(id, t, e);
}

env *update(env *u, env *e)
{
  for( ; u; u = u->next)
    e = assign(u->id, u->value, e);
  return e;
}

term *value(const char *id, env *e)
{
  env *e1;

  for(e1 = e; e1; e1 = e1->next){
    if(streq(id, e1->id)){
      return e1->value;
    }
  }
  printf("identifier %s not bound", id);
  pr_env(e);
  printf("\n");
  panic("value");
  return NULL;
}

term *substitute(term *, env *);

env *create_env(id_list *ids, const char *pname,
                term_list *terms, env *parent_env)
{
  env *e = empty_env;

  for( ; ids ; ids = ids->next ){
    if(!terms){
      printf("Too few actual parameters "
             "for process %s\n", pname);
      panic("create_env (1)");
    }
    e = mk_env(ids->first,
               substitute(terms->first,
                          parent_env), e);
    terms = terms->next;
  }
  if(terms){
    printf("Too many actual parameters "
           "for process %s\n", pname);
    panic("create_env (2)");
  }
  return e;
}
```

C.6 Matching

A specification of matching and substitution has been given in Section 5.3.

C.6.1 `match.h`

```
extern env *nomatch;

term      *substitute(term *, env *, env *);
term_list *substitute_list(term_list *,
                           env *, env *);
env       *match(term *, term *, env *, env *,
                 env *, env *, env *);
env       *match_list(term_list *, term_list *,
                      env *, env *, env *,
                      env *, env *);
```

C.6.2 `match.c`

```
#include "toolbus.h"
#include "terms.h"
#include "env.h"
#include "procdef.h"
#include "match.h"
#include "utils.h"

env *nomatch = (env *) 314;

/*--- substitution -----------------------------*/

term *substitute(term *t,
                 env *form_env, env *loc_env)
{
  switch(t->kind){
  case t_integer: case t_string:
    return t;
  case t_var:
    return value(t->symbol, loc_env);
  case t_form:
    return value(t->symbol, form_env);
  case t_appl:
      return
        mk_term(t_appl,
                t->symbol,
```

```
                substitute_list(t->args,
                                form_env,
                                loc_env));
  default: panic("subst");
  }
  return NULL;
}

term_list *substitute_list(term_list *tl,
                           env *form_env,
                           env * loc_env)
{
  term_list *tmp, *res;

  for(res=tmp=cp_term_list(tl);
      tmp;
      tmp = tmp->next){
    tmp->first = substitute(tmp->first,
                            form_env, loc_env);
  }
  return res;
}

env *match(term *t1,                 /* pattern */
           term *t2,   /* subject to be matched */
           env *f1,   /* formal env for pattern */
           env *l1,    /* local env for pattern */
           env *f2,   /* formal env for subject */
           env *l2,    /* local env for subject */
           env *bindings) /* variable bindings  */
{
  if(!(t1->kind == t_var || t1->kind == t_form
      || t1->kind == t2->kind))
    return nomatch;

  switch(t1->kind){
  case t_integer:
  case t_string:
    if(streq(t1->symbol, t2->symbol))
      return bindings;
    else
      return nomatch;
  case t_var:
    {
      env *nwb = mk_env(t1->symbol,
                        substitute(t2, f2, l2),
                        bindings);
      return nwb;
    }
  case t_form:
    return
      match(value(t1->symbol,f1),
            substitute(t2,f2, l2),
            f1, l1, f2, l2, bindings);
  case t_appl:
    { env *e1;

      if(!streq(t1->symbol, t2->symbol))
        return nomatch;
      e1 = match_list(t1->args,
                      t2->args,
                      f1, l1, f2, l2, bindings);
      if(e1 == nomatch)
        return nomatch;
      else
        return e1;
    }
  default: panic("match");
  }
  return NULL;
}

env *match_list(term_list *tl1, /* pattern list */
                term_list *tl2, /* subject list */
                env *f1,
                env *l1,
                env *f2,
                env *l2,
                env *bindings)
{
  while(tl1){
    if(tl2 == NULL)
      return nomatch;
    bindings = match(tl1->first, tl2->first,
                     f1, l1, f2, l2, bindings);
    if(bindings == nomatch)
      return nomatch;
    tl1 = tl1->next;
    tl2 = tl2->next;
  }
  return (tl2 != NULL) ? nomatch : bindings;
}
```

C.7 Process expressions

A specification of process expression was given in Section 3.2.2. This specification was extended in Section 5.4 where the notion of "action prefix form" was introduced. In the implementation we represent the sorts ATOM, PROC and AP-FORM by the structure `proc`.

C.7.1 procdef.h

```
typedef enum pkind
{
  a_snd_msg, a_rec_msg, a_snd_note, a_rec_note,
  a_no_note, a_subscribe, a_unsubscribe,
  a_snd_eval, a_snd_do, a_ack_event,
  a_rec_event, a_rec_value, a_delta, a_create,
  p_plus, p_dot, p_star, p_call, ap_semi, ap_plus
} pkind;

#define IS_ATOM(p) \
   ((p) && (a_snd_msg <= (p)->kind) && \
    ((p)->kind <= a_create))

#define IS_PROC(p) \
   ((p) && (p_plus <= (p)->kind) && \
    ((p)->kind <= p_call))

#define IN_PREFIX_FORM(p) \
   ((p)->kind == ap_semi || \
    (p)->kind == ap_plus || \
    (p)->kind == a_delta)

#define IN_NORMAL_FORM(p)\
   ((p) && ((p)->kind == ap_plus))

#define IS_DELTA(p)\
   ((p) && ((p)->kind == a_delta))
```

```
typedef struct proc
{
  pkind kind;
  union {
    struct proc_list *pargs;
    struct appl      *appl;
  } u;
} proc;

#define KIND(p)  (p->kind)
#define PARGS(p) (p->u.pargs)
#define PARG1(p) (p->u.pargs->first)
#define PARG2(p) (p->u.pargs->next->first)
#define ID(p)    (p->u.appl->id)
#define TARGS(p) (p->u.appl->args)

typedef struct proc_list
{
  proc             *first;
  struct proc_list *next;
} proc_list;

pkind     communicate(proc *);
proc      *mk_atom(pkind, term_list *);
proc      *mk_create(char *, term_list *);

proc_list *mk_proc_list(proc *, proc_list *);
void       pr_proc_list(proc_list *, char *);
proc_list *append_proc_list(proc_list *,
                            proc_list *);

proc      *mk_proc_op(pkind, proc *, proc *);
proc      *mk_proc_call(char *, term_list *);

void       pr_proc(proc *);
proc_list *cp_proc_list(proc_list *);
proc      *cp_proc(proc *);

typedef struct proc_def
{
  char    *name;
  id_list *formals;
  proc    *proc;
} proc_def;

proc_def  *mk_proc_def(char *, id_list *,
                       proc *);
void       pr_proc_def(proc_def *);

typedef struct proc_def_list
{
  proc_def             *first;
  struct proc_def_list *next;
} proc_def_list;

proc_def_list
          *mk_proc_def_list(proc_def *,
                            proc_def_list *);
void       pr_proc_def_list(proc_def_list *);
proc_def  *definition(const char *,
                      proc_def_list *);

env       *match_atom(proc *, proc *,
                      env *, env *, env *, env *);
env       *match_note(proc *, term *, env *,
                      env *);
```

C.7.2 procdef.c

```
#include "toolbus.h"
#include "terms.h"
#include "env.h"
#include "match.h"
#include "procdef.h"

/*--- atoms ------------------------------------*/

pkind communicate(proc *at)
{
  switch(at->kind)
    {
    case a_rec_msg: return(a_snd_msg);
    case a_rec_note: return(a_snd_note);
    default:
      panic("communicate");
    }
  return NULL;
}

proc *mk_atom(pkind k, term_list *args)
{
  proc *a = NEW(proc);
  appl *ap = NEW(appl);

  ap->id = "*unused*";
  ap->args = args;
  a->kind = k;
  a->u.appl = ap;
  return a;
}

proc *mk_create(char *proc_name, term_list *args)
{
  proc *a = mk_atom(a_create, args);
  a->u.appl->id = proc_name;
  return a;
}

void pr_atom(proc *a)
{
  char *s = "** error **";

  assert(IS_ATOM(a));
  switch(a->kind){
  case a_snd_msg: s = "snd-msg"; break;
  case a_rec_msg: s = "rec-msg"; break;
  case a_snd_note: s = "snd-note"; break;
  case a_rec_note: s = "rec-note"; break;
  case a_no_note: s = "no-note"; break;
  case a_rec_event: s = "rec-event"; break;
  case a_ack_event: s = "ack-event"; break;
  case a_snd_eval: s = "snd-eval"; break;
  case a_snd_do: s = "snd-do"; break;
  case a_rec_value: s = "rec-value"; break;
  case a_delta: s = "delta"; break;
  case a_create: s = "create"; break;
  case a_subscribe: s = "subscribe"; break;
  case a_unsubscribe: s = "unsubscribe"; break;
  default:
    panic("pr_atom");
  }
  fprintf(stderr, "%s", s);
  if(a->kind == a_create){
    printf(" %s", ID(a));
  }
```

```
  if(TARGS(a)){
    fprintf(stderr, "(");
    pr_term_list(TARGS(a));
    fprintf(stderr, ")");
  }
}

/*--- proc_list --------------------------------*/

proc_list *mk_proc_list(proc *first,
                        proc_list *next)
{
  proc_list *pl = NEW(proc_list);

  pl->first = first;
  pl->next = next;
  return pl;
}

void pr_proc_list(proc_list *pl, char *sep)
{
  bool first = true;

  for( ; pl; pl = pl->next){
    if(first)
      first = false;
    else
      fprintf(stderr, "%s", sep);
    pr_proc(pl->first);
  }
}

proc_list *append_proc_list(proc_list *pl1,
                            proc_list *pl2)
{
  proc_list *last, *tmp;

  pl1 = cp_proc_list(pl1);
  last = tmp = pl1;

  for( ; tmp; tmp = tmp->next){
    if(tmp->next != NULL)
      last = tmp->next;
  }
  last->next = pl2;
  return pl1;
}

proc_list *cp_proc_list(proc_list *pl)
{
  proc_list head, *prev = &head;

  head.next = NULL;
  for( ; pl; pl = pl->next){
    prev->next = NEW(proc_list);
    prev = prev->next;
    prev->next = NULL;
    prev->first = pl->first;
  }
  return head.next;
}

proc *lift(proc *p)             /* atom => atom; */
{
  if(IS_ATOM(p)){
    proc *nwp = NEW(proc); KIND(nwp) = ap_semi;
    PARGS(nwp) = mk_proc_list(p, NULL);
    return nwp;
  } else
    return p;
}

proc *mk_proc_op(pkind p_op, proc *p, proc *q)
{
  proc *r;
  proc_list *pargs;

  DEBUG(
    fprintf(stderr, "\nmk_proc_op(%d)\n\t", p_op);
    pr_proc(p); fprintf(stderr, "\n\t");
    pr_proc(q); fprintf(stderr, "\n");
    fflush(stderr);)

  switch(p_op){
  case p_dot:
    if(IN_PREFIX_FORM(p))
      goto ap_semi;
    goto p_star;
  case p_plus:
    if(IN_PREFIX_FORM(p) && IN_PREFIX_FORM(q))
      goto ap_plus;
    goto p_star;
  case p_star:
  p_star:
    r = NEW(proc); KIND(r) = p_op;
    PARGS(r) =
      mk_proc_list(p, mk_proc_list(q, NULL));
    DEBUG(pr_proc(r);) return r;

  case ap_semi:
  ap_semi:
    if(KIND(p) == a_delta)
      /* delta ; q = delta */
      return p;
    r = NEW(proc); KIND(r) = ap_semi;
    if(IS_ATOM(p) || KIND(p) == a_delta){
      switch(KIND(q)){
      case ap_semi:
        /* p ; (q1;...;qn) = (p;q1;...;qn) */
        PARGS(r) = mk_proc_list(p, PARGS(q));
        DEBUG(pr_proc(r);) return r;
      default:
        /* p ; q */
        PARGS(r) =
          mk_proc_list(p, mk_proc_list(q, NULL));
        DEBUG(pr_proc(r);) return r;
      }
    }
    if(KIND(p) == ap_semi){
       switch(KIND(q)){
       case ap_semi:
         /* (p1;...;pn) ; (q1;...;qm) =
              (p1;...;pn;q1;...;qm) */
         PARGS(r) =
           append_proc_list(PARGS(p), PARGS(q));
         DEBUG(pr_proc(r);) return r;
       default:
         /* (p1;...pn) ; q = (p1;...pn;q) */
         PARGS(r) =
         append_proc_list(PARGS(p),
                          mk_proc_list(q, NULL));
         DEBUG(pr_proc(r);) return r;
       }
    }
```

```
    if(KIND(p) == ap_plus){
      /* <p1,...,pn>;q = <p1;q,...,pn;q> */
      KIND(r) = ap_plus;
      PARGS(r) =  cp_proc_list(PARGS(p));
      for(pargs = PARGS(r);
          pargs;
          pargs = pargs->next){
        pargs->first =
          mk_proc_op(ap_semi, pargs->first, q);
      }
      DEBUG(pr_proc(r);) return r;
    }
    panic("mk_proc_op: case ap_semi");

  case ap_plus:
  ap_plus:
    if(KIND(p) == a_delta)
      /* <delta, q> = q */
      return q;
    if(KIND(q) == a_delta)
      /* <p, delta> = p */
      return p;
    r = NEW(proc); KIND(r) = ap_plus;
    if(KIND(p) == ap_plus)
      if(KIND(q) == ap_plus)
        /* <<p11,...,p1n>,<q1,...,qm>> =
           <p11,...,p1n,q1,...,qm> */
        PARGS(r) =
          append_proc_list(PARGS(p), PARGS(q));
      else {
        /* <<p11,...,p1n>,q> =  <p11,...,p1n,q> */
        q = lift(q);
        assert(IN_PREFIX_FORM(q));
        PARGS(r) =
          append_proc_list(PARGS(p),
                           mk_proc_list(q, NULL));
      }
    else
    if(KIND(q) == ap_plus){
      /* <p1, <q1,...qn>> =  <p1, q1,...qn> */
      p = lift(p);
      assert(IN_PREFIX_FORM(p));
      PARGS(r) =
        append_proc_list(mk_proc_list(p, NULL),
                         PARGS(q));
    } else {
      /* <p, q> */
      p = lift(p); assert(IN_PREFIX_FORM(p));
      q = lift(q); assert(IN_PREFIX_FORM(q));
      PARGS(r) =
        mk_proc_list(p, mk_proc_list(q, NULL));
    }
    DEBUG(pr_proc(r);) return r;

  default:
    panic("mk_proc_op");
  }
  return NULL;
}

proc *mk_proc_call(char *id, term_list *args)
{ proc *p = NEW(proc);
  appl *ap = NEW(appl);

  ap->id = id;
  ap->args = args;
  p->u.appl = ap;
  p->kind = p_call;
  p->u.appl = ap;
  return p;
}

void pr_proc(proc *p)
{
  char *s, *lp = "(", *rp = ")";

  if(!p){
    fprintf(stderr, "<< null proc >>"); return;
  }

  switch(KIND(p)){
  case p_plus:
    s ="+"; goto star;
  case ap_plus:
    s = ","; lp = "<"; rp = ">"; goto star;
  case ap_semi:
    s = ";"; goto star;
  case p_dot:
    s ="."; goto star;
  case p_star:
    s = "*";
  star:
    fprintf(stderr, "%s", lp);
    pr_proc_list(PARGS(p), s);
    fprintf(stderr, "%s", rp);
    return;
  case p_call:
    fprintf(stderr, "%s", ID(p));
    if(TARGS(p)){
      fprintf(stderr, "(");
      pr_term_list(TARGS(p));
      fprintf(stderr, ")");
    }
    return;
  default:
    if(IS_ATOM(p))
      pr_atom(p);
    else
      panic("pr_proc");
  }
}

proc *cp_proc(proc *p)
{
  switch(KIND(p)){
  case p_call:
    return p;
  case p_plus: case p_dot: case p_star:
  case ap_plus: case ap_semi:
    { proc *newp = NEW(proc);
      KIND(newp) = KIND(p);
      PARGS(newp) = cp_proc_list(PARGS(p));
      return newp;
    }
  default:
    if(IS_ATOM(p))
      return p;
    else
    panic("cp_proc");
  }
  return NULL;
}
```

```
/*--- process definitions ----------------------*/

proc_def *mk_proc_def(char *name,
                      id_list *formals,
                       proc *p)
{
  proc_def *pd = NEW(proc_def);
  id_list *f;

  pd->name = name;
  for(f = formals ; f; f = f->next) {
    char buf[256];
    sprintf(buf, "%s$%s", f->first, name);
    f->first = strdup(buf);
  }
  pd->formals = formals;
  pd->proc = p;
  return pd;
}

void pr_proc_def(proc_def *pd)
{
  fprintf(stderr, "\nprocess %s =  \n", pd->name);
  pr_proc(pd->proc);
  fprintf(stderr, "\n");
}

proc_def_list *
   mk_proc_def_list(proc_def *first,
                    proc_def_list *next)
{
  proc_def_list *pdl = NEW(proc_def_list);

  pdl->first = first;
  pdl->next = next;
  return pdl;
}

void pr_proc_def_list(proc_def_list *pdl)
{
  for( ; pdl; pdl = pdl->next){
    pr_proc_def(pdl->first);
  }
}

proc_def *definition(const char *name,
                     proc_def_list *pdl)
{
  for( ; pdl; pdl = pdl->next){
    if(streq(pdl->first->name, name))
      return pdl->first;
  }
  return NULL;
}

/*--- matching --------------------------------*/

env *match_atom(proc *a1,            /* pattern */
                proc *a2,            /* subject */
                env *f1,  /* formal env pattern */
                env *l1,   /* local env pattern */
                env *f2,  /* formal env subject */
                env *l2)   /* local env subject */
{ pkind k1, k2;

  assert(IS_ATOM(a1) && IS_ATOM(a2));
  k1 = a1->kind;
  k2 = a2->kind;
  if((k1 == a_rec_msg) && (k2 == a_snd_msg))
    /* ok */;
  else
    return nomatch;
  return
    match_list(TARGS(a1), TARGS(a2),
               f1, l1, f2, l2, NULL);
}

env *match_note(proc *a1,            /* pattern */
                term *evnt,          /* subject */
                env *f1,  /* formal env pattern */
                env *l1)   /* local env pattern */
{
  assert(IS_ATOM(a1));
  return
    match_list(TARGS(a1), evnt->args,
               f1, l1, f1, l1, NULL);
}
```

C.8 Interpreter

A specification of the TOOLBUS interpreter was given in Section 5.5. Observe that the functions

- `expand`,
- `msg_steps`,
- `note_steps`,
- `distr_note`,
- `atomic_steps`,
- `rec_from_tool_step`, and
- `all_internal_steps`

are direct implementations of the functions with a similar name appearing in the specification.

C.8.1 interpreter.h

```
extern proc_def_list *definitions;
                          /* list of all process
                             definitions       */
#define MAXPROC 100       /* max. # of processes
                             in toolbus        */
extern int nproc;         /* process counter <=
                             MAXPROC           */

typedef struct proc_repr
{
  proc      *proc;         /* process expression
                             defining this
                             process           */
  char      *name;        /* its name          */
  env       *formals;     /* bindings for formal
                             parameters        */
  env       *locals;      /* bindings for local
                             variables         */
  id_list   *subscriptions;
```

```
                             /* note subscriptions */
  term_list *notes;
                             /* queue of notes
                                awaiting treatment */
} proc_repr;

extern proc_repr toolbus[];

proc *expand(proc *, int);
proc *has_atom(int, bool (*)(proc *));
proc *next(proc *, int);
env  *find_msg(int, proc **, int *, proc **);
bool  msg_steps(void);
env  *has_matching_note(int,proc **,term_list **);
bool  note_steps(void);
void  distr_note(int, proc *);
bool  atomic_steps(void);
bool  rec_from_tool_step(term *);
void  all_internal_steps(void);

extern bool write_to_tool(char *, term *);
extern void create_process(char *, term_list *);
extern void pr_toolbus(void);
extern find_tool_index(char *);
extern bool connect_tool(char *, char *host,
                         int, int);
```

C.8.2 interpreter.c

```
#include "toolbus.h"
#include "terms.h"
#include "env.h"
#include "procdef.h"
#include "match.h"
#include "utils.h"
#include "interpreter.h"

/*--- expand  ----------------------------------*/

proc *expand(proc *p, int proc_id)
{
  proc *p1, *p2;

  DEBUG(printf("expand: (kind = %d)\n", p->kind);
        pr_proc(p); printf("\n"); fflush(stdout);)

  switch(KIND(p)){
  case p_plus:
    /* exp(p1 + p2) = <exp(p1), exp(p2)> */
    p1 = PARG1(p); p2 = PARG2(p);
    return
      mk_proc_op(ap_plus,
                 expand(p1, proc_id),
                 expand(p2, proc_id));

  case p_dot:
    p1 = PARG1(p); p2 = PARG2(p);
    return
      mk_proc_op(ap_semi,
                 expand(p1, proc_id), p2);

  case p_star:
    /* exp(p1*p2) = <exp(p1);(p1*p2), exp(p2)> */
    p1 = PARG1(p); p2 = PARG2(p);
    return
      mk_proc_op(ap_plus,
                 mk_proc_op(ap_semi,
                            expand(p1,proc_id),p),
                 expand(p2, proc_id));
  case p_call:
    {
      proc_def *pdef;
      char *pname = ID(p);
      term_list *actuals = TARGS(p);

      pdef = definition(pname, definitions);
      if(!pdef)
        panic("expand -- undefined process");

      actuals =
        substitute_list(actuals,
                        toolbus[proc_id].formals,
                        toolbus[proc_id].locals);
      toolbus[proc_id].formals =
        update(
          create_env(pdef->formals,
                     pname,
                     actuals,
                     toolbus[proc_id].formals),
          toolbus[proc_id].formals);
      return expand(pdef->proc, proc_id);
    }
  case a_delta:
    return p;
  default:
    if(IS_ATOM(p))
      return p;
    else {
      pr_proc(p);
      panic("expand");
    }
  }
  return NULL;
}

/*--- has_atom --------------------------------*/

proc *has_atom(int i, bool (* test)(proc *))
{
  proc *p = toolbus[i].proc;
  proc_list *pargs;

  assert(KIND(p) == ap_plus);

  for(pargs = PARGS(p);
      pargs;
      pargs = pargs->next){
    proc *at, *dt = pargs->first;
    if(IS_DELTA(dt))
      continue;
    assert(KIND(dt) == ap_semi);
    at = PARGS(dt)->first;
    if((*test)(at)){
      return dt;
    }
  }
  return NULL;
}

bool tst_snd_note(proc *at)
{
  if(IS_DELTA(at))
    return false;
  assert(IS_ATOM(at));
```

```
  return KIND(at) == a_snd_note;
}

bool tst_rec_msg(proc *at)
{
  if(IS_DELTA(at))
    return false;
  assert(IS_ATOM(at));
  return
    (KIND(at) == a_rec_msg);
}

bool tst_eval_do_ack(proc *at)
{
  return
    (KIND(at)==a_snd_eval) ||
      (KIND(at)==a_snd_do) ||
        (KIND(at)==a_ack_event);
}

bool tst_create(proc *at)
{
  return KIND(at)==a_create;
}

bool tst_no_note(proc *at)
{
  return KIND(at)==a_no_note;
}

bool tst_subs_unsubs(proc *at)
{
  return
    (KIND(at)==a_subscribe) ||
      (KIND(at)==a_unsubscribe);
}

/*--- next -------------------------------------*/
proc *next(proc *p, int proc_id)
{
  proc * nxt;
  proc_list *pargs;

  assert(IN_PREFIX_FORM(p));

  DEBUG(printf("next: "); pr_proc(p);
        printf("\n"); fflush(stdout);)
  switch(KIND(p)){
  case ap_semi:
    if(!PARGS(p)->next)
      panic("next");
    pargs = PARGS(p)->next;
    if(pargs->next){
      proc *ef = expand(pargs->first, proc_id);
      PARGS(p) = pargs->next;
      nxt = mk_proc_op(ap_semi, ef, p);
    } else {
      nxt = expand(pargs->first, proc_id);
    }
    assert(IN_PREFIX_FORM(nxt));
    if(KIND(nxt) != ap_plus){
      proc * nxt1 = NEW(proc);
      KIND(nxt1) = ap_plus;
      PARGS(nxt1) = mk_proc_list(nxt, NULL);
      DEBUG(printf("next returns:\n");
            fflush(stdout); pr_proc(nxt1);)
      return nxt1;
    } else
      return nxt;
  default:
    panic("next (2)");
  }
  return NULL;
}

/*--- find_msg --------------------------------*/

#define CHANGE change = gchange = true;

env *find_msg(
    int rec_i,          /* index of receiving
                           process              */
    proc **the_rec_p,   /* pointer to its process
                           expression           */
    int *snd_i,         /* pointer to index of
                           sending process      */
    proc **the_snd_p)   /* pointer to its process
                           expression           */
{
  proc *receiver = toolbus[rec_i].proc;
  proc_list *rec_args;

  DEBUG(printf("find_msg(%d)\n", rec_i);
        pr_proc(receiver);
        printf("\n"); fflush(stdout);)

  assert(IN_NORMAL_FORM(receiver));

  for(rec_args = PARGS(receiver);
      rec_args;
      rec_args = rec_args->next){
    proc *rec_p = rec_args->first, *rec_at;

    if(IS_DELTA(rec_p))
      continue;
    assert(KIND(rec_p) == ap_semi);
    rec_at = PARG1(rec_p);
    assert(IS_ATOM(rec_at));
    if(rec_at->kind == a_rec_msg){
      int i;
      pkind snd_kind = communicate(rec_at);

      for(i = 0; i <= nproc; i++){
        proc *p = toolbus[i].proc;
        proc_list *snd_args;

        assert(IN_NORMAL_FORM(p));
        if(i == rec_i)
          continue;
        for(snd_args = PARGS(p);
            snd_args;
            snd_args = snd_args->next){
          proc *snd_p = snd_args->first, *snd_at;

          if(IS_DELTA(snd_p))
            continue;
          assert(KIND(snd_p) == ap_semi);
          snd_at = PARGS(snd_p)->first;
          if(IS_DELTA(snd_at))
            continue;
          assert(IS_ATOM(snd_at));
          if(snd_at->kind == snd_kind){
            env *bindings =
              match_atom(
```

```
                rec_at, snd_at,
                toolbus[rec_i].formals,
                toolbus[rec_i].locals,
                toolbus[i].formals,
                toolbus[i].locals);
            if(bindings != nomatch){
              *the_rec_p = rec_p;
              *snd_i = i;
              *the_snd_p = snd_p;
              return bindings;
            }
          }
        }
      }
    }
  }
  return nomatch;
}

/*--- msg_steps --------------------------------*/

bool msg_steps(void)
{
  int i, j;
  bool change = true, gchange = false;

  while(change){
    change = false;

    for(i = 0; i <= nproc; i++){
      proc *p, *rp;
      env *bindings =
        find_msg(i, &p, &j, &rp);

      if(bindings != nomatch){
        toolbus[i].locals =
          update(bindings, toolbus[i].locals);
        toolbus[i].proc = next(p, i);
        toolbus[j].proc = next(rp, j);
        CHANGE;
      }
    }
  }
  return gchange;
}

/*--- has_matching_note ------------------------*/

env *has_matching_note(
    int i,              /* index of process    */
    proc **rp,          /* resulting process
                           after reading i's
                           own note queue      */
    term_list **rnotes) /* modified note queue  */
{
  proc *p = toolbus[i].proc;
  proc_list *args;
  proc *dt, *at;

  if(!toolbus[i].notes)
    return nomatch;
  assert(KIND(p) == ap_plus);

  for(args = PARGS(p); args; args = args->next){
    dt = args->first;
    if(IS_DELTA(dt))
      continue;
    assert(KIND(dt) == ap_semi);
    at = PARG1(dt); assert(IS_ATOM(at));
    if((KIND(at) == a_rec_note) ||
       (KIND(at) == a_no_note)){
      term_list head, *prev, *nts;

      prev = &head;
      for(head.next = nts = toolbus[i].notes;
          nts;
          nts = nts->next){
        env *bindings =
          match_note(at, nts->first,
                     toolbus[i].formals,
                     toolbus[i].locals);
        if(bindings != nomatch){
          *rp = dt;
          prev->next = nts->next;
          *rnotes = head.next;
          return bindings;
        }
        prev = nts;
      }
    }
  }
  return nomatch;
}

/*--- note_steps -------------------------------*/

bool note_steps(void)
{
  int i;
  bool change = true, gchange = false;

  while(change){
    change = false;

    for(i = 0; i <= nproc; i++){
      proc *p;
      term_list *nts;

      if((p = has_atom(i, tst_snd_note))){
        distr_note(i, PARG1(p));
        toolbus[i].proc = next(p, i);
        CHANGE;
      } else {
        env *bindings =
          has_matching_note(i, &p, &nts);

        if(bindings != nomatch){
          if(KIND(PARG1(p)) == a_rec_note){
            toolbus[i].locals =
              update(bindings, toolbus[i].locals);
            toolbus[i].proc = next(p, i);
            toolbus[i].notes = nts;
            CHANGE;
          }
        } else if((p = has_atom(i, tst_no_note))){
          toolbus[i].proc = next(p, i);
          CHANGE;
        }
      }
    }
  }
  return gchange;
}
```

```
/*--- distr_note -------------------------------*/

void distr_note(int snd_i, proc *snd_at)
{
  term *ev;
  char *id = "** error **";
  term *trm1;
  term_list *snd_args;
  int i;

  assert(IS_ATOM(snd_at) &&
         KIND(snd_at) == a_snd_note);

  snd_args =
    substitute_list(TARGS(snd_at),
                    toolbus[snd_i].formals,
                    toolbus[snd_i].locals);
  ev = mk_term(t_appl, "note", snd_args);
  trm1 = snd_args->first;
  switch(trm1->kind){
  case t_integer: case t_string: case t_appl:
    id = trm1->symbol; break;
  default:
    panic("distr_note (2)");
  }

  for(i = 0; i <= nproc; i++){
    if(i == snd_i)
      continue;
    if(elem(id, toolbus[i].subscriptions)){
       toolbus[i].notes =
         append_term_list(toolbus[i].notes, ev);
     }
  }
}

/*--- atomic_steps -----------------------------*/

bool atomic_steps(void)
{
  int i;
  bool change = true, gchange = false;

  while(change){
    proc *p;

    change = false;
    for(i = 0; i <= nproc; i++)
      {
        if((p = has_atom(i, tst_eval_do_ack))){
          term_list *args = TARGS(PARG1(p));
          char *s = "** error **";

          switch(KIND(PARG1(p)))
            {
            case a_snd_eval:
              s = "eval";  break;
            case a_snd_do:
              s = "do"; break;
            case a_ack_event:
              s = "ack-event"; break;
            default:
              panic("atomic_steps");
            }
          args =
            substitute_list(args,
                            toolbus[i].formals,
                            toolbus[i].locals);
          if(KIND(args->first) != t_appl){
            TBmsg("Illegal tool name %t\n",
                  args->first);
            continue;
          }
          if(!write_to_tool(
                     args->first->symbol,
                     mk_term(t_appl, s, args)))
            continue;
          toolbus[i].proc = next(p, i);
          CHANGE;
        } else if((p = has_atom(i, tst_create))){
          char *pname; term_list * args;
          pname = ID(PARG1(p));
          args =
            substitute_list(TARGS(PARG1(p)),
                            toolbus[i].formals,
                            toolbus[i].locals);
          create_process(pname, args);
          toolbus[i].proc = next(p, i);
          CHANGE;
        } else
          if((p = has_atom(i,tst_subs_unsubs))){
            char *id =
              TARGS(PARG1(p))->first->symbol;
            id_list *subs =
              toolbus[i].subscriptions;

            if(KIND(PARG1(p)) == a_subscribe)
              toolbus[i].subscriptions =
                mk_id_list(id, subs);
            else
              toolbus[i].subscriptions =
                delete(id, subs);
            toolbus[i].proc = next(p, i);
            CHANGE;
          }
      }
  }
  return gchange;
}

/*--- rec_from_tool_step -----------------------*/

bool rec_from_tool_step(term *e)
{
  int i;
  pkind search_kind;

  if(!e)
    return false;
  if(streq(e->symbol, "print")){
    pr_toolbus();
    return true;
  }
  if(streq(e->symbol, "event"))
    search_kind = a_rec_event;
  else if(streq(e->symbol, "value"))
    search_kind = a_rec_value;
  else
    goto ignore;

  for(i = 0; i <= nproc; i++){
    proc *p = toolbus[i].proc, *rec_p, *rec_at;
    proc_list *pargs;
```

```
    assert(KIND(p) == ap_plus);
    for(pargs = PARGS(p);
        pargs;
        pargs = pargs->next){
      rec_p = pargs->first;
      if(IS_DELTA(rec_p))
        continue;
      assert(KIND(rec_p) == ap_semi);
      rec_at = PARGS(rec_p)->first;
      assert(IS_ATOM(rec_at));

      if(KIND(rec_at) == search_kind){
        env *bindings =
          match_list(TARGS(rec_at),
                     e->args,
                     toolbus[i].formals,
                     toolbus[i].locals,
                     toolbus[i].formals,
                     toolbus[i].locals,
                     NULL);
        if(bindings != nomatch){
          toolbus[i].locals =
            update(bindings, toolbus[i].locals);
          toolbus[i].proc = next(rec_p, i);
          return true;
        }
      }
    }
  }
 ignore:
  TBmsg("IGNORING EVENT: %t\n", e);
  return false;
}

/*--- all_internal_steps -----------------------*/

void all_internal_steps(void)
{
  while(msg_steps() ||
        atomic_steps() ||
        note_steps())
    {
      /* do nothing (:-) */
    }
}
```

C.9 General interprocess communication and the server protocol

The communication between ToolBus and tools is based on sockets. First, we provide a basic layer for socket creation and socket connection. In Section C.9.3, we give the server side of the ToolBus interconnection protocol described in Section 10.2. The client side of the protocol is given in Section D.3.

C.9.1 sockets.h

```
int        TBmakeInPort(char *, int);
int        TBmakeOutPort(char *, int);
void       TBdestroyPort(int);
void       putInt(int, const char *, int);
int        getInt(const char *, int);
int        createWellKnownSocket(char *, int);
int        connectWellKnownSocket(char *, int);

extern int gethostname(char *, int);
extern void bzero(void *, int);
extern void bcopy(const void *, void *, int);
extern int setsockopt(int, int, int,
                      const void *, int);
extern int buf_size;
```

C.9.2 sockets.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/param.h>
#include <sys/uio.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <string.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>

#include "toolbus.h"
#include "terms.h"
#include "utils.h"
#include "sockets.h"

int getInt(const char *tname, int port)
{
  char buf[TB_MAX_HANDSHAKE], got_tool_name[256];
  int n, nread, nitems;
  extern int errno;

  if((nread =
      mread(port, buf, TB_MAX_HANDSHAKE)) < 0){
    TBmsg("getInt on port %d: %s\n",
          port, strerror(errno));
  }
  nitems = sscanf(buf,"%s %d",got_tool_name,&n);
  if(nitems != 2 ||
     streq(tname, got_tool_name) == false)
    n = -1;
  if(verbose)
    TBmsg("getInt: got \"%s\", return: %d\n",
          buf, n);
  return n;
}

void putInt(int port, const char *tname,
            int n)
{
  char buf[TB_MAX_HANDSHAKE];
  sprintf(buf, "%s %d", tname, n);
  if(verbose)
    TBmsg("putInt: %s\n", buf);
  if(write(port, buf, TB_MAX_HANDSHAKE) < 0)
    perror("putInt");
}
```

```
int connectsocket (const char *host, int port)
{
  int sock;
  char name[128];
  struct sockaddr_un usin;
  struct sockaddr_in isin;
  struct hostent *hp;
  int isConnected = 0;
  int attempts = 0;

  while (!isConnected){
    if (host == NULL){
      sprintf (name, "/usr/tmp/%d", port);
      if ((sock = socket(AF_UNIX,SOCK_STREAM,0))
          < 0){
        perror("Cant open socket");
        exit(1);
      }
      /* Initialize the socket address to the
         server's address. */
      usin.sun_family = AF_UNIX;
      strcpy (usin.sun_path, name);

      /* Connect to the server. */
      if (connect(sock, &usin,sizeof(usin)) < 0){
        close(sock);
        if (attempts > 1000){
          TBmsg("cannot connect, giving up\n");
          exit (-1);
        }
        attempts++;
      } else {
        isConnected = 1;
      }
      chmod (name, 0777);
    } else {
      if((sock = socket(AF_INET,SOCK_STREAM,0))
         < 0){
        perror("Cannot open socket");
        exit(1);
      }

      /* Initialize the socket address to the
         server's address. */
      bzero((char *) &isin, sizeof(isin));
      isin.sin_family = AF_INET;

      /* to get host address */
      hp = gethostbyname(host);
      if (hp == NULL){
        TBmsg("cannot get host name\n");
        exit(-1);
      }
      bcopy (hp->h_addr, &(isin.sin_addr.s_addr),
             hp->h_length);
      isin.sin_port = htons(port);

      TBmsg("try connecting to ToolBus");
      /* Connect to the server. */
      if(connect(sock, &isin, sizeof(isin)) < 0){
        close(sock);
        if (attempts > 1000){
          TBmsg("cannot connect, giving up\n");
          exit (-1);
        }
        attempts++;
      } else {
        isConnected = 1;
      }
    }
  }
  setsockopt(sock, IPPROTO_TCP, TCP_NODELAY,
             (char *)&sock, sizeof sock);
  setsockopt(sock, SOL_SOCKET, SO_SNDBUF,
             (char *)&buf_size, sizeof(int));
  return sock;
}

int createsocket (const char *host, int port)
{
  int sock, msgsock, length;
  struct sockaddr_un usin;
  struct sockaddr_in isin;
  int   opt = 1;
  char name[128];
  int attempts = 0;

  /* Create a socket */
  if (host == NULL){
    sprintf (name, "/usr/tmp/%d", port);
    unlink (name);

    if((sock=socket(AF_UNIX,SOCK_STREAM,0)) < 0){
      perror("opening stream socket");
      return (-1);
    }

    /* Initialize socket's address structure */
    usin.sun_family = AF_UNIX;
    strcpy (usin.sun_path, name);
  } else {
    if((sock=socket(AF_INET,SOCK_STREAM,0)) < 0){
      perror("opening stream socket");
      return (-1);
    }
    /* Initialize socket's address structure */
    isin.sin_family = AF_INET;
    isin.sin_addr.s_addr = INADDR_ANY;
    isin.sin_port = htons(port);
  }

#ifdef sgi
  /* Solaris doesn't know SO_REUSEPORT */
  setsockopt(sock, SOL_SOCKET, SO_REUSEPORT,
             (char *)&opt, sizeof opt);
#endif
  setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
             (char *)&opt, sizeof opt);
  setsockopt(sock, SOL_SOCKET, SO_RCVBUF,
             (char *)&buf_size, sizeof(int));
  /* Assign an address to this socket */
  if(verbose)
    TBmsg("Binding %s\n", name);
  if (host == NULL)
    while(bind(sock,&usin,
               strlen(usin.sun_path)+
               sizeof(usin.sun_family))
          < 0){
      if (attempts > 1000){
        TBmsg("cannot connect, giving up\n");
        exit (-1);
      }
      attempts++;
      chmod (name, 0777);
```

```
    }
  else
    while (bind (sock,&isin,sizeof(isin)) < 0){
      if (attempts > 1000){
        TBmsg("cannot connect, giving up\n");
        exit (-1);
      }
      attempts++;
    }

  /* Prepare socket queue for connect requests */
  listen(sock,1);

  if(ToolBus &&
     ((port==TB_INPORT) || (port==TB_OUTPORT)))
    return sock;
  else {
    if (host == NULL){
      length = sizeof(usin);
      msgsock = accept(sock, &usin, &length);
      if (msgsock < 0) {
        perror("accept");
        return (-1);
      }
    } else {
      length = sizeof(isin);
      msgsock = accept(sock, &isin, &length);
      if (msgsock < 0) {
        perror("accept");
        return (-1);
      }
    }
    close (sock);
    return msgsock;
  }
}

/* General wrappers following the DTM model */

char *is_local(char *other)
{
  return streq(this_host,other) ? NULL : other;
}

int TBmakeInPort (char *host, int port)
{
  int n;
  n = createsocket (is_local(host), port);
  if(verbose)
    TBmsg("TBmakeInPort(%d) returns %d\n",
          port, n);
  return n;
}

int TBmakeOutPort (char *host, int port)
{
  int n;
  n = connectsocket (is_local(host), port);
  if(verbose)
    TBmsg("TBmakeOutPort (%d) returns %d\n",
          port,n);
  return n;
}

void TBdestroyPort (int port)
{
  close (port);
}

int createWellKnownSocket(char *host, int port)
{
  return TBmakeInPort (host, port);
}

int connectWellKnownSocket(char *host, int port)
{
  return TBmakeOutPort(host, port);
}
```

C.9.3 server.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>

#include "toolbus.h"
#include "terms.h"
#include "utils.h"
#include "sockets.h"

extern int WellKnownSocketIn;
extern int WellKnownSocketOut;
extern int ntool;
extern int find_tool_index(char *);
extern int connect_tool(char *, char *, int, int);

int mk_server_ports()
{
  int in = TB_ERROR,  /* Input port id  */
      out = TB_ERROR; /* Output port id */

  in = createWellKnownSocket(this_host,
                             TB_INPORT);

  if(in < 0)
    return TB_ERROR;
  out = createWellKnownSocket(this_host,
                              TB_OUTPORT);
  if (out < 0) {
    TBdestroyPort(in);
    return TB_ERROR;
  }
  WellKnownSocketIn = in;
  WellKnownSocketOut = out;
  return TB_OK;
}

#define cleanup() \
        if(msgsock >= 0) close(msgsock); \
        if(outmsg >= 0) close(msgsock); \
        if(in >= 0) TBdestroyPort(in); \
        if(out >= 0) TBdestroyPort(out);

int accept_client()
{
/* SERVER SIDE CONNECT --
   1. accept client on well known socket.
   2. send unique port id (counter += 4)
   3. mk ports.
   4. sync.
*/
  int msgsock = TB_ERROR,
      outmsg = TB_ERROR,
```

```
         in = TB_ERROR,
        out = TB_ERROR;
int nread, n, inport;
char buf[TB_MAX_HANDSHAKE];
char tname[256], hname[256];

msgsock = accept(WellKnownSocketIn, 0, 0);
if(msgsock < 0){
  perror("accept");
  cleanup();
  return TB_ERROR;
}
if((nread =
    mread(msgsock,buf,TB_MAX_HANDSHAKE)) <= 0){
  perror("reading toolname");
  cleanup();
  return TB_ERROR;
}
if(sscanf(buf, "%s %s", tname, hname) != 2){
  TBmsg("toolname & hostname: format error\n");
  cleanup();
  return TB_ERROR;
}
if(verbose)
  TBmsg("got toolname & host: %s, %s\n",
        tname, hname);

outmsg = accept(WellKnownSocketOut, 0, 0);
if(outmsg < 0){
  cleanup();
  perror("outmsg");
  return TB_ERROR;
}
n = find_tool_index(tname);
if(n < 0){
  cleanup();
  return TB_ERROR;
}
inport = 2 * n;

putInt(outmsg, tname, inport);
if ((in = TBmakeInPort(this_host, inport))
    == TB_ERROR){
  cleanup ();
  return TB_ERROR;
}
if ((out = TBmakeOutPort(hname,
                         inport + 1))
    == TB_ERROR){
  cleanup();
  return TB_ERROR;
}

close(msgsock);
close(outmsg);

if((n = getInt(tname, in)) != inport){
  TBmsg("got %d should be %d",
        n, inport);
  putInt(out, tname, -1);
  cleanup();
  return TB_ERROR;
} else {
  putInt(out, tname, inport);
  connect_tool(tname, hname, in, out);
  return TB_OK;
}
  return TB_OK;
}
```

C.10 Utilities

Here we introduce the following utility functions which are shared by both the ToolBus implementation and tools:

`parse_term`: parse an incoming message as term.

`TBmakeTerm`: parse a given format string as term and insert appropriate subterms at occurrences of `%t`, `%d`, or `%s`.

`TBwriteTerm`: write a given term to a file.

`TBprintf`: write output (including terms) to file.

`TBmsg`: write a diagnostic message.

`TBmatch`: match a term against given pattern.

The *data exchange protocol* used for exchanges messages is as follows:

- At the lowest level, data consist of a length indication l, followed by a colon, followed by l data bytes. The length indication plus colon take `LENSPEC` bytes. This can be summarized as follows:

$$\underbrace{\text{LLLLLLL}}_{\text{length}}:\underbrace{\text{DDDDDDDDDDDDDDD}}_{\text{data bytes}}$$

- Terms are linearized by simply printing them in prefix form without any intervening layout symbols. The data representation of strings needs special attention. To allow the exchange of arbitrary, binary, data we need to treat string constants as completely opaque entities. To achieve this we represent them as a string quote, a length indication l, a colon, l data bytes, and a closing string quote. This can be summarized as follows:

$$"\underbrace{\text{LLLLLLL}}_{\text{length}}:\underbrace{\text{DDDDDDDDDDDDDDD}}_{\text{data bytes}}"$$

  In this way, the end of the string can be determined without inspecting its contents.

C.10.1 utils.h

```
#define TB_MAX_BUFFER 10000  /* maximum buf size
                                for read/write  */
#define TB_MAX_OVERFLOW 5000

#define TB_MAX_HANDSHAKE 512 /* size of handshake
                                buffer          */

#define TB_OK         0
#define TB_ERROR      -1
#define TB_INPORT     8123
#define TB_OUTPORT    8124

extern char *tool_name;
extern char  buffer[TB_MAX_BUFFER];
extern char *buf_ptr;

extern int overflowPort;
extern bool  stand_alone;
extern bool  verbose;

extern char this_host[MAXHOSTNAMELEN];
extern char single_prompt[];

typedef term *(*TBcallback)(term *);

term *read_term(void);
void  write_term(int, term *);
int   read_from_stdin(void);
term *parse_buffer(void);
int multi_read(int, bool);
int mread(int, char *, int);
char *strndup(char *, char *);
int get_char(void);
void skip_layout(void);

void  TBmsg(char *, ...);
void  TBprintf(FILE *, char *, ...);
bool  TBmatch(term *, char *, ...);
void  TBwriteTerm(FILE *, term *);
term *TBmakeTerm(char *, ...);
char *strerror(int);

extern int select(int, fd_set *, fd_set *,
                  fd_set *exceptfds,
                  struct timeval *);
extern void bcopy(const void *, void *, int);
extern void bzero(void *, int);
extern int gethostname (char *, int);
```

C.10.2 utils.c

```
#include "toolbus.h"
#include "terms.h"
#include "utils.h"

extern void bcopy(const void *, void *, int);

char buffer[TB_MAX_BUFFER]; /* i/o buffer       */
char *buf_ptr;              /* read and write
                               ptr in buffer    */
int buf_size = TB_MAX_BUFFER;

char overflowBuffer[TB_MAX_OVERFLOW];
int  overflowPort = -1;
int  overflowCnt = 0;

char this_host[MAXHOSTNAMELEN];
char single_prompt [] =
     "Enter term (followed by ; character):\n";

int  lastc;

#define LENSPEC 8
#define BUF_PTR_TO_BEGIN_OF_DATA \
        buf_ptr = &buffer[LENSPEC]

bool debug = false;
bool ToolBus = true;
bool verbose = false;     /* no verbose logging */
bool stand_alone = false; /* not running alone  */
char *tool_name = "ToolBus";
bool toBuffer = false;
FILE *toFile = stderr;

/*--- parse_term/TBmakeTerm --------------------*/

int get_char(void)
{
  return lastc = *buf_ptr++;
}

void skip_layout(void)
{
  while(lastc && isspace(lastc))
    get_char();
}

void parse_error(char *s)
{
  TBmsg("%s\n", s);
}

char *strndup(char *bgn, char *end)
{
  char *r,  *s;

  r = s = (char *) malloc(sizeof(*bgn) *
                          (end - bgn + 1));
  while(bgn < end)
    *s++ = *bgn++;
  *s++ = '\0';
  return r;
}

va_list mk_term_args = NULL;

term *parse_term0(void)
{
  char *begin;

  skip_layout();
  begin = buf_ptr - 1;

  if(isupper(lastc)){     /* variable or formal */
    get_char();
    while(isalnum(lastc) || (lastc == '-')) {
      get_char();
    }
    return
      mk_term(t_var,
           resolve(strndup(begin, buf_ptr - 1),
                   tool_name),
```

```
            NULL);
} else
if(isdigit(lastc)){                /* integer */
  get_char();
  while(isdigit(lastc)) {
    get_char();
  }
  return
    mk_term(t_integer,
            strndup(begin, buf_ptr - 1),
            NULL);
} else
if(lastc == '"'){                  /* string */
  int len = 0;
  char *str;

  get_char();
  while(isdigit(lastc)){
    len = len * 10 + (lastc - '0');
    get_char();
  }
  if(lastc != ':'){
    parse_error("Error in string length spec");
    return NULL;
  }
  if(buf_ptr + len >= &buffer[buf_size]){
    parse_error("String length > buffer size");
    return NULL;
  }
  str = strndup(buf_ptr, buf_ptr + len);
  if(*(buf_ptr + len) != '"'){
    parse_error("Missing end quote in string");
    return NULL;
  }
  buf_ptr += len + 1;
  get_char();
  return mk_term(t_string, str, NULL);
} else
if(islower(lastc)){            /* application */
  char *id;
  term_list args;
  term * arg;

  args.next = NULL;
  get_char();
  while(isalnum(lastc) || (lastc == '-')) {
    get_char();
  }
  id = strndup(begin, buf_ptr - 1);
  skip_layout();
  if(lastc == '('){
    do {
      get_char();
      arg = parse_term0();
      if(!arg) {
        parse_error("Syntax error in argument");
        return NULL;
      }
      args.next =
        append_term_list(args.next, arg);
      skip_layout();
    } while (lastc == ',');
    if(lastc != ')'){
      parse_error("Missing , or ) in term");
      return NULL;
    } else
      get_char(); skip_layout();
    }
    return mk_term(t_appl, id, args.next);
  } else
  if(lastc == '%'){   /* insert term from
                         TBmakeTerm's arguments */
    int c;
    c = get_char();
    get_char(); skip_layout();
    switch(c){
    case 'd':
      { char cbuf[64];
        sprintf(cbuf, "%d",
                va_arg(mk_term_args, int));
        return
          mk_term(t_integer, strdup(cbuf), NULL);
      }
    case 's':
      return
        mk_term(t_string,
                va_arg(mk_term_args, char *),
                NULL);
    case 't':
      return va_arg(mk_term_args, term *);
    default:
      parse_error("Illegal character "
                  "follows % in input");
      return NULL;
      }
  } else {
    parse_error("Unexpected character");
    return NULL;
  }
}

term *parse_term(void)
{
  mk_term_args = NULL;
  get_char();
  return parse_term0();
}

term *TBmakeTerm(char * fmt, ...)
{
  term * res;

  va_start(mk_term_args, fmt);
  if(strlen(fmt) >= buf_size)
    panic("format size exceeds buffer size");
  BUF_PTR_TO_BEGIN_OF_DATA;
  strcpy(buf_ptr, fmt);
  get_char();
  res = parse_term0();
  va_end(mk_term_args);
  mk_term_args = NULL;
  return res;
}

/*--- read from a channel and parse as term ----*/

int read_from_stdin()
{
  int c;
  char *ptr;

  ptr = buf_ptr = &buffer[LENSPEC];
  while((c = fgetc(stdin)) != EOF){
    if(c == ';')
```

```
      break;
    else
      *ptr++ = c;
  }
  *ptr++ = '\0';
  return (ptr - buffer) + LENSPEC;
}

term *parse_buffer(void)
{
  term * trm;

  BUF_PTR_TO_BEGIN_OF_DATA;
  skip_layout();
  if(verbose)
    TBmsg("receive: %s\n", buf_ptr);

  if((trm = parse_term()))
    return trm;
  else {
    BUF_PTR_TO_BEGIN_OF_DATA;
    TBmsg("term ignored: \"%s\"\n", buf_ptr);
    return NULL;
  }
}

int multi_read(int fd, bool overflow)
{
  int len, cnt;

  if(overflow){
    assert(fd == overflowPort);
    overflowPort = -1;
    cnt = overflowCnt;
    bcopy(overflowBuffer, buffer, cnt);
  } else
    cnt = read(fd, buffer, buf_size);

  if(cnt <= 0)
    return cnt;
  if(cnt <= LENSPEC)
    return -1;

  if(sscanf(buffer, "%d:", &len) != 1){
    TBmsg("Ignore message (no length indication):"
          " %s\n", buffer);
    return -1;
  }

  len += LENSPEC;
  if(len >= buf_size){
    panic("message exceeds buffer size");
  }
  if(cnt > len){
    overflowPort = fd;
    overflowCnt = cnt - len;
    if(overflowCnt > TB_MAX_OVERFLOW)
      panic("Overflow buffer too small");
    bcopy(&buffer[len], overflowBuffer,
          overflowCnt);
    buf_ptr = buffer + len;
  } else {
    overflowPort = -1;
    buf_ptr = buffer + cnt;
  }

  while (buf_ptr < &buffer[len]){
    cnt = read(fd, buf_ptr,
               &buffer[buf_size] - buf_ptr);
    if(cnt <= 0)
      return -1;
    buf_ptr += cnt;
  }
  *buf_ptr++ = '\0'; /* CHECK */
  return len;
}

int mread(int fd, char *buf, int len)
{
  int cnt = 0, n;

  while(cnt < len){
    if((n = read(fd, &buf[cnt], len - cnt)) <= 0)
      return -1;
    cnt += n;
  }
  return cnt;
}

/*--- TBwriteTerm -----------------------------*/

void write_buffer(int out)
{
  extern int errno;
  int len = buf_ptr - buffer - LENSPEC;

  if(len <= 0){
    TBmsg("zero or negative len in write_buffer\n");
  }
  sprintf(buffer, "%-.*d", LENSPEC-1, len);
  buffer[LENSPEC-1] = ':';
  if(write(out, buffer, len + LENSPEC) < 0)
    TBmsg("write failed (%s)\n", strerror(errno));
}

void write_term(int out, term *e)
{
  if(!e)
    return;
  if(verbose || stand_alone)
    TBmsg("send: %t\n", e);

  if(stand_alone)
    return;

  BUF_PTR_TO_BEGIN_OF_DATA;
  toBuffer = true;
  pr_term(e);
  *buf_ptr++ = '\0';
  write_buffer(out);
  toBuffer = false;
  toFile = stderr;
}

void TBwriteTerm(FILE *f, term *t)
{
  if(t) {
    toBuffer = false;
    toFile = f;
    pr_term(t);
    toFile = stderr;
  }
}
```

```
void print(const char *s)
{
  if(toBuffer){
    int n = strlen(s);
    if(buf_ptr + n >= & buffer[buf_size])
      panic("buffer overflow\n");
    strcpy(buf_ptr, s);
    buf_ptr += n;
  } else {
    fprintf(toFile, s);
  }

}

/*--- TBprintf/TBmsg----------------------------*/

void gen_printf(FILE *f, char *fmt, va_list args)
{
  char *p, *sval;
  int ival;
  term *trm;
  term_list *tl;

  for(p = fmt; *p; p++){
    if(*p != '%'){
      fputc(*p, f);
      continue;
    }
    switch(*++p){
    case 'd':
      ival = va_arg(args, int);
      fprintf(f, "%d", ival);
      break;
    case 's':
      for(sval = va_arg(args, char *);
          *sval;
          sval++)
        fputc(*sval, f);
      break;
    case 't':
      trm = va_arg(args, term *);
      toBuffer = false; toFile = f;
      pr_term(trm);
      toFile = stderr;
      break;
    case 'l':
      tl = va_arg(args, term_list *);
      toBuffer = false; toFile = f;
      pr_term_list(tl);
      toFile = stderr;
      break;
    default:
      fputc(*p, f);
      break;
    }
  }
  fflush(f);
}

/*--- TBprintf ---------------------------------*/

void TBprintf(FILE *f, char *fmt, ...)
{
  va_list args;

  va_start(args, fmt);
  gen_printf(f, fmt, args);
  va_end(args);
}

/*--- TBmsg ------------------------------------*/

void TBmsg(char *fmt, ...)
{
  va_list args;

  va_start(args, fmt);
  fprintf(stderr, "%12s -- ", tool_name);
  gen_printf(stderr, fmt, args);
  va_end(args);
}

/*--- TBmatch ----------------------------------*/

va_list match_args;
char *match_fmt;

bool TBmatch1(term *);

bool TBmatch(term *t, char *fmt, ...)
{
  bool res;

  va_start(match_args, fmt);
  match_fmt = fmt;
  res = TBmatch1(t);
  va_end(match_args);
  return res;
}

void skip_layout_in_fmt()
{
  while(isspace(*match_fmt))
    match_fmt++;
}

bool TBmatch1(term *t)
{
  char **ps, *q;
  int *pi;
  term **pt;
  term_list *tl;

  for( ; *match_fmt; match_fmt++){

    switch(*match_fmt){
    case ' ': case '\t': case '\n':
      continue;
    case '%':
      match_fmt++;
      switch(*match_fmt++){
      case 's':
        if(t->kind != t_string)
          return false;
        else {
          ps = va_arg(match_args, char **);
          *ps = t->symbol;
          return true;
        }
      case 'd':
        if(t->kind != t_integer)
          return false;
        else {
          pi = va_arg(match_args, int *);
```

```
        *pi = atoi(t->symbol);
        return true;
      }
    case 't':
      pt = va_arg(match_args, term **);
      *pt = t;
      return true;
    default:
      TBmsg("TBmatch -- illegal format %s",
            match_fmt - 2);
      exit(0);
    }
  case '"':
    /* -- must be a string -- */
    if(t->kind != t_string)
      return(false);
    for(q = t->symbol; *q; q++){
      if(*q != *match_fmt)
        return (*q == '\0') ? true : false;
      else
        match_fmt++;
    }
    return false;
  case '0': case '1': case '2': case '3':
  case '4': case '5': case '6': case '7':
  case '8': case '9':
    /* -- must be an integer -- */
    for(q = t->symbol; *q; q++){
      if(*q != *match_fmt)
        return (*q == '\0') ? true : false;
      else match_fmt++;
    }
  default:
    if(islower(*match_fmt)){
      /* -- must function symbol -- */
      if(t->kind != t_appl)
        return false;
      for(q = t->symbol; *q; q++){
        if(*q != *match_fmt){
          return false;
        } else
          match_fmt++;
      }
      skip_layout_in_fmt();
      if(*match_fmt != '(')
        return t->args ? false : true;
      match_fmt++;
      for(tl = t->args;
          tl;
          tl = tl->next){
        if(!TBmatch1(tl->first))
          return false;
        else {
          skip_layout_in_fmt();
          if(tl->next && *match_fmt++ != ',')
            return false;
        }
      }
      skip_layout_in_fmt();
      return
        (*match_fmt++ == ')') ? true : false;
    }
    return false;
  }
 }
 return true;
}
```

```
void panic(const char *s)
{
  fprintf(stderr, "%s -- panic -- %s\n",
          tool_name, s);
  abort();
}

char *strerror(int n)
{
  extern char *sys_errlist[];

  return sys_errlist[n];
}
```

C.11 Typechecking

The typechecking of scripts amounts to definition versus use checking of names and detection of recursive use of named processes. Checks currently *not* done are:

- Check that all variables are defined before use.
- Check that all `rec-note` atoms are always preceded by an appropriate `subscribe` atom.
- Check that for each `snd-msg` atom there exists at least one matching `rec-msg` atom.
- Consistency of the terms send to and received from tools.

C.11.1 typecheck.h

```
void use_tl(term_list *, char *, int, int);
void use(char *, char *, int, int);

void calls(char *, char *);
bool typecheck(void);
```

C.11.2 typecheck.c

```
#include "toolbus.h"
#include "terms.h"
#include "env.h"
#include "procdef.h"
#include "utils.h"
#include "interpreter.h"

extern bool exists_tool(char *);

typedef struct entry
{
  char *id;
  char *kind;
  int   lino;
  int   nargs;
} entry;

#define MAXSYMBOL 200

entry symboltable[MAXSYMBOL];
```

```
int nsymbol = -1;

typedef struct call_rel
{
  char *id1;
  char *id2;
} call_rel;

#define MAXCALL 500

call_rel calltable[MAXCALL];
int ncall = -1;

entry *find_use(char *id)
{
  int i;
  for(i = 0; i <nsymbol; i++){
    if(streq(id, symboltable[i].id))
       return &symboltable[i];
  }
  return NULL;
}

void use(char *id, char *kind,
         int lino, int nargs)
{
  entry *e;

  if((e = find_use(id))){
    if(!streq(kind, e->kind)){
      fprintf(stderr,
              "Line %d: use of %s (%s) conflicts "
              "with use on line %d (%s)\n",
              lino, id, kind, e->lino, e->kind);
    } else
    if(streq(kind, "pname") && nargs != e->nargs){
      fprintf(stderr,
              "Line %d: number of arguments of "
              "%s conflicts with use on line %d\n",
              lino, id, e->lino);
    }
  } else {
    if(nsymbol == MAXSYMBOL)
      panic("Too many symbols in ToolBus script, "
            "increase MAXSYMBOL");
    nsymbol++;
    e = &symboltable[nsymbol];
    e -> id = id;
    e->kind = kind;
    e->lino = lino;
    e->nargs = nargs;
  }
}

void use_tl(term_list *args, char *kind,
            int lino, int nargs)
{
  use(args->first->symbol, kind, lino, nargs);
}


bool find_call_rel (char *id1, char *id2)
{
  int i;
  for(i = 0; i <= ncall; i++)
    if(streq(id1, calltable[i].id1) &&
       streq(id2, calltable[i].id2))
```

```
      return true;
  return false;
}

bool calls(char *id1, char *id2)
{
  if(!find_call_rel(id1, id2)){
    if(ncall == MAXCALL)
      panic("Too many calls in call relation, "
            "increase MAXCALL");
    ncall++;
    calltable[ncall].id1 = id1;
    calltable[ncall].id2 = id2;
    return true;
  }
  return false;
}

void closure()
{
  int i, j;
  bool change = true;

  while(change){
    change = false;

    for(i = 0; i <= ncall; i++){
      for(j = 0; j <= ncall; j++){
        if(i != j){
          if(streq(calltable[i].id2,
                   calltable[j].id1))
            if(calls(calltable[i].id1,
                     calltable[j].id2))
              change = true;
        }
      }
    }
  }
}

int typecheck()
{
  proc_def *pd;
  entry *e;
  int i, nerrors = 0;
  extern int lino;

  for(e = symboltable;
      e <= &symboltable[nsymbol];
      e++){
    if(streq(e->kind, "pname")){
      if(!(pd = definition(e->id, definitions))){
        nerrors++;
        fprintf(stderr,
                "Line %d: process %s used but "
                "not defined\n", e->lino, e->id);
      } else
      if(e->nargs != length_id_list(pd->formals)){
        nerrors++;
        fprintf(stderr,
                "Line %d: process %s wrong number"
                " of formal/actual parameters\n",
                e->lino, e->id);
      }
    } else
    if(streq(e->kind, "tool")){
      if(!exists_tool(e->id)){
```

```
        nerrors++;
        fprintf(stderr,
                "Line %d: tool %s used but not "
                "defined\n", e->lino, e->id);
      }
    }
  }

  closure();
  for(i = 0; i <= ncall; i++){
    if(streq(calltable[i].id1,
             calltable[i].id2)){
      nerrors++;
      fprintf(stderr,
              "Line %d: recursive use of %s\n",
              lino, calltable[i].id1);
    }
  }
  return nerrors == 0;
}
```

C.12 TOOLBUS representation

The TOOLBUS itself is implemented as a separate Unix process that interprets a given TOOLBUS 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: for all tools with a non-empty command in their definition execute the tool as a separate Unix process. All other tools are not yet executed, but it is assumed that their execution will be started independently and that they 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 very much resembling the TB-REPR defined in Section 5.4.
- 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.

C.12.1 toolbus.c

```
#include "toolbus.h"
#include "terms.h"
#include "env.h"
#include "procdef.h"
#include "utils.h"
#include "interpreter.h"
#include "typecheck.h"

extern int mk_server_ports(void);
extern int accept_client(void);
extern int yyparse(void);

FILE *fscript;

/*--- tools -----------------------------------*/

#define MAXTOOL 100    /* max # of tools connected
                          to toolbus           */
int ntool = -1;        /* tool counter<=MAXTOOL */

typedef struct tool_def{
  int  index;                /* in tools array */
  char *id;                /* name of the tool */
  char *host;   /* host where tool is executing */
  char *command;        /* command (Unix level) */
  int  pid;                 /* Process id of tool */
  int  out;  /* connection from toolbus to tool */
  int  in;
} tool_def;

tool_def tools[MAXTOOL];

void add_tool(char *id, char *host, char *command)
{
  ntool++;
  if(ntool == MAXTOOL) panic("too many tools");
  tools[ntool].index = ntool;
  tools[ntool].id = id;
  if(streq(host, ""))
    host = this_host;
  tools[ntool].host = host;
  tools[ntool].command = command;
  tools[ntool].in = -1;
  tools[ntool].out = -1;
}

bool write_to_tool(char *id, term *e)
{
  int i, out;

  for(i = 0; i <= ntool; i++){
    if(streq(id, tools[i].id)){
      if((out = tools[i].out) < 0)
        if(stand_alone){
          TBmsg("send to %s: %t\n", id, e);
          return true;
        } else
          return false;
      write_term(out, e);
      return true;
    }
  }
  return false;
}

tool_def *find_tool(char *id)
```

```
{ int i;
  for(i = 0; i <= ntool; i++)
    if(streq(id, tools[i].id))
      return &tools[i];
  return NULL;
}

bool exists_tool(char *id)
{
  return find_tool(id) ? true : false;
}

int find_tool_index(char *id)
{
  tool_def *td = find_tool(id);
  if(!td){
    TBmsg("Tool \"%s\" not defined\n", id);
    return TB_ERROR;
  }
  return td->index;
}

bool connect_tool(char *id, char *host,
                  int in, int out)
{
  tool_def *td = find_tool(id);

  if(!td)
    return false;
  td->host = strdup(host);
  td->in = in;
  td->out = out;
  TBmsg("connect_tool(%s,%s,%d,%d)\n",
        id, host, in, out);
  return true;
}

#include <sys/types.h>
#include <unistd.h>

#define NTOOLARGS 4
void create_tools(void)
{ int i, pid;
  extern int errno;
  const char *std_args[NTOOLARGS];

  if(mk_server_ports() < 0){
    TBmsg("Cannot create input/output ports "
          "of ToolBus\n");
    exit(-1);
  }
  if(stand_alone)
    return;
  for(i = 0; i <= ntool; i++){
    if(streq(tools[i].command, ""))
      continue;
    std_args[0]           = tools[i].id;
    std_args[1]           = "-TB_HOST";
    std_args[2]           = this_host;
    std_args[NTOOLARGS-1] = NULL;

    if((pid = fork())){
      /* toolbus: the parent */
      if(pid < 0){
        TBmsg("Can't fork while "
              "creating tool %s (%s)\n",
              tools[i].command, strerror(errno));
      } else
        tools[i].pid = pid;
    } else {
      /* tool: the child */
      if(execvp(tools[i].command, std_args) < 0){
        TBmsg("Can't execute tool %s (%s)\n",
              tools[i].command, strerror(errno));
        exit(-1);
      }
    }
  }
}

/*--- ToolBus representation -------------------*/

int nproc = -1;   /* process counter <= MAXPROC */
proc_repr toolbus[MAXPROC];
                           /* the actual ToolBus */

proc_def_list *definitions;
      /* list of process defs in current script */

void add_proc_def(char *pname,
             id_list *formals,
             proc *p)
{
  struct proc_def *pd =
    mk_proc_def(pname, formals, p);

  definitions = mk_proc_def_list(pd, definitions);
}

proc *expand(proc *, int);

void create_process(char *pname, term_list *args)
{
  proc_def *pd = definition(pname, definitions);
  proc *p;

  if(nproc == MAXPROC)
    panic("too many processes");
  nproc++;

  toolbus[nproc].name = pd->name;
  toolbus[nproc].formals =
    create_env(pd->formals,
               pd->name,args,
               empty_env);
  toolbus[nproc].locals = empty_env;
  toolbus[nproc].subscriptions = NULL;
  toolbus[nproc].notes = NULL;
  p = pd->proc;
  p = mk_proc_op(p_dot, p, mk_atom(a_delta,NULL));
  p = expand(p, nproc);
  if(KIND(p) != ap_plus){
    proc * q = NEW(proc);
    KIND(q) = ap_plus;
    PARGS(q) = mk_proc_list(p, NULL);
    p = q;
  }
  toolbus[nproc].proc = p;
}

void create_toolbus()
{ id_list *pl;
  extern id_list *toolbus_ids;  /*created by
                                  parser */
```

```
  for(pl = toolbus_ids; pl; pl = pl->next){
    create_process(pl->first, NULL);
  }
  create_tools();
}

void pr_toolbus_i(int i)
{
  fprintf(stderr, "PROCESS %d (%s)\n", i,
          toolbus[i].name);
  pr_proc(toolbus[i].proc);
  fprintf(stderr, "\n");
  if(toolbus[i].formals){
    fprintf(stderr, "Formals: ");
    pr_env(toolbus[i].formals);
    fprintf(stderr, "\n");
  }
  if(toolbus[i].locals){
    fprintf(stderr, "locals: ");
    pr_env(toolbus[i].locals);
    fprintf(stderr, "\n");
  }
  if(toolbus[i].subscriptions){
    fprintf(stderr, "Subscriptions: ");
    pr_id_list(toolbus[i].subscriptions, ", ");
    fprintf(stderr, "\n");
  }
  if(toolbus[i].notes){
    fprintf(stderr, "Notes: ");
    pr_term_list(toolbus[i].notes);
    fprintf(stderr, "\n");
  }
}

void pr_toolbus_entry(char *name)
{
  int i;
  for(i = 0; i <= nproc; i++)
    if(streq(name, toolbus[i].name)){
      pr_toolbus_i(i);
      return;
    }
  fprintf(stderr,
          "Process %s is not part of ToolBus\n",
         name);
}

void pr_toolbus()
{ int i;
  for(i = 0; i <= nproc; i++)
    pr_toolbus_i(i);
}

/* --- reading ---------------------------------*/

int WellKnownSocketIn = TB_ERROR;
int WellKnownSocketOut = TB_ERROR;

int read_from_any_channel()
{
  int i, nelem, error;
  fd_set read_template;
  extern int errno;

  if(overflowPort >= 0){
    nelem = multi_read(overflowPort, true);
    if(nelem > 0)
      return nelem;
  }

retry:
  FD_ZERO(&read_template);
  for(i = 0; i <= ntool; i++){
    if(tools[i].in >= 0){
      FD_SET(tools[i].in,&read_template);
    }
  }
  if(stand_alone)
    FD_SET(0,&read_template);
  else
    FD_SET(WellKnownSocketIn,&read_template);

  if((error = select(FD_SETSIZE, &read_template,
                     NULL, NULL, NULL)) >= 0){
    if(stand_alone){
      nelem = read_from_stdin();
      return nelem;
    } else if(FD_ISSET(WellKnownSocketIn,
                       &read_template)){
      accept_client();
      all_internal_steps();
    }
/* schedule HERE */
    for(i = 0; i <= ntool; i++){
      if((tools[i].in >= 0) &&
         FD_ISSET(tools[i].in,&read_template)){
        nelem = multi_read(tools[i].in, false);
        if(nelem == 0){
          tools[i].in = -1;
          TBmsg("lost connection with %s\n",
                tools[i].id);
          goto retry;
        }
        if(nelem < 0){
          TBmsg("read failed (%s)\n",
                strerror(errno));
          goto retry;
        }
        return nelem;
      }
    }
    goto retry;
  } else {
    TBmsg("select failed (%s)\n",
          strerror(errno));
    goto retry;
  }
}

term *read_term(void)
{
  int nelem;
  term *trm;

  while(true) {
    nelem = read_from_any_channel();
    if(nelem < 0){
      panic("read_term: cannot find "
            "ready input channel");
      continue;
    }
    if((trm = parse_buffer())){
      return trm;
```

```
      }
  }
}

/*--- interpreter ------------------------------*/

void prompt()
{
  if(verbose & debug)
    pr_toolbus();
  if(stand_alone)
    fprintf(stderr, "%s", single_prompt);
}

void interpreter()
{ term * e;

  fprintf(stderr,
          "Starting ToolBus interpreter ...\n");
  all_internal_steps();

  prompt();
  while(true){
    if((e = read_term())){
      rec_from_tool_step(e);
      all_internal_steps();
    }
    prompt();
  }
}

#include <signal.h>

void interrupt_handler(int sig,
                       int code,
                       struct sigcontext *sc){
  exit(-1);
}

/*--- main program -----------------------------*/

int main(int argc, char *argv[])
{
  int i = 1;
  char *vb;

  tool_name = "ToolBus";
  gethostname(this_host, MAXHOSTNAMELEN);
  fprintf(stderr, "host = %s\n", this_host);
  ToolBus = true;
  while (i < argc){
    if(streq(argv[i],"-TB_DEBUG")){
      debug = true; i++;
    } else if(streq(argv[i], "-TB_VERBOSE")){
      verbose = true; i++;
    } else if(streq(argv[i],"-TB_SINGLE")){
       stand_alone = true; i++;
    } else {
      if(!(fscript = fopen(argv[i], "r"))){
        fprintf(stderr, "%s: cannot open: %s\n",
                argv[0], argv[i]);
        exit(1);
      }
      break;
    }
  }
 if((vb = getenv("TB_VERBOSE")) &&
     streq(vb ,"true"))
    verbose = true;

  signal(SIGINT, interrupt_handler);
  if(yyparse() == 0){
    fclose(fscript);
    fscript = NULL;
    if(typecheck()){
      create_toolbus();
      interpreter();
    } else
      fprintf(stderr,
             "Typechecking error(s) in script\n");
  } else {
    printf("Syntax error(s) in script\n");
    exit(-1);
  }
  return 0;
}
```

C.13 Syntax of ToolBus scripts

Here we give Lex/Yacc definitions of extended ToolBus scripts defined earlier in Section 10.

C.13.1 script.l

```
%{                                /* -*-C-*- mode */
int lino = 1;     /* line number in source file */
char *last_lex;      /* last lexical token read */
void ll_sv(void);
%}

layout        [ \t\n\r]
comment       "%%".*
ws            {layout}|{comment}
id            [a-z][A-Za-z0-9\-]*
name          [A-Z][A-Za-z0-9\-]*
string        "\""[^\"]*"\""
int           [0-9]+
%%
\n            { lino++; }
{ws}          {/* */}
snd-msg       { ll_sv();return(SND_MSG); }
rec-msg       { ll_sv();return(REC_MSG); }
snd-note      { ll_sv();return(SND_NOTE); }
rec-note      { ll_sv();return(REC_NOTE); }
no-note       { ll_sv();return(NO_NOTE); }
subscribe     { ll_sv();return(SUBSCRIBE); }
unsubscribe   { ll_sv();return(UNSUBSCRIBE); }
rec-event     { ll_sv();return(REC_EVENT); }
snd-ack-event { ll_sv();return(SND_ACK_EVENT); }
snd-eval      { ll_sv();return(SND_EVAL); }
snd-do        { ll_sv();return(SND_DO); }
rec-value     { ll_sv();return(REC_VALUE); }
toolbus       { ll_sv();return(TOOLBUS); }
define        { ll_sv();return(DEFINE); }
delta         { ll_sv();return(DELTA); }
tool          { ll_sv();return(TOOL); }
host          { ll_sv();return(HOST); }
command       { ll_sv();return(COMMAND); }
create        { ll_sv();return(CREATE); }
{string}      { ll_sv();
                yylval.string =
```

```
                strndup
                  (&yytext[1],
                   &yytext[strlen(yytext) - 1]
                   );
                return(STRING);
              }
{int}         { ll_sv();
               yylval.string = strdup(yytext);
               return(INT);
              }
{id}          { ll_sv();
               yylval.string = strdup(yytext);
               return(IDENT);
              }
{name}        { ll_sv();
               yylval.string = strdup(yytext);
               return(NAME);
              }
.             { ll_sv();return(yytext[0]);}
%%
char msg[100];

void ll_sv()
{
  last_lex = yytext;
}

char *get_token(int tk)
{
  switch(tk){
  case IDENT:
    sprintf(msg, "id \"%s\"", last_lex);
    return msg;
  case NAME:
    sprintf(msg, "name \"%s\"", last_lex);
    return msg;
  case INT:
    sprintf(msg, "int \"%s\"", last_lex);
    return msg;
  case STRING:
    sprintf(msg, "string \"%s\"", last_lex);
    return msg;
  case SND_MSG: case REC_MSG:
  case SND_NOTE: case REC_NOTE: case NO_NOTE:
  case REC_EVENT: case SND_ACK_EVENT:
  case SND_EVAL: case SND_DO: case REC_VALUE:
  case TOOLBUS: case DEFINE: case DELTA:
  case TOOL: case HOST: case COMMAND: case CREATE:
  case SUBSCRIBE: case UNSUBSCRIBE:
    sprintf(msg, "keyword: \"%s\"", last_lex);
    return msg;
  default:
    sprintf(msg, "character: \"%c\"",
            last_lex[0]);
    return msg;
  }
}
```

C.13.2 script.y

```
%{
#include "toolbus.h"
#include "terms.h"
#include "env.h"
#include "procdef.h"
#include "match.h"
#include "utils.h"
#include "typecheck.h"

extern void add_proc_def(char *,id_list *,proc *);
extern void add_tool(char *, char *, char *);

extern FILE * fscript;

/*--- Yacc stack type --------------------------*/

typedef union {          /* Union of types that   */
  char      *string;     /* appear on parse stack */
  term      *term;
  char      *id;
  term_list *term_list;
  appl      *appl;
  proc      *proc;
  proc_def  *proc_def;
  id_list   *id_list;
} yystype;

#define YYSTYPE yystype   /* Yacc stack entries */

char *current_proc = "";  /* Name of the current
                             process definition */

                          /* parameter list of
                             current definition */
id_list *current_formals = NULL;
                          /* names in toolbus
                             configuration      */
id_list *toolbus_ids = NULL;

%}

%token INT
%token STRING
%token IDENT
%token SND_MSG
%token REC_MSG
%token SND_NOTE
%token REC_NOTE
%token NO_NOTE
%token SUBSCRIBE
%token UNSUBSCRIBE
%token REC_EVENT
%token SND_ACK_EVENT
%token SND_EVAL
%token SND_DO
%token REC_VALUE
%token TOOLBUS
%token DEFINE
%token CREATE
%token NAME
%token DELTA
%token TOOL
%token HOST
%token COMMAND

%start script    /* Start symbol of the grammar */

%left '+'
%right '.'
%left '*'

%%

term : INT
```

```
        { $$.term =
            mk_term(t_integer,$1.string,NULL); }
    | STRING
        { $$.term =
            mk_term(t_string, $1.string, NULL); }
    | NAME
        { $$.term =
            mk_term(elem($1.string,
                         current_formals)
                    ? t_form : t_var,
                    resolve($1.string,
                            current_proc),
                    NULL);
        }
    | IDENT
        { $$.term =
            mk_term(t_appl,$1.string,NULL); }
    | IDENT '(' term_list ')'
        { $$.term = mk_term(t_appl, $1.string,
                            $3.term_list);
        }
    ;

term_list:
      term
        { $$.term_list = mk_term_list($1.term,
                                      NULL);
        }
    | term ',' term_list
        { $$.term_list =
            mk_term_list($1.term,
                         $3.term_list);
        }
    ;

term_list_tail:

        { $$.term_list = NULL; }
    | ',' term_list
        { $$.term_list = $2.term_list; }
    ;

atom : SND_MSG '(' term_list ')'
        { $$.proc =
            mk_atom(a_snd_msg, $3.term_list);
        }
    | REC_MSG '(' term_list ')'
        { $$.proc =
            mk_atom(a_rec_msg, $3.term_list);
        }
    | SND_NOTE '(' term_list ')'
        { $$.proc =
            mk_atom(a_snd_note, $3.term_list);
          use_tl($3.term_list, "note", lino, 0);
        }
    | REC_NOTE '(' term_list ')'
        { $$.proc =
            mk_atom(a_rec_note, $3.term_list);
          use_tl($3.term_list, "note", lino, 0);
        }
    | NO_NOTE '(' term_list ')'
        { $$.proc =
            mk_atom(a_no_note, $3.term_list);
          use_tl($3.term_list, "note", lino, 0);
        }

    | REC_EVENT '(' term_list ')'
        { $$.proc =
            mk_atom(a_rec_event, $3.term_list);
          use_tl($3.term_list, "tool", lino, 0);
        }
    | SND_ACK_EVENT '(' term_list')'
        { $$.proc =
            mk_atom(a_ack_event, $3.term_list);
          use_tl($3.term_list, "tool", lino, 0);
        }
    | SND_EVAL '(' term_list ')'
        { $$.proc =
            mk_atom(a_snd_eval, $3.term_list);
          use_tl($3.term_list, "tool", lino, 0);
        }
    | SND_DO '(' term_list ')'
        { $$.proc =
            mk_atom(a_snd_do, $3.term_list);
          use_tl($3.term_list, "tool", lino, 0);
        }
    | REC_VALUE '(' term_list ')'
        { $$.proc =
            mk_atom(a_rec_value, $3.term_list);
          use_tl($3.term_list, "tool", lino, 0);
        }
    | DELTA
        { $$.proc = mk_atom(a_delta, NULL); }
    | CREATE '(' NAME term_list_tail ')'
        { $$.proc =
            mk_create($3.string, $4.term_list);
          use($3.string, "pname", lino,
              length_term_list($4.term_list));
        }
    | SUBSCRIBE '(' term_list ')'
        { $$.proc =
            mk_atom(a_subscribe, $3.term_list);
        }
    | UNSUBSCRIBE '(' term_list ')'
        { $$.proc =
            mk_atom(a_unsubscribe, $3.term_list);
        }

    ;
proc : atom
        { $$.proc = $1.proc; }
    | proc '+' proc
        { $$.proc =
            mk_proc_op(p_plus, $1.proc, $3.proc);
        }
    | proc '.' proc
        { $$.proc =
            mk_proc_op(p_dot,$1.proc,$3.proc);
        }
    | proc '*' proc
        { $$.proc =
            mk_proc_op(p_star,$1.proc,$3.proc);
        }
    | '(' proc ')'
        { $$.proc = $2.proc; }
    | NAME actuals
        { $$.proc =
            mk_proc_call($1.string,$2.term_list);
          use($1.string, "pname", lino,
              length_term_list($2.term_list));
          calls(current_proc, $1.string);
        }
    ;
```

```
actuals:
       '(' term_list ')'
         { $$.term_list = $2.term_list; }
     |
         { $$.term_list = NULL; }
     ;

formals:
       '(' pname_list ')'
         { $$.id_list = $2.id_list; }
     |
         { $$.id_list = NULL; }
     ;

proc_def_head:
       DEFINE NAME formals
         { $$.string =  current_proc = $2.string;
           current_formals = $3.id_list;
         }
     ;

proc_def:
       proc_def_head '=' proc
         { add_proc_def($1.string,
                        current_formals,
                        $3.proc);
           current_proc = "";
           current_formals = NULL;
         }
     ;

proc_def_list:
       proc_def | proc_def proc_def_list
     ;

pname_list:
       NAME
         { $$.id_list =
             mk_id_list($1.string, NULL);
         }
     | NAME ',' pname_list
         { $$.id_list =
             mk_id_list($1.string, $3.id_list);
         }
     ;

host:
       HOST '=' STRING
         { $$.string = $3.string; }
     |
         { $$.string = "\"\""; }
     ;

command:
       COMMAND '=' STRING
         { $$.string = $3.string; }
     |
         { $$.string = ""; }
     ;

tool_def:
       TOOL IDENT '{' host command '}'
         { add_tool($2.string, $4.string,
                    $5.string);
         }
     ;

tool_def_list:
       tool_def | tool_def tool_def_list
     ;

toolbus:
       TOOLBUS '(' pname_list ')'
         { $$.id_list = $3.id_list; }
     ;

script:
       proc_def_list tool_def_list toolbus
         { toolbus_ids = $3.id_list; }
     ;
%%

#include "lex.yy.c"

void yyerror(char *s)
{
  TBmsg("line %d: %s, near %s\n", lino, s,
        get_token(yychar));
  exit(1);
}
```

D The Tool Library

The tool library is a collection of procedures intended to make it easier to write ToolBus tools. Major parts of the ToolBus implementation given before are reused here.

D.1 General header for tools

```
#include "toolbus.h"
#include "terms.h"
#include "utils.h"
#include "tool.h"
```

D.2 Tool

Here we implement the following utilities that are only relevant for tools:

`TBinit`: initialize the tool.

`TBaddInPort`: add an alternative input port.

`TBeventloop`: event loop for tool.

D.2.1 `tool.h`

```
#include <stdio.h>
#include <stdarg.h>

int TBinit(char *, int, char **, TBcallback);
int TBaddInPort(int, TBcallback);
int TBeventloop(void);

typedef struct inport       /* connection info  */
{                           /* for input ports  */
  int        in;
  TBcallback callback;
} inport;
```

D.2.2 tool.c

```
#include "toolbus.h"
#include "terms.h"
#include "utils.h"
#include "tool.h"

extern int mkports(char *, int *, int *);

#define TB_MAX_INPORT 10    /* max # of InPorts
                               per tool         */
int ninports = -1;          /* # of connections
                               currently in use */

inport inportset[TB_MAX_INPORT];

int toToolBus;              /* port to ToolBus  */

char buffer[TB_MAX_BUFFER]; /* i/o buffer       */

char *buf_ptr;              /* read and write
                               ptr in buffer    */

int TBaddInPort(int in, TBcallback fun)
{
  if(ninports == TB_MAX_INPORT)
    panic("Too many inports");
  ninports++;
  inportset[ninports].in = in;
  inportset[ninports].callback = fun;
  return TB_OK;
}

int TBinit(char *tname, int argc, char *argv[],
       TBcallback fun)
{
  char *vb;
  char host_toolbus[MAXHOSTNAMELEN];
  int fromToolBus, i = 1;

  tool_name = tname;
  ToolBus = false;

  gethostname(this_host, MAXHOSTNAMELEN);
  strcpy(host_toolbus, this_host);
  while(i < argc){
    if(streq(argv[i], "-TB_HOST")){
      if(strlen(argv[++i]) > MAXHOSTNAMELEN)
        panic("Name of ToolBus host too long");
      strcpy(host_toolbus, argv[i]);
      i++;
    } else if(streq(argv[i], "-TB_SINGLE")){
      stand_alone = true;
      i++;
    }
  }

  if((vb = getenv("TB_VERBOSE")) &&
     streq(vb ,"true"))
    verbose = true;

  if(stand_alone){
    TBaddInPort(0, fun);
    return TB_OK;
  }

  if(mkports(host_toolbus, &fromToolBus,
           &toToolBus) == TB_ERROR){
    TBmsg("Can't connect to ToolBus\n");
    exit(0);
  }

  TBaddInPort(fromToolBus, fun);
  return TB_OK;
}

int read_from_any_channel(TBcallback *funptr)
{
  int i, nelem, error;
  fd_set read_template;
  extern int errno;

  if(overflowPort >= 0){
    nelem = multi_read(overflowPort, true);
    if(nelem > 0)
      return nelem;
  }

retry:
  FD_ZERO(&read_template);
  for(i = 0; i <= ninports; i++){
    FD_SET(inportset[i].in,&read_template);
  }

  if((error = select(FD_SETSIZE, &read_template,
                     NULL, NULL, NULL))){

    for(i = 0; i <= ninports; i++){
      if(FD_ISSET(inportset[i].in,
                  &read_template)){
        if(inportset[i].in == 0){
          nelem = read_from_stdin();
        } else
          nelem = multi_read(inportset[i].in,
                             false);
        if(nelem == 0){
          TBmsg("lost connection with ToolBus\n");
  exit(-1);
          }
        if(nelem < 0){
          TBmsg("read failed (%s)\n",
                strerror(errno));
          goto retry;
        }
        *funptr = inportset[i].callback;
        return nelem;
      }
    }
  } else {
    TBmsg("select failed (%s)\n",
          strerror(errno));
    goto retry;
  }
  return TB_ERROR;
}

term *tool_read_term(void)
{
  int nelem;
  term *trm;
  TBcallback fun;

  while(true){
    if(stand_alone){
      fprintf(stderr, "%s", single_prompt);
```

```
      fflush(stderr);
    }
    nelem = read_from_any_channel(&fun);
    if(nelem < 0){
      panic("tool_read_term: cannot find "
            "ready input channel");
      continue;;
    }
    if((trm = parse_buffer()))
      return (*fun)(trm);
  }
}

/*--- TBeventloop ------------------------------*/

int TBeventloop(void)
{ term *e;

  while(true)
    if((e = tool_read_term()))
      write_term(toToolBus, e);
}
```

D.3 The client protocol

Here we implement the client side of the ToolBus protocol (see Section C.9.3 for the server side of the protocol).

D.3.1 client.c

```
#include "toolbus.h"
#include "terms.h"
#include "utils.h"
#include "sockets.h"

/* --- CLIENT SIDE CONNECT -------------------- */

/* client side mkports --
   1. connect to server
   2. get port id
   3. mk ports
   4. sync
 */

#define cleanup() \
        if(in >= 0) close(in);\
        if(out >= 0) close(out);\
        if(*tin >= 0) TBdestroyPort(*tin);\
        if(*tout >= 0) TBdestroyPort(*tout);

int mkports (char *tb_host, int *tin, int *tout)
{
  int portin, n;
  char buf[TB_MAX_HANDSHAKE];
  extern errno;

  int in = TB_ERROR,  /* Input port id  */
     out = TB_ERROR;  /* Output port id */

  *tin = *tout = TB_ERROR;

 retry:
  /* 1. connect to well known socket */
  out =
    connectWellKnownSocket(tb_host, TB_INPORT);
  if (out < 0)  return TB_ERROR;

  /* client coming from "this_host",
     open connection */
  sprintf(buf, "%s %s", tool_name, this_host);

  if(write(out, buf, TB_MAX_HANDSHAKE) < 0){
    TBmsg("can't write (%s)\n", strerror(errno));
    cleanup();
    return TB_ERROR;
  }

  /* connect to well known socket */
  in =
    connectWellKnownSocket(tb_host, TB_OUTPORT);
  if (in < 0) { cleanup (); return TB_ERROR; }

  /* 2. get port number from server */
  portin = getInt (tool_name, in);

  /* close well known sockets */
  TBdestroyPort (in);
  TBdestroyPort (out);

  /* sanity check */
  if (portin < 0){
    TBmsg("mkports, retrying ...\n");
    goto retry;
  }

  /* 3. make ports (out, in) */
  /* create the read end, connect to write end */

  if ((*tout = TBmakeOutPort(tb_host, portin))
      == TB_ERROR){
    cleanup ();
    return TB_ERROR;
  }

  if ((*tin = TBmakeInPort(this_host, portin+1))
      == TB_ERROR){
    cleanup ();
    return TB_ERROR;
  }

  /* 4. synchronize with server */
  putInt (*tout, tool_name, portin);
  if((n = getInt (tool_name, *tin)) != portin){
    TBmsg("got %d should be %d\n", n, portin);
    cleanup();
    return TB_ERROR;
  }
  return TB_OK;
}
```