Chapter 1. docGuide to ToolBus Programming

Hayco de Jong
Paul Klint
Arnold Lankamp
Pieter Olivier

2008-07-15 09:48:10 +0200 (Tue, 15 Jul 2008)

Table of Contents

Introduction 2
  Background and Motivation 2
  ToolBus requirements 3
  The ToolBus architecture 4
  How to go from here? 5
Tscripts 6
  Terms 6
  Matching 6
  Types 7
  Tscripts in detail 8
Examples of Tscripts 14
  Calculator Example 14
  Auction Example 20
  Wave Example 28
Executing ToolBus and tools 32
  Common arguments 32
  ToolBus arguments 32
  Tool arguments 33
ToolBus tools 34
  The global structure of a ToolBus tool 34
  Adapters for tools and languages 35
  Automatic generation of tool interfaces 36
Writing ToolBus tools in C 37
  The include file `atb-tool.h` 37
  The tool library `libATB.a` 38
  The ToolBus API 38
  Compiling ToolBus tools written in C 41
  Automatic generation of C tool interfaces 42
Writing ToolBus tools in Java 46
  The `AbstractJavaTool` class 47
  Tool definition in the hello script 48
  The hello example in Java 48
Writing ToolBus tools in other languages 49
  Writing tools in Tcl/TK 49
  Python 50
  Perl 50
Reference Information 50
  The syntax of Tscripts 50

Built-in functions 52
Synopsis of ToolBus primitives 57
Historical notes 59
Bibliography 59
To Do 60

Warning

This document is in state of creation and will further evolve. It provides a description of the classic C-based ToolBus as well as of the Java-based ToolBusNG. Eventually, it will exclusively focus on ToolBusNG. See the section called “*To Do*” [60].

Introduction

Background and 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.

Three aspects of heterogeneous, distributed, systems should be considered: *coordination*, *representation* and *computation*.

Coordination. Coordination is the way in which program and system parts interact with each other using, ordinary procedure calls, remote procedure calls (RPC), remote method invocation (RMI), and others.

Representation. Representation is the language and machine neutral format for data being exchanged between components.

Computation. Computation is done by specialized program code that carries out a specific task, e.g., providing a user-interface, providing database access, and the like.

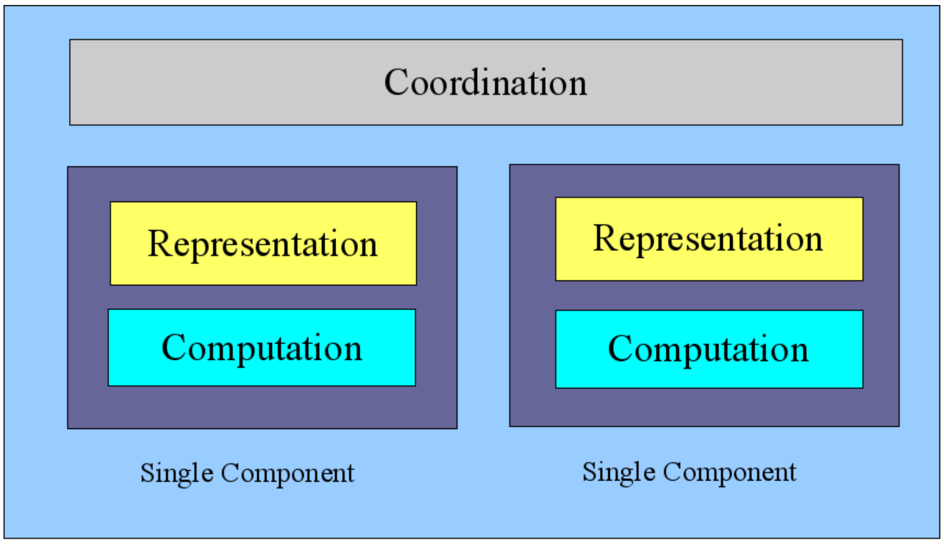
Our key assumption is as follows:

Important

A rigorous separation of coordination from computation is the key to flexible and reusable systems.

A system organization that respects this separation is shown in

Figure 1.1. Separating coordination from computation



We propose to get control over the possible interactions between software components (*tools*) by forbidding direct inter-tool communication. Instead, all interactions are controlled by a process-oriented *script* that formalizes all the desired interactions among tools. This leads to a component interconnection architecture resembling a hardware communication bus, and therefore we call it a ``ToolBus".

ToolBus requirements

Given the motivation for the ToolBus we can briefly summarize the requirements that the ToolBus should satisfy:

- Provide a flexible interconnection architecture for software components that are not only written in different languages and executing on different hardware and software platforms, but are also running in a distributed fashion on a system of networked computers and devices. **Rationale**: it is more and more common that applications are built using existing commercial or open source components. This reduces implementation effort but increases the need for usable interconnection technology.

- Provide good control over the communication between components. **Rationale**: component integration requires both control over the communication between components and over the data that are being exchanged (see next requirement).

- Provide a uniform data exchange mechanism between heterogeneous components. **Rationale**: a common understanding about data formats is needed in order to exchange data between components.

- The description of communication should be based on existing concurrency theory and provide the option for formal verification of the cooperation between software components. **Rationale**: when components are integrated that run on different machines or on multi-core machines, it is unavoidable that concurrency is taken into account and to use existing theory to describe it. The long term perspective of checking formal aspects of these cooperation is appealing for certain, safety-critical, applications.

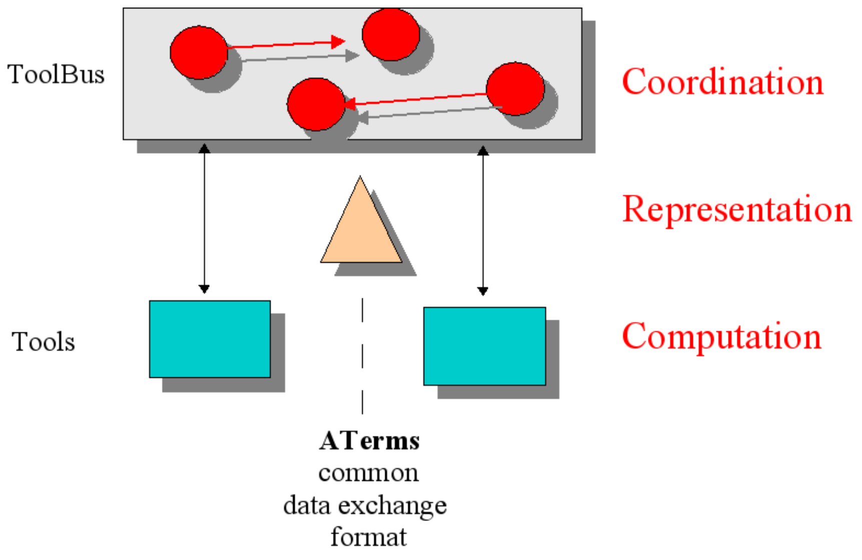
- Provide relatively simple application descriptions that can be understood by most programmers. **Rationale**: we don't want to frighten programmers by using formal notations.

- Provide multi-lingual support, at least C, Java, ASF+SDF, Tcl/Tk, and possibly Perl, Python and Ruby should be supported. Rationale: various tools of interest are currently implemented in the first four languages, and the last three languages are interesting for future developments.

The ToolBus architecture

The global architecture of the ToolBus is shown in Figure 1.2, “Global organization of the ToolBus” [4]. The ToolBus serves the purpose of defining the cooperation of a variable number of *tools* T_i (i = 1, ..., m) that are to be combined into a complete system. The internal 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.

Figure 1.2. Global organization of the ToolBus



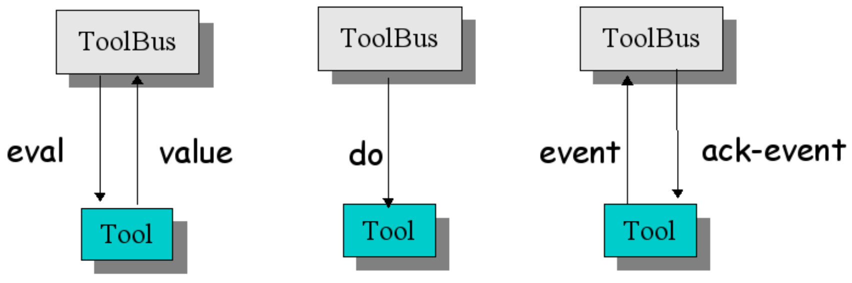
The ToolBus itself consists of a variable number of processes P_i (i = 1, ..., n)[1]The parallel composition of the processes P_i represents the intended behaviour of the whole system. Tools are external, computational activities, most likely corresponding with operating system level tasks. They come into existence either by an execution command issued by the ToolBus or their execution is initiated externally, in which case an explicit connect command has to be performed by the ToolBus. 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.

Communication inside the ToolBus. Inside the ToolBus, there are two communication mechanisms available. First, a process can send a *message* (using `snd-msg`) which should be received, synchronously, by one other process (using `rec-msg`). Messages are intended to request a service from another process. When the receiving process has completed the desired service it may inform the sender, synchronously, by means of another message (using `snd-msg`). The original sender can receive the reply using `rec-msg`. By convention, part of the original message is contained in the reply (but this is not enforced).

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`). 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*.

[1]By ``processes'' we mean here computational activities *inside* the ToolBus as opposed to, for instance, processes at the operating system level. When confusing might arise, we will call the former ``ToolBus processes'' and the latter ``operating system level tasks''.

Figure 1.3. Communication between ToolBus and tools

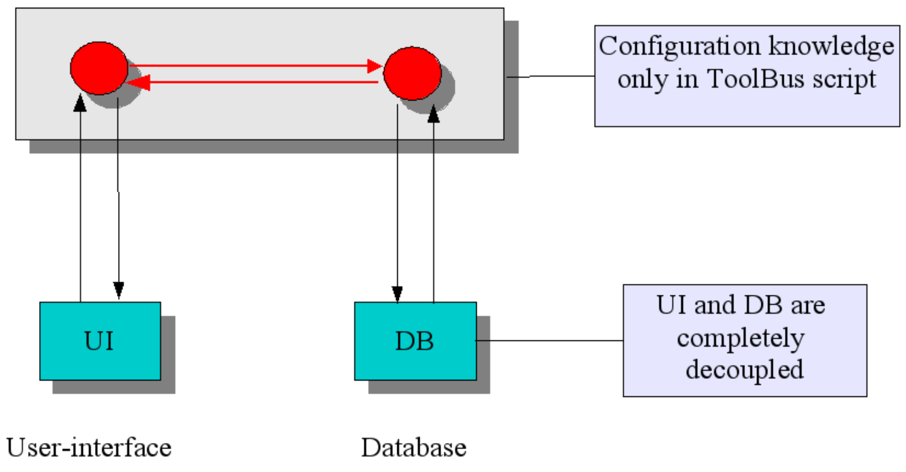


Communication between ToolBus and tools. The communication between ToolBus and tools is based on handshaking communication between a ToolBus process and a tool. A process may send messages in several formats to a tool (`snd-eval`, `snd-do`, and `snd-ack-event`) while a tool may send the messages `snd-event` and `snd-value` to a ToolBus process. There is no direct communication possible between tools. These communication patterns are shown in Figure 1.3, “Communication between ToolBus and tools” [5].

The execution and termination of the tools attached to the ToolBus can be explicitly controlled. It is also possible to connect or disconnect tools that have been executing independently of the ToolBus.

Knowledge separation. Equipped with the mechanisms provided by the ToolBus, careful control over application knowledge can be achieved as shown in Figure 1.4, “Knowledge separation in ToolBus-based application” [5] where an application is depicted consisting of a user-interface (UI) and a database (DB). In a more conventional approach, elements of the user-interface, say a button, would be directly connected with functions in the database component and a strong coupling between the two components would be the result. Using the ToolBus, the two components can be completely oblivious of each other. It is only in the ToolBus script that they are configured to work together. The extra level of indirection introduced by the ToolBus thus leads to extra flexibility and decoupling.

Figure 1.4. Knowledge separation in ToolBus-based application


Configuration knowledge only in ToolBus script
UI
DB
UI and DB are completely decoupled
User-interface
Database

How to go from here?

After this brief motivation and explanation of the ToolBus architecture it is time to delve into more details. In the remainder of this chapter, we will have a look at the following topics:

- ToolBus scripts (or Tscripts, for short).
- How to write ToolBus tools.

- A brief peek at the ToolBus implementation.
- Historical notes

Tscripts

Tscripts describe how the tools in an application cooperate. They allow the definition of a collection of concurrent processes that can communicate with each other and with the tools in the application.

Terms

Tscripts make heavy use of *terms*, simple prefix expressions that are used to exchange structured data between processes and tools. Terms are recursively defined as follows:

- A Boolean constant, integer constant, real constant, or string constant is a term, e.g., `true`, `37`, `314e-12`, or `"rose"`.
- A *value occurrence* of a variable is a term, e.g., `X`, `InitialAmount`, or `Highest-Bid`.

  Important

  Variables always start with a capital letter. A value occurrence serves the purpose of using the current value of a variable.

- A *result occurrence* of a variable is a term, e.g., `X?`, `InitialAmount?` or `Highest-Bid?`.

  Important

  A result occurrence of a variable plays a role when this term is *matched* with another term. In the case that the match succeeds, the corresponding part of the other term is assigned to the result variable.

- A single *identifier* is a term, e.g., `f`, `pair`, or `zero`.

  Important

  Identifier always start with a lowercase letter.

- A *function application* is a term, e.g., `pair("rose", address("STREE", 12345)`.
- A *list* is a term, e.g., `[a, b, c]` or `[a, 1.25, "lost"]`.
- A placeholder is a term, e.g., `<int>` or `add(<int>,<int>)`.

Matching

Term matching is used for several purposes in the ToolBus:

- To determine which actions can communicate with each other. For instance, a `snd-msg` and a `rec-msg` can only communicate if their arguments match.
- To transfer information between sender and receiver.
- To do case analysis, for instance, when receiving events from a tool.

Intuitively, the matching between two terms works as follows:

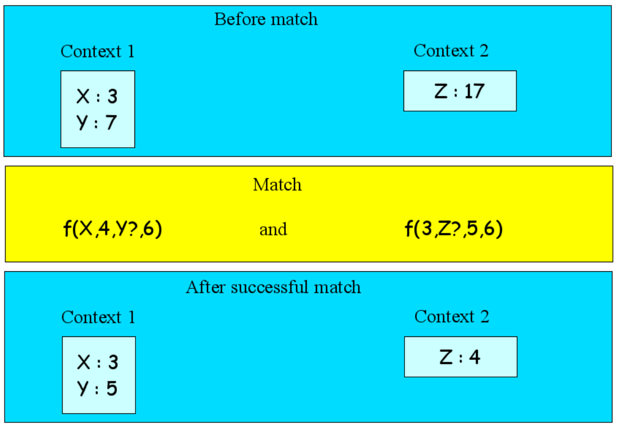
- Two terms match if they are structurally identical.
- For a value occurrence of a variable: use its current value.
- For a result occurrence of a variable: assign the matched subterm of the other term to the variable (but make this only permanent if the overall match succeeds).

This illustrated in Figure 1.5, “Example of term matching”[7]. Before the match, two contexts are given. Each context associates some variables with a value. For instance, Context 1 associates the value `3` with variable `X`. For each context a term is given and the challenge is to match these two terms and to observe the effects on the two contexts. The matching of the two terms can be understood as follows:

- The top level function names are identical (both `f`) and both have the same number of arguments. The left term and the right term match if their arguments match.
- The first argument in the left term is `X` and `3` in the right term. Since, `X` has value `3` in Context 1, they match.
- The second argument in the left term is `4` and `Z?` in the right term. By assigning `4` to `Z` in Context 2 we achieve a match.
- The third argument in the left term is `Y?` and `5` in the right term. Here we achieve a match by assigning `5` to `Y` in Context 1.
- The fourth and last argument of both terms is `6` and thus matches.

The net result is that both terms match and that Context 1 and Context 2 are modified as shown at the bottom of the figure.

Figure 1.5. Example of term matching



Types

The ToolBus uses a type system that is a compromise between the safety of static checking and the flexibility of dynamic typing. Another objective of the type system is to provide sufficient information to enable the automatic generation of adapter code for tools. Type are defined as follows:

- `bool`, `int`, `real` and `str` are the types of the elementary terms.
- `list` is the type of arbitrary lists.
- `list(`*`Type`*`)` is the type of lists with elements of type *`Type`*. For instance, `list(int)` is the type of lists of integers.

- *`Id`* is the type of all terms with function symbol *`Id`* (this allows the declaration of partial types). The type `f`, thus corresponds to the terms `f`, `f(1)`, `f("abc", 3)` and the like.

- `Id(`*`Type`*$_1$`, ..., `*`Type`*$_n$`)` is the type of terms with function symbol *`Id`* and the given types *`Type`*$_1$, ..., *`Type`*$_n$a s argument types. For instance, `f(int,str)` accepts `f(3,"abc")` but not `f(3)`.

- [*`Type`*$_1$, ..., *`Type`*$_n$] is the type of a list of elements with the given types *`Type`*$_1$, ..., *`Type`*$_n$*`.`* For instance, `[int, str]` accepts `[1, "abc"]`, but not `[1,2,3]`.

- `term` is the type of an arbitrary term. And is used as escape from the more precise typing by the preceding types.

Types are used in the following ways:

- All variables have a type.

- Types are statically checked whenever possible. Only in the case of type `term`, dynamic checks are needed.

- Types play a role during matching: a match can also fail if the types of corresponding subterms are unequal. For instance, given `I` as `int` variable, `S` as `str` variable and `T` as `term` variable,

  - `f(13)` and `f(I?)` will match.

  - `f(13)` and `f(S?)` will fail.

  - `f(13)` and `f(T?)` will succeed.

Tscripts in detail

Overall structure

A Tscript can define the following ingredients:

- A *process definition* consisting of a process name, optional parameters and a process expression that describes the behaviour of this process.

- A *tool definition* consisting of a tool name and some operational details, such as the command to execute when the tool is started.

- A *ToolBus configuration* consisting of one or more process names (optionally followed by actual parameters) that will be created when the application is started. A Tscript may contain more than one ToolBus configuration.

- An *include file* that contains another Tscript that will be literally included.

- A *constant definition*.

- A *conditional* that allows the conditional inclusion or exclusion of parts of the Tscript.

A first example

Before delving into the details of Tscripts, it is good to have a look at the hello world application `hello1.tb` shown in Example 1.1, “hello1.tb” [9].

Example 1.1. hello1.tb

```
process HELLO ❶
is ❷
  printf(“Hello world, my first Tscript!\n”) ❸

toolbus(HELLO) ❹
```

Notes:

❶ Here starts the definition of a process with name `HELLO`.

❷ After the keyword `is` follows the process expression that defines the behaviour of this process.

❸ The process expression consists of a single action that prints a string.

❹ Define the initial ToolBus configuration, in this case only process HELLO will be started.

Running this example will yield the following command line dialog:

```
1> toolbus hello1.tb
Hello world, my first Tscript!
2>
```

Becoming more courageous, we show now a more ambitious Tscript `hello2.tb` in Example 1.2, “hello2.tb” [10] that does not print the hello string itself, but executes a tool to compute it.

Example 1.2. hello2.tb

```
process HELLO is ❶
  let H : hello, ❷
      S : str    ❸
  in
         execute(hello, H?) ❹ .   ❺
         snd-eval(H, get_text) .  ❻
         rec-value(H, text(S?)).  ❼
         printf(S)                ❽
  endlet

tool hello is {command = “hello” } ❾
toolbus(HELLO)                     Ⓐ
```

Notes:

❶ Define a process `HELLO`.

❷ Use a `let ... in ... endlet` construct to declare local variables. Variable `H` is declared with type `hello`.

❸ Variable `S` is declared with type `str`.

❹ Execute the `hello` tool (according to the tool definition at ❾ [10]). The resulting tool identifier is assigned to variable `H`. Observe that the name of the tool and the type of `H` are identical.

❺ Use the sequential composition operator `.` to combine atom actions into a larger process expression.

❻ Send an evaluation request to the tool we have just executed. `H` identifies the tool instance, and `get_text` is the term to be sent to the hello tool.

❼ In response to the evaluation request, the hello tool returns a value of the form `text("Hello world from my first tool")`. The actual text is extracted by the result variable `S?`.

❽ Print the string value of `S`.

❾ The definition for the hello tool. It contains the name of an executable program to be run when this tool is executed.

Ⓐ The initial ToolBus configuration consisting of just the `HELLO` process.

All the Tscript primitives (including the ones that occur in these two simple examples) will now be described in more depth.

Process primitives

During execution, the ToolBus consists of a parallel composition of processes. The ToolBus configurations define the processes that are created at the start of the start of the application, but later on processes may die and new ones may be created.

Each process has a local state in the form of private local variables. These variables get their value through assignment and matching. They are only visible inside each process.

Processes are built-in up from atomic actions (detailed below) and atomic actions can be combined into process expressions using the following operators:

- *Sequential composition* P_1 . P_2. First the actions in $P_1$are executed and then the ones in P_2.
- *Choice* P_1 + P_2. A choice is made between the first action in $P_1$and the first action in P_2. This choice is based on two criteria:

- An action to be selected must be *enabled*.
- If more than one action is enabled, a random choice is made.

There are various ways in which an action can be enabled (this depends on the precise action):

- An associated condition evaluates to true (see conditional and guarded command, below).
- An associated timing constraint is true.
- Required external tool results are available.
- Communication conditions are satisfied.

Once the choice for the first action has been made all remaining actions of the selected process expression P_1 or P_2 are executed as well.

- *Parallel composition* P_1 `||` P_2. The actions in $P_1$and P_2 are executed in parallel. This means that the sequential order of the actions in P_1respectively P_2 is respected but that apart from this constraint the actions can be executed in arbitrary order.
- *Iteration* P_1 `*` P_2. P_1 is executed repeatedly, until an action of P_2 is executed. Execution then continues with the remaining actions of P_2.
- *Conditional* `if` T `then` P_1 `else` P_2 `fi`. The test T is evaluated and if the result is true then P_1 is executed, otherwise P_2 is executed. Note that the evaluation of the test does not count as a separate atomic action; the test is effectively attached to the first atom of P_1respectively P_2.
- *Guarded command* `if` T `then` P `fi`. The test T is evaluated and if the result is true then P_1 is executed, otherwise this command deadlocks.

Local variables

As we have seen local variables play a key role in the execution of Tscripts. They are defined using the `let` construct:

- let Var1 : Type1, ... in P endlet. Variables Var1, ... are declared with respective type Type1, ... These variables act as local variables during the execution of the process expression P. P may contain other `let` constructs.

Primitive actions

- *Deadlock* `delta`. This constant represents the process that cannot execute any further steps. During execution deadlock is always avoided as long as this is possible. A process that end in deadlock effectively terminates and disappears.
- *Silent step* `tau`. This constant represents one internal step in a process and resemble a dummy statement in a conventional programming language.
- Print printf. An action for generating formatted output.
- *Assignment* V `:=` T. The term `T` is evaluated as expression (using the built-in functions) and the result is assigned to the local variable V.

Messages: synchronous communication primitives

Synchronous communication resembles an ordinary phone call: it involves two processes that can communicate at the same instant in time. In ToolBus terminology *messages* are used for synchronous communication. There are two primitives involved:

- `snd-msg` sends a message to another process.
- `rec-msg` receives a message from another process.

Two requirements have to be satisfied before communication can take place:

- The arguments of `snd-msg` and `rec-msg` match with each other.
- In addition, `snd-msg` respectively `rec-msg` are enabled in each process.

When communication takes place, the effects of the argument matching is recorded in the local state of each process and both continue execution. The observant reader may have noticed that sending and receiving is actually symmetric: by way of result variables in the arguments of `snd-msg` and `rec-msg` information may flow from sender to receiver and *vice versa*.

Notes: asynchronous broadcasting primitives

Asynchronous communication resembles conventional e-mail: it involves one sending and zero or more receiving processes that read the communicated information at a later instant in time. In ToolBus terminology *notes* are used for asynchronous communication. There four primitives involved:

- `subscribe(`*`T`*`)`. Subscribes a process to notes that match the term *`T`*.
- `unsubscribe(`*`T`*`)`. Unsubscribes a process from notes that match *`T`*.
- `snd-note(`*`T`*`)`. Broadcast the term *`T`* to all subscribed processes. Effectively, *`T`* is placed in the private inbox of each subscribed process to be read at a later moment.
- `rec-note(`*`T`*`)`. Receive a note that matches *`T`*. Effectively, the private inbox is searched for a note that matches *`T`*.
- `no-note(`*`T`*`)`. There is no note that matches *`T`* in the private inbox.

Using named processes

A process definition associates a name *`Pnm`* (optimally followed by parameters) with a process expression *`P`*. These process names can be used in two ways in process expressions:

- An *inline process expression* *`Pnm`*`(...)`: Effectively, this amount to macro substitution: *`Pnm`* is replaced by the process expression *`P`* (after proper parameter substitution).
- A *process creation* `create(`*`Pnm`*`(...), `*`Pid`*`?)`: a completely new process is created that runs in parallel with all other processes currently running in the ToolBus. The process identifier of this new process is assigned to *`Pid`*.

Tool primitives

There two possible scenarios for a ToolBus tool. In scenario 1, the tool is executed from the ToolBus, the tool receives a number of evaluation requests and/or generates an number of events, and finally, the ToolBus decides to terminate the execution of the tool. A variation of scenario 1 is that the tool decides to disconnect from the ToolBus and continues execution disconnect from the ToolBus application. In scenario 2, the tool is executed separately and starts its cooperation by requesting a connection with the ToolBus. Once connected, it follows the same steps as in scenario 1. The following primitives achieve this (also see Figure 1.3, “Communication between ToolBus and tools”[5] for the various communication patterns between ToolBus and tools):

- `execute(`*`Tnm`*`, `*`Tid`*`?)`: Execute a tool with name *`Tnm`*. The result is a *tool identifier* that is bound to *`Tid`*. Tool identifiers are unique; if more than one instance of the same tool is executing they can be distinguished via their tool identifier. There are two additional constraints:
  - The Tscript should contain a tool definition for *`Tnm`*.

  - The variable Tid should have a type that corresponds with the tool name, i.e., it should be declared as `Tid : Tnm`. Why? Well this in this way the implementation can track via the type of the tool identifier in each tool request, *which* tool it is and that information is essential for the automatic generation of adapter code.

- `snd-terminate(Tid, T)`: terminates the execution of the tool instance `Tid`. The term `T` contains a reason for the termination and is usually printed by the tool on termination.
- `rec-connect(Tnm, Tid?)`: receive a connection request for a tool with name `Tnm`. `rec-connect` is very similar to `execute`. The only difference is the initiating party: for `execute` the ToolBus and for `rec-connect` the tool.
- `rec-disconnect(Tid?)`: receive a disconnection request from a tool. It does not matter whether the connection with the tool was originally established via `execute` or `rec-connect`.
- `snd-eval(Tid, T)`: send an evaluation request to a tool. All value occurrences in `T` are first replaced by their value before sending `T` to the tool. It is up to the tool to interpret the term. The usual scenario is that the outermost function symbol of `T` is identical to the name of a procedure in the tool and that procedure is called. The ToolBus can only send one evaluation request at a time. Only when the request is cancelled, or a value is returned by the tool, the next request can be sent to the tool.
- `rec-value(Tid, T)`: receive a value from a tool in response to a previous `snd-eval` request. `T` has to match the value from the tool; this is useful for case distinctions. In many case, T consists of a single result variable, or a is a term that contains result variables.
- `snd-do(Tid, T)`: send an evaluation request to a tool but do not expect a return value. Typically used to implement printing or logging activities.
- `rec-event(Tid,` T_1`, ...)`: receive an event from a tool. Events need not be handled one-by-one. The same tool may generate more than one event provided that the value of argument T_1 differs. T_1 thus serves as identification for this event.
- `snd-ack-event(Tid,` T_1`)`: acknowledge the completion of the handling of a previous event. Since, T_1 is identical to the T_1 in a preceding `snd-event` and is used to identify that event.

Timing primitives

Time can play an important role in applications, be it as ingredient in a protocol that prescribes certain time constraints, be it as watchdog that certain operations are carried out in time. The general approach in Tscripts is that a delay or timeout may be attached to every atom action. Delays and timeouts may be relative to the current time or they may be specified in absolute time. The primitives are as follows (for arbitrary atomic action `A`):

- *Relative delay*: `A delay(E)`. Atom `A` can only become enabled after `E` seconds have passed.
- *Absolute delay*: `A abs-delay(Year, Month, Day, Hour, Min, Sec)`. Atom `A` can only become enabled after the specified absolute date and time.
- *Relative timeout*: `A timeout(E)`. Atom `A` is only enabled during the next `E` seconds.
- *Absolute timeout*: `A abs-timeout(Year, Month, Day, Hour, Min, Sec)`. Atom `A` is only enabled until the specified absolute date and time.

Expressions

Terms can occur in Tscripts on various locations. In the majority of cases these terms are used as such; only variables are replaced by their value but no further evaluation of terms take place. There are, however, two exceptions to this general rule. In three cases, terms are evaluated:

- The test in `if T then ... fi` and `if T then ... else ... fi`.
- The right-hand side of the assignment `V := T`.
- In delays or timeouts.

The term is evaluated in a bottom-up manner, i.e., first arguments are evaluated and then the function is applied. Here are some examples:

- `not(true)` evaluates to `false`.
- `add(mul(2,3), 4)` evaluates to `10`.
- `greater(6,5)` evaluates to `true`.
- `first([9, 8, 7])` evaluates to `9`.

A detailed overview of all built-in functions is given in XXX. They can be summarized as follows:

- Functions on *Booleans*: `not`, `and`, `or`.
- Functions on *Integers*: `add`, `sub`, `mul`, `div`, `mod`, `less`, `less-equal`, `greater`, `greater-equal`.
- Functions on *lists*: `first`, `next`, `get`, `put`, `join`, `member`, `subset`, `diff`, `inter`, `size`.
- *Miscellaneous* functions: `equal`, `not-equal`, `process-id`, `process-name`, `current-time`, `quote`.

Examples of Tscripts

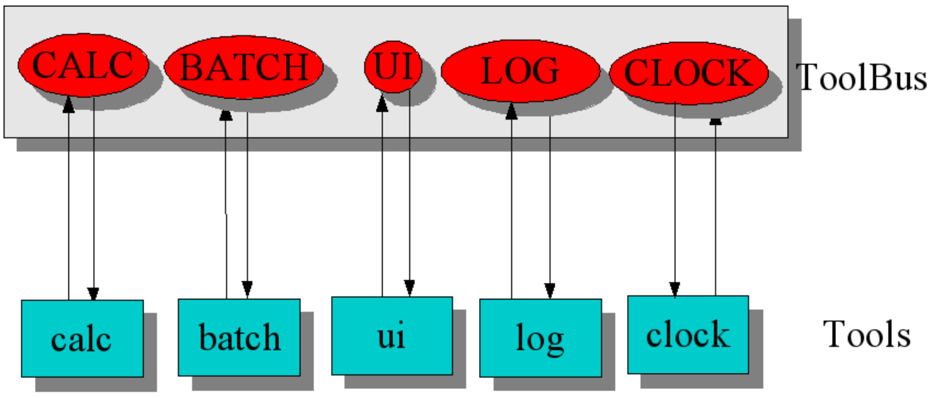
We are now ready to have a look at some larger examples of Tscripts.

Calculator Example

The calculator example illustrates how a calculator tool that can compute simple arithmetic expressions is shared by cooperating processes. The overall architecture is shown in Figure 1.6, “Architecture of the clock application” [15]. The application consists of the following 5 processes:

- `CALC`: the calculator *process* that regulates the access to the calculator *tool* `calc`, see Example 1.4, “Process `CALC` and tool `calc`” [16].
- `BATCH`: a batch process that uses the tool `batch` to read an expression from file, calculate its value and write the result back to file, see Example 1.5, “Process `BATCH` and tool `batch`” [16].
- `UI`: a user-interface process that uses the tool ui to allow a user to enter an expression and get its value back, see Example 1.6, “Process `UI` and tool `ui`” [17]. Observe that the processes `BATCH` and `UI` are both competing for the shared resource calculator (implemented by the process `CALC` and the tool `calc`).
- `LOG`: a logging process that maintains a log of all calculations that have been performed by the application, see Example 1.11, “Process `LOG` and tool `log`” [19].
- `CLOCK`: a clock process that uses the tool `clock` to provide the current time, see Example 1.13, “Process `CLOCK` and tool `clock`” [20].

Figure 1.6. Architecture of the clock application



The global structure of the Tscript `calc.tb` is sketched in Example 1.3, “Global structure of `calc.tb`” [15].

Example 1.3. Global structure of `calc.tb`

```
process CALC is ...
tool calc is ...
     See Example 1.4, “Process CALC and tool
          calc” [16]

process BATCH is ...
tool batch is ...
     See Example 1.5, “Process BATCH and tool
          batch” [16]

process UI is ...
     See Example 1.6, “Process UI and tool
          ui” [17]

process CALC-BUTTON is ...
     See Example 1.7, “Process CALC-BUTTON” [18]

process LOG-BUTTON is ...
     See Example 1.8, “Process LOG-BUTTON” [18]

process TIME-BUTTON is ...
     See Example 1.8, “Process LOG-BUTTON” [18]

process QUIT-BUTTON is ...
     See Example 1.10, “Process QUIT-BUTTON” [19]

process LOG is ...
     See Example 1.11, “Process LOG and tool
          log” [19]

process CLOCK is ...
     See Example 1.13, “Process CLOCK and tool
          clock” [20]

toolbus(CALC, BATCH, UI, LOG, CLOCK)
     See Example 1.14, “ToolBus configuration for calculator demo” [20]
```

Example 1.4. Process `CALC` and tool `calc`

```
process CALC is
    let Tid : calc, E : str, V : term
    in
         execute(calc, Tid?). ❶
          (❷
            rec-msg(compute, E?) . ❸
            snd-eval(Tid, expr(E)) . rec-value(Tid, val(V?)) . ❹
            snd-msg(compute, E, V) . snd-note(compute(E, V))  ❺
          )* delta ❻
    endlet

tool calc is { command = “calc”}
```

Notes:

❶ Execute the `calc` tool. The tool identifier is assigned to the variable `Tid`, that is of type `calc`.

❷ Begin of endless loop.

❸ Receive a compute message.

❹ Send an evaluation request to the `calc` tool and receive its value back.

❺ Send a reply to the original compute request. By convention, the original message is included in the reply. Also send a note regarding this (expression,result) pair for the sake of logging.

❻ End of the endless loop.

Example 1.5. Process `BATCH` and tool `batch`

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

tool batch is {command = “batch”}
```

Notes:

❶ Send an evaluation request to the `batch` tool and receive an expression back.

❷ Communicate with the `CALC` process (and thus with the `calc` tool) to get the expression evaluated.

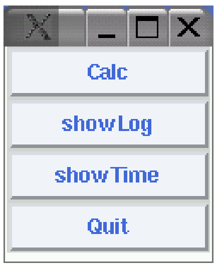
❸ Send the result back to the `batch` tool.

The user-interface is shown in Figure 1.7, “The calc GUI” [17] and behaves as follows:

- When the user presses Calc, a dialog window appears to enter an expression. The result is shown in a separate window. See Figure 1.8, “Dialog resulting from `CALC-BUTTON`” [18].

- Pressing showLog displays all calculations so far.
- Pressing showTime displays the current time in a separate window.
- Pressing Quit end the application.

Figure 1.7. The calc GUI



Example 1.6. Process `UI` and tool `ui`

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

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

Notes:

❶ `CALC-BUTTON` and `LOG-BUTTON` are mutually exclusive and they can be activated indefinitely.

❷ `TIME-BUTTON` is independent and can also be repeated indefinitely.

❸ `QUIT-BUTTON` is also independent but be activated only once (for obvious reasons).

Also observe the extensive use of named process expressions like, for instance, `CALC-BUTTON` to give an high-level overview of the `UI` process. See Example 1.7, "Process `CALC-BUTTON`"[18], Example 1.8, "Process `LOG-BUTTON`"[18], Example 1.9, "Process `TIME-BUTTON`"[19], and Example 1.10, "Process `QUIT-BUTTON`" [19] for their definitions.

Example 1.7. Process `CALC-BUTTON`

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

Notes:

❶ The Calc button is pressed; the `ui` tool generates an event.

❷ Ask the `ui` tool for an expression, see Figure 1.8, “Dialog resulting from `CALC-BUTTON`” [18] for examples.

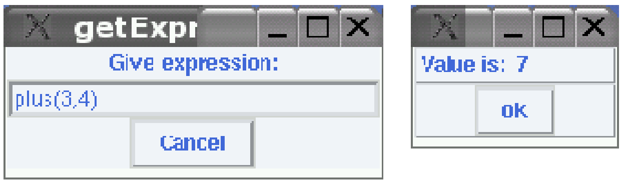
❸ The user cancels the dialog; no further actions are needed.

❹ The user has entered an expression. Communicate with the CALC process to compute a value.

❺ Ask the `ui` tool to display the value.

❻ Acknowledge the event to the tool.

Figure 1.8. Dialog resulting from `CALC-BUTTON`



Give expression:

plus(3,4)

Value is: 7

Example 1.8. Process `LOG-BUTTON`

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

Example 1.9. Process `TIME-BUTTON`

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

Example 1.10. Process `QUIT-BUTTON`

```
process QUIT-BUTTON(Tid : ui) is
        rec-event(Tid, button(quit)) .
        shutdown("End of calc demo")
```

Example 1.11. Process `LOG` and tool `log`

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

Notes:

❶ Receive a note about a computation that has taken place and log it.

❷ Receive a message to show the value of the current log. Retrieve it from the `log` tool and return the result.

An alternative way to describe the LOG process is shown in Example 1.12, "Process `LOG1`: maintaining the log inside the ToolBus"[19]. Instead of running a separate log tool, the process LOG1 maintains the log in a local process variable TheLog. See the section called "*Built-in functions*" [52] for a description of the function `join` that is used this example.

Example 1.12. Process `LOG1`: maintaining the log inside the ToolBus

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

Example 1.13. Process **CLOCK** and tool **clock**

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

The complete ToolBus configuration that describes the start of the calculator demo is shown in Example 1.14, “ToolBus configuration for calculator demo”[20]. It starts the mentioned processes in parallel and from that moment on user-interaction and the activities of the BATCH process will drive the execution.

Example 1.14. ToolBus configuration for calculator demo

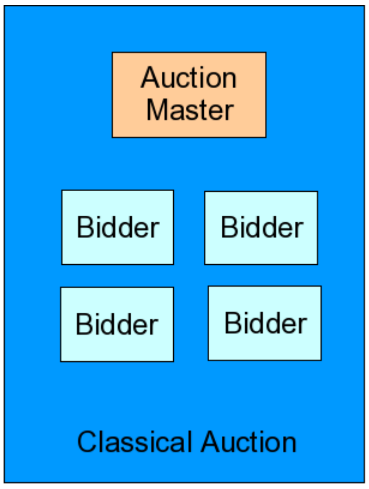
```
toolbus (CALC, BATCH, UI, LOG, CLOCK)
```

Auction Example

In the *classical auction*, the auction master and all bidders are in the same room and interact with each other according to a fixed protocol. It is shown in Figure 1.9, “Classical auction”[20]. The steps in the protocol are:

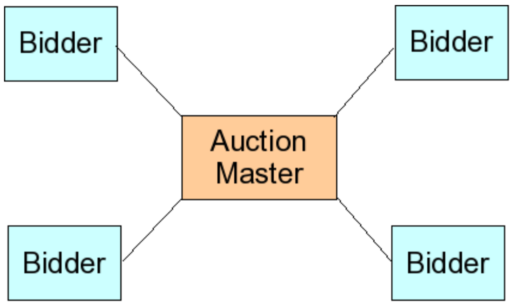
1. The auction master introduces a new item for sale and sets an initial price for it.
2. Next, bidders raise their hand and shout a new bid that is to be acknowledged by the auction master.
3. Step 2 is repeated as long as new bids come in.
4. When no new bids are being made, the auction master asks for "any higher bid?" and waits during a fixed period.
5. If no new bids come in during this period, the auction master declares the item for sale to be sold to the highest bidder.
6. If a new bid comes in during this period, the procedure continues with step 2.

Figure 1.9. Classical auction



A striking aspect of the classical auction is that the auction master and the bidders can *see* each other. This is a great communication and synchronization tool. In the case of a *distributed auction*, auction master and bidders are on different locations and can only communicate via the Internet, see Figure 1.10, “Distributed auction” [21].

Figure 1.10. Distributed auction

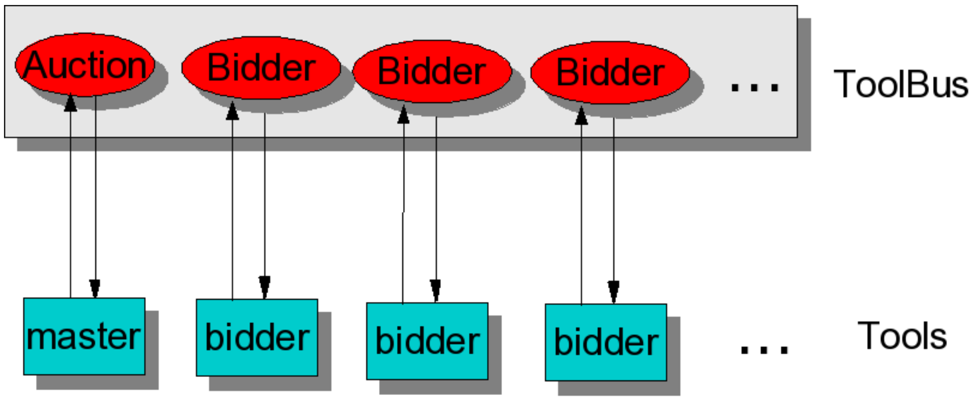


The communication and synchronization in a distributed auction has to be described explicitly and requires answers to questions like:

- How are bids synchronized?
- How to inform bidders about the highest bid?
- How to decide when bidding is over and the item is to be sold?
- How to handle bidders that come and go during the auction?

The auction application to be described answers these questions and has an architecture as shown in Figure 1.11, “Architecture of the auction example” [21].

Figure 1.11. Architecture of the auction example



The auction application consists of a variable number of processes:

- `Auction`: the process Auction orchestrates the complete auction and is controlled by the tool `master` that enables the auction master to offer new items for sale and to monitor the progress of the auction.
- `Bidder`: for each new bidder that enters the auction a new process `Bidder` and tool `bidder` are created. The tool bidder keep the user informed and allows her or him to submit new bids.

The global structure of the Tscript auction.tb is sketched in Example 1.15, “Global structure of `auction.tb`” [22].

Example 1.15. Global structure of `auction.tb`

```
process Auction is ...
     See Example 1.16, “Process Auction” [22]

tool master is ...

process ConnectBidder is ...
     See Example 1.17, “Process ConnectBidder” [23]

tool batch is ...

process OneSale is ...
     See Example 1.18, “Process OneSale” [24]

process Bidder is ...
     See Example 1.22, “Process Bidder” [27]

toolbus(Auction)
```

Example 1.16. Process `Auction`

```
process Auction is
  let Mid : master, Bid : bidder
  in
      execute(master, Mid?) .       (1)
      ( ConnectBidder(Mid, Bid?)    (2)
      +
        OneSale(Mid)
      ) *
      rec-event(Mid, quit) .         (3)
      shutdown("Auction is closed")  (4)
  endlet


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

Notes:

(1) Execute the `master` tool.

(2) Repeat:

- Add new bidders between sales, or
- Perform one sale.

(3) Until auction master quits

(4) Close the auction application.

Example 1.17. Process `ConnectBidder`

```
process ConnectBidder(Mid : master, Bid : bidder?) is
  let Pid : int, Name : str
  in
     rec-connect(Bid?) .                ❶
     create(Bidder(Bid), Pid?) .        ❷
     snd-eval(Bid, get-name) .          ❸
     rec-value(Bid, name(Name?)) .
     snd-do(Mid, new-bidder(Bid, Name))❹
  endlet
```

Notes:

❶ Receive a connection request from a new `bidder` tool.

❷ Create a new `Bidder` process that orchestrates the behaviour of this bidder.

❸ Ask `bidder` for its name.

❹ Send the name of the bidder to the `master` tool.

Example 1.18. Process `OneSale`

```
process OneSale(Mid : master) is
  let Descr : str,                  ❶
      InAmount : int,               ❷
      Amount : int,                 ❸
      HighestBid : int,             ❹
      Final : bool,                 ❺
      Sold : bool,
      Bid : bidder                  ❻
  in rec-event(Mid, new-item(Descr?, InAmount?)) . ❼
      HighestBid := InAmount .
      snd-note(new-item(Descr, InAmount)) .        ❽
      Final := false . Sold := false .
      (
         Here is the main logic of OneSale          ❾
      ) * if Sold then
             snd-ack-event(Mid, new-item(Descr, InAmount))
          fi
  endlet
```

Notes:

❶ `Descr` contains a textual description of the current item for sale.

❷ `InAmount` is the initial amount asked for the item.

❸ `Amount` is the value of the current bid.

❹ `HighestBid` is the highest bid so far for this item.

❺ Two Boolean values control the logic of the bidding process. `Final` is true when the call for final bids has been issued and `Sold` is true when the item has been sold.

❻ Bidder is a new bidder tool that has connected during the sale.

❼ The auction master wants to initiate the sale of a new item.

❽ Inform all connected bidders about the new item that is for sale.

❾ The detailed logic is explained in Example 1.19, “Process `OneSale`, main logic”[25], Example 1.20, “Process `OneSale`, handling one bid”[25], and Example 1.21, “Process OneSale, handling other cases” [26].

Example 1.19. Process `OneSale`, main logic

```
( if not(Sold) then ... ❶ fi
+ if not(or(Final, Sold)) then ... ❷fi
+ if and(Final, not(Sold)) then ... ❸fi
+ ConnectBidder(Mid, Bid?) ... ❹
) * if Sold then ... fi
```

The main process logic consists of four parts:

❶ Handle one incoming bid, see Example 1.20, “Process `OneSale`, handling one bid” [25].

❷ Start the "any higher bid" procedure, see Example 1.21, “Process OneSale, handling other cases” [26].

❸ Sell the item when no further bids are received, see Example 1.21, “Process OneSale, handling other cases” [26].

❹ Connect a bidder during the sale, see Example 1.21, “Process OneSale, handling other cases” [26].

Example 1.20. Process `OneSale`, handling one bid

```
( if not(Sold) then
    rec-msg(bid(Bid?, Amount?)) .          ❶
    snd-do(Mid, new-bid(Bid, Amount)) .    ❷
    if less-equal(Amount, HighestBid) then
        snd-msg(Bid, rejected)             ❸
    else
        HighestBid := Amount .             ❹
        snd-msg(Bid, accepted) .           ❺
        snd-note(update-bid(Amount)) .     ❻
        snd-do(Mid, update-highest-bid(Bid, Amount)) . ❼
        Final := false
    fi
  fi
+ if not(or(Final, Sold)) then    ... fi
+ if and(Final, not(Sold)) then ... fi
+ ConnectBidder(Mid, Bid?) ...
) * if Sold then ... fi
```

Notes:

❶ Receive a bid from a bidder.

❷ Inform the auction master about the new bid.

❸ Reject the bid when it is too low.

❹ Remember this bid as the highest bid so far.

❺ Inform the bidder that his bid is accepted.

❻ Inform all connected bidders that there is higher bid.

❼ Update the status of the auction master.

Example 1.21. Process OneSale, handling other cases

```
( if not(Sold) then ... fi
+ if not(or(Final, Sold))  ❶ then
      snd-note(any-higher-bid) delay(sec(10)) .    ❷
      snd-do(Mid, any-higher-bid(10)) .
      Final := true                                ❸
  fi
+ if and(Final, not(Sold)) ❹ then
      snd-note(sold(HighestBid)) delay(sec(10)) .  ❺
      Sold := true                                 ❻
   fi
+ ConnectBidder(Mid, Bid?) .                       ❼
   snd-msg(Bid, new-item(Descr, HighestBid)) .     ❽
   Final := false                                  ❾
) * if Sold then ... fi
```

Notes:

❶ The item is not yet sold, but we have not yet asked for a final bid.

❷ Wait for 10 seconds and then ask for final bids.

❸ Inform the auction master and remember that we asked for final bids.

❹ The item is not yet sold but we have already asked for final bids.

❺ Wait another 10 seconds and inform all bidders that the item has been sold.

❻ Record that the item is sold.

❼ During the sale a new bidder wants to connect.

❽ Inform the new bidder about the progress of the auction.

❾ Restart the bidding procedure; this overrules the call for final bids.

Example 1.22. Process `Bidder`

```
process Bidder(Bid : bidder) is
  let Descr : str,  Amount : int,  Acceptance : term
  in
     subscribe(new-item(<str>, <int>)) .          ❶
     subscribe(update-bid(<int>)) .
     subscribe(sold(<int>)) .
     subscribe(any-higher-bid) .
     ( ( rec-msg(Bid, new-item(Descr?, Amount?)) ❷
         + rec-note(new-item(Descr?, Amount?))    ❸
         + rec-disconnect(Bid) ❹. delta
        ) .
       snd-do(Bid, new-item(Descr, Amount)) .     ❺
       ( rec-event(Bid, bid(Amount?)) .           ❻
         snd-msg(bid(Bid, Amount)) .              ❼
         rec-msg(Bid, Acceptance?) .
         snd-do(Bid, accept(Acceptance)) .
         snd-ack-event(Bid, bid(Amount))
       )*
     + rec-note(update-bid(Amount?)) .            ❽
       snd-do(Bid, update-bid(Amount))
     + rec-note(any-higher-bid) .
       snd-do(Bid, any-higher-bid)
     + rec-disconnect(Bid) . delta                ❾
     ) *
       rec-note(sold(Amount?)) .                  🅐
       snd-do(Bid, sold(Amount))
     )* delta
  endlet
```

Notes:

❶ Subscribe to all relevant notes.

❷ Get information about the current item for sale, directly after connecting.

❸ Get information about the current item for sale during regular progress of the auction.

❹ Disconnect between sales.

❺ Inform the bidder tool by updating the information about the item for sale.

❻ This bidder want to bid on the current item for sale.

❼ Pass this bid on and await its acceptance. The result is returned to the bidder tool and the event is acknowledged.

❽ Handle the informative notes about the update of the highest bid and the request for any higher bids.

❾ Disconnect during a sale.

🅐 The item is sold.

Wave Example

Have you ever considered a (guitar) string that is attached at both ends and wondered how the movements of the strings can be simulated? Although this is a completely atypical application of the ToolBus, it is fun to do so we will delve into the details.

In mathematical physics, the vibrating string is described by the so-called one-dimensional wave equation that describes a discrete approximation of the continuous string. The discretization is achieved by sampling the amplitude of the string at certain points $i = 1, \ldots N$, where N is the number of points. The amplitude at point i at time t can now be described by $y_{i+1}(t)$ (see also Figure 1.12, “One-dimensional wave equation” [28]) that is defined as follows:

$$y_i(t+\#t) = F(y_i(t), y_i(t-\#t), y_{i-1}(t), y_{i+1}(t))$$

In other words, the amplitude at point i and time t depends on:

- the current amplitude,
- the previous amplitude at this point,
- the current amplitude of the left neighbour, and
- the current amplitude of the right neighbour.

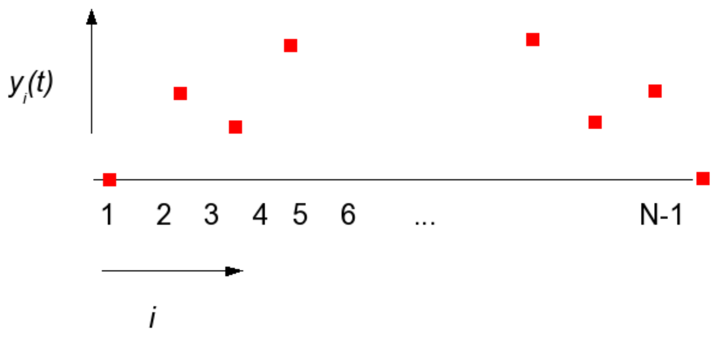
It also depends on the function F defined as follows:

$$F(z_1,z_2,z_3,z_4) = 2z_1 - z_2 + (c\,\#t/\#x)^2\ (z_3\ -\ 2z_1 +\ z_4)$$

where

- Δx is the (small) interval between sampling points, and
- c is a constant representing the propagation velocity of the wave.

Figure 1.12. One-dimensional wave equation



After these preparations, we have to define the architecture of a ToolBus application that can simulate the behaviour of a string. The key idea is to use a separate ToolBus process to represent the behaviour of each sampling point. The architecture is shown in Figure 1.13, “Architecture of the wave example” [29] and consists of the following processes and tools:

- Process `Pend` models an end point of the string, see Example 1.26, “Process `Pend`”[30] .Two instances are used to model the left and right end point.

- Process P models one sampling point, see Example 1.27, “Process P”[31]. *N*-1 instances are used to model all intermediate points.
- The auxiliary process F computes the function *F* discussed above, see Example 1.28, “Auxiliary process F” [32].
- Process makeWave constructs *N* connected instances of processes P and two end points Pend, see Example 1.24, “Process MakeWave” [29].
- The tool display visualizes the simulation.

Figure 1.13. Architecture of the wave example

The global structure of the Tscript wave.tb is shown in Example 1.23, “Global structure of wave.tb” [29].

Example 1.23. Global structure of wave.tb

```
process MakeWave ... See Example 1.24, “Process MakeWave” [29]
process Pend ...     See Example 1.26, “Process Pend” [30]
process P ...        See Example 1.27, “Process P” [31]
process F ...        See Example 1.28, “Auxiliary process F” [32]

toolbus(MakeWave(...))
```

Example 1.24. Process MakeWave

```
process MakeWave(N : int) is
  let Tid : display, Id : int, I : int, L : int, R : int
  in
     execute(display, Tid?) .         (1)
     snd-do(Tid, mk-wave(N)) .

     create(Pend(Tid, 0, 1), Id?).    (2)
     L := sub(N,1) .
     create(Pend(Tid, N, L), Id?) .

     I := 1 .                         (3)
     if less(I, N) then
        L := sub(I, 1) . R := add(I, 1) .
        create(P(Tid, L, I, R, 1.0, 1.0), Id?) .
        I := add(I, 1)
     fi *

     shutdown("end") delay(sec(60))  (4)
  endlet

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

Notes:

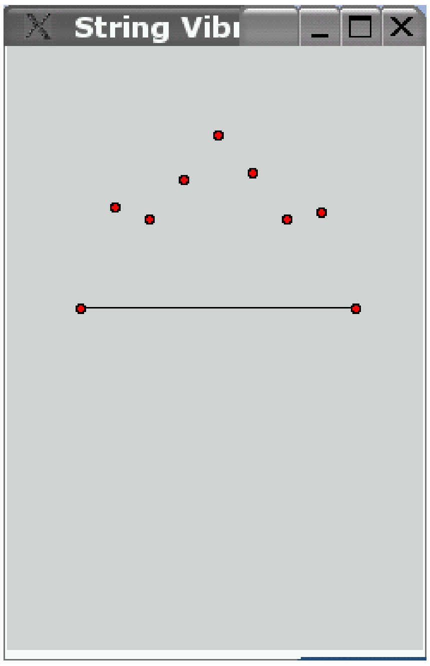
(1) Execute the display tool and initialize it to show *N* points. Figure 1.14, “User-interface of wave demo” [30] shows the display tool in action during a later stage of the simulation.

(2) Create the two end points with index 0 and *N*.

(3) Create the intermediate points 1,..., *N*-1 in a loop.

(4) Run the demo for one minute.

Figure 1.14. User-interface of wave demo



Example 1.25. ToolBus configuration for wave

```
toolbus(MakeWave(8))
```

Example 1.26. Process Pend

```
process Pend(Tid : display, I ❶: int, NB ❷: int) is
  let W : real
  in
   ( rec-msg(NB, I, W?) || snd-msg(I, NB, 0.0) || ❸
     snd-do(Tid, update(I, 0.0))                 ❹
   ) * delta
  endlet
```

Notes:

❶ `I` is the index of this end point.

❷ `NB` is the index of the neighbouring point.

❸ Receive the amplitude of the neighbour and send our own zero amplitude to the neighbour. Since a parallel operator `||` is used, these communications can appear in any order.

❹ Display the zero amplitude of this end point on the display.

Example 1.27. Process P

```
process P(Tid : display, ❶
          L : int,       ❷
          I : int,       ❸
          R : int,       ❹
          Dstart : real, ❺
          Estart : real  ❻
         ) is
  let AL : real, AR : real, D : real, D1 : real, E : real
  in
     D := Dstart . E := Estart .
     ( (  rec-msg(L, I, AL?)        ❼
       || rec-msg(R, I, AR?)
       || snd-msg(I, L, E)          ❽
       || snd-msg(I, R, E)
       || snd-do(Tid, update(I, E)) ❾
       ) .
       D1 := E .
       F(E, D, AL, AR, E?) .        Ⓐ
       D := D1
     ) * delta
  endlet
```

Notes:

❶ `Tid` is the tool identifier of the display tool.

❷ `L` is the index of the left neighbour of this point.

❸ `I` is the index of this point.

❹ `R` is the index of the right neighbour of this point.

❺ `Dstart` is the previous amplitude of this point.

❻ `Estart` is the current amplitude of this point.

❼ Receive amplitudes from our neighbours.

❽ Send our own amplitude to our neighbours.

❾ Show current amplitude on the display.

Ⓐ Compute a new value for our amplitude by applying `F`.

Example 1.28. Auxiliary process F

```
process F(Z1 : real, Z2 : real, Z3 : real, Z4 : real, Res : real?) is
  let CdTdX2 : real
  in (1)
     CdTdX2 := 0.01 . (2)
     Res := radd(rsub(rmul(2.0, Z1), Z2), (3)
                 rmul(CdTdX2, (4)
                      radd(rsub(Z3, rmul(2.0, Z1)), Z4))) (5)
  endlet
```

Notes:

(1) Recall that we are computing $2z_1 - z_2 + (c\ \Delta t/\Delta x)^2\ (z_3 - 2\ z_1 + z_4)$ and that the main challenge is to write this formula in prefix form.

(2) Take an arbitrary (small) value for $(c\ \Delta t/\Delta x)^2$.

(3) $2z_1 - z_2 + \ldots$

(4) $(c\ \Delta t/\Delta x)^2 * \ldots$

(5) $\ldots + (z_3 - 2z_1 + z_4)$

Executing ToolBus and tools

The ToolBus interpreter (**toolbus**) and all tools have some standard program arguments in common, but they have some specific arguments as well. In this section we describe all possible program arguments and the way to execute **toolbus** and tools.

Common arguments

ToolBus and tools have the following optional arguments in common:

- `-help`: prints a description of all arguments of the ToolBus or tool.
- `-P`*`port_name`*: defines the "well known socket" *`port_name`* to which all tools temporarily connect in order to set up their own private socket that connects them permanently to the ToolBus interpreter. When omitted, socket 8998 will be used in case of the C ToolBus and a random free port in case of the ToolBus NG.

  Warning

  Not yet implemented.

Note that explicit arguments defining the sockets are *only* needed when several C ToolBus interpreters are running simultaneously on the *same* host machine.

ToolBus arguments

The *`script_name`* (see below) given as argument to the ToolBus is always preprocessed by a preprocessor before it is parsed as a Tscript. In this way, directives like, e.g., `#define`, `#include` and `#ifdef` can be used freely in Tscripts. The following preprocessor arguments are accepted by the **ToolBus** command:

- `-I`*`dir`*: append directory *`dir`* to the list of directories searched for include files.
- `-D`*`name`*: defines *`name`* with the string `"1"` as its definition.

- `D`*`name`*`=`*`defn`*: defines *`name`* with *`defn`* as definition.

Other arguments specific for the ToolBus command are:

Warning

The following arguments will probably be supported differently in ToolBusNG.

- `-viewer`: execute the ToolBus viewer that enables step-by-step execution and inspection of the state of each process state.
- `-gentifs`: only generate tool interfaces for all tools used in the script in a language independent format. For a script file named `script.tb` the tool interfaces are written to `script.tifs`. Do not execute the script.

  ### Warning

  Not yet implemented.

- `-fixed-seed`: use a fixed seed for the random generator used by the interpreter for scheduling processes and selecting alternatives in processes. By default, the random generator is initialized with the current time the **ToolBus** command is given. Using the `-fixed-seed` option makes the execution of the script reproducible across multiple runs of the **ToolBus** command.

  ### Warning

  Not yet implemented.

- `-S`*`script_name`*: any other argument is the name of the ToolBus script to be interpreted.

As an example, consider first

```
toolbus -Shello.tb
```

which starts interpreting the script `hello.tb`. Next, consider

```
toolbus -Imy-include-dir -DCNT=33 -Swave.tb
```

which searches the directory `my-include-dir` for files used in `#include` directives in the script `wave.tb` and it will define the name `CNT` with value `33`. All occurrences of `CNT` in the script will be replaced by this value before parsing it as a Tscript. Finally,

```
toolbus -gentifs -Shello.tb
```

produces the tool interfaces file `hello.tifs`.

Tool arguments

Warning

This section needs some work.

Arguments specific for tools are:

- `-TB_HOST` *`host_name`*: defines the host machine *`host_name`* on which the ToolBus interpreter is running and to which the tool should be connected. When omitted, the ToolBus interpreter should be running on the same host as the tool.
- `-TB_TOOL_NAME tool_name`: the tool name as defined in the Tscript (added automatically, when a tool is executed by the ToolBus).
- `-TB_TOOL_ID` *`Id`*: internal tool identifier of this tool execution (added automatically, when a tool is executed by the ToolBus).

The execution of a tool can start in two ways:

- The tool is started by an `execute` command in the Tscript.
- The initiative to execute the tool is taken outside the ToolBus. This requires that the script contains a `rec-connect` for this particular tool.

When ToolBus and tool are running on different host machines, it is important to define the host machine on which the ToolBus interpreter is running when starting the execution of the tool. As an example, consider the **hello** application described in Example 1.2, “hello2.tb” [10]. The **hello** tool will be executed by the ToolBus using the command

```
hello -P8998 -TB_HOST host1.institute.nl
```

when running on machine `host1.institute.nl`. Suppose, we replace the explicit `execute` in Example 1.2, “hello2.tb” [10] by a `rec-connect` as shown in Figure~\ref{fig:hello3.tb}. We may then manually start the **hello** tool by typing

```
hello
```

where we use the default values for the input/output sockets and assume that tool and ToolBus interpreter are both running on the same host (i.e., `host1.institute.nl`). Starting the execution from *another* host is achieved by typing (on, say, `host2.institute.nl`):

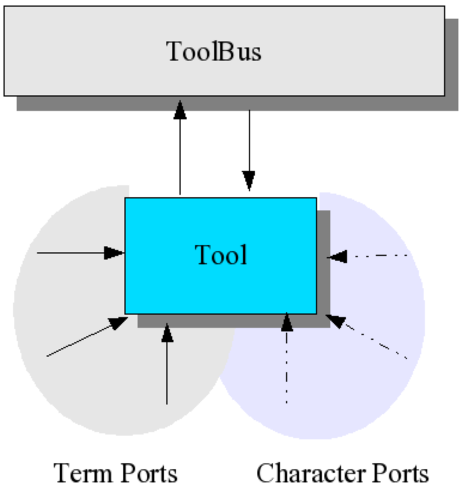
```
hello -P8998 -TB_HOST host1.institute.nl
```

ToolBus tools

There are some general issues to understand about ToolBus tools and we cover them here. First, the global structure of a tool is explained in the section called “*The global structure of a ToolBus tool*” [34]. Next, we describe how tool adapters work in the section called “*Adapters for tools and languages*” [35]. Finally, we cover in the section called “*Automatic generation of tool interfaces*” [36] the automatic generation of tool interfaces that is needed for some tool implementation languages .

The global structure of a ToolBus tool

Figure 1.15. Global tool organization



In its simplest form, a tool is a box connected via an input and an output port to a ToolBus. In the most general case, a tool has

- one input port from the ToolBus to the tool and can receive tree structures (terms) via this port;

- one output port from the tool to the ToolBus and can send terms to the ToolBus via this port;
- zero or more *term ports* to receive terms from other sources;
- zero or more *character ports* to receive character data from other sources.

This global, architectural, structure of a tool is shown in Figure 1.15, “Global tool organization” [34]. With each input port, an *event handler* is associated that takes care of the processing of the data received via that port and is responsible for returning a result (if any). One tool may thus contain several event handlers. When a request is received, the following steps are taken:

- The data received are parsed to check that they form a legal ToolBus term *T*. (If this is impossible, a warning message is generated).
- The event handler is called with *T* as argument.
- The event handler can do arbitrary processing needed to decompose *T*, to determine what has to be done, and perform any desired computation.
- The event handler returns either:
  - a legal ToolBus term representing a reply to be sent back to the ToolBus.
  - `NULL` indicating that there is no reply.

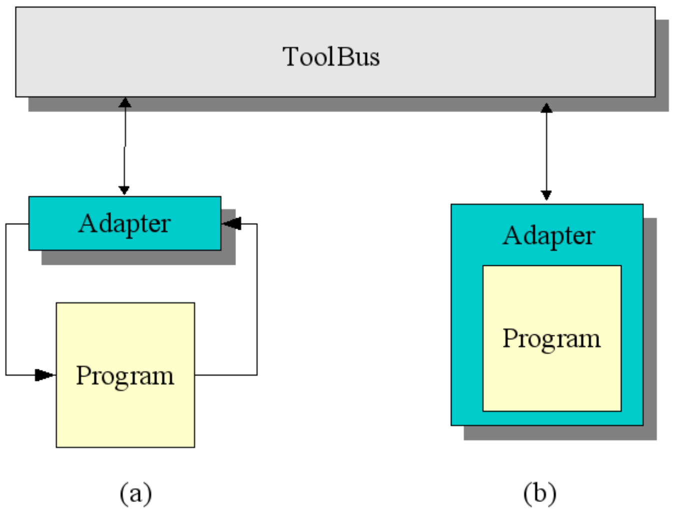
The global mode of operation of a tool is now:

- receive data on any input port and respond to this by sending some term (or `NULL`) to the ToolBus; or
- take the initiative to send a term to the ToolBus (typically to inform the ToolBus about some external event).

A tool is thus on the one hand a reactive engine that responds to a request from the ToolBus and returns the result back to the ToolBus in the form of a term (e.g., calculate the value of some expression), but on the other hand it can also take the initiative to send a term to the ToolBus (e.g., generate an event when a user pushes some button).

Adapters for tools and languages

Figure 1.16. Two organizations of a tool adapter



The main purpose of adapters is to act as small *wrappers* around existing programs or programming languages in order to transform them into tools that can be connected to the ToolBus. There exist two global strategies for constructing adapters:

- The adapter and the program to be adapted are executed as separate (Unix) processes. This structure is sketched in Figure 1.16, “Two organizations of a tool adapter”[35]. The advantage of this approach is that no access is needed to the source code of the program: it can remain a black box. Another advantage is that adapters may be reused for the adaptation of different programs. A possible disadvantage is some loss in efficiency.

  In this category a further subdivision is possible:

  - The program is executed once as a child process of the adapter and all `snd-eval`/`snd-do` requests are directed to this child process. The program can thus maintain an internal state between requests.
  - The same program is executed as a child process of the adapter for each `snd-eval`/`snd-do` request.
  - A different program is executed as a child process of the adapter for each `snd-eval`/`snd-do` request.

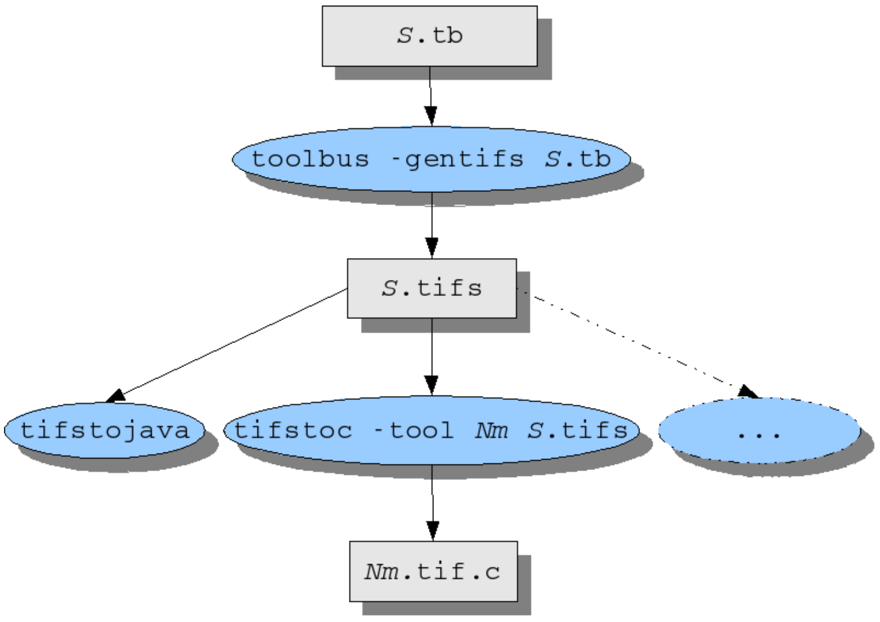
- Integrate the adapter and the software to be adapted into a single (Unix) process. This approach permits the most detailed adaptation of the program and is also the most efficient solution. This approach leads, however, to potentially less reusable adapters than the previous approach.

In order to achieve some uniformity, the current collection of adapters have the following optional program arguments in common:

- `-cmd`: the (default) program to be executed by the adapter. All arguments of the adapter that follow `-cmd` are interpreted as the name and arguments of the program to be executed.
- All tool arguments, see the section called “*Tool arguments*” [33].

Automatic generation of tool interfaces

Figure 1.17. Automatic generation of tool interfaces



The interface code for each tool depends on the particulars of the Tscript in which it is used. Changing the number of arguments in an evaluation request to the tool, or adding a new request, requires making changes to the interface code that are easily forgotten and therefore error prone. Another observation is that the interface code for different tools has a lot in common. An obvious solution to both problems is to *generate* tool interfaces automatically, given a Tscript. This generation process is shown in Figure 1.17, “Automatic generation of tool interfaces” [36] and consists of two steps:

- Generate a language-independent description of all tool interfaces used in the script. This amounts to a static analysis of all tool communication in the script. It is achieved by using the `-gentifs` option of the ToolBus interpreter. For instance,

  ```
  toolbus -gentifs hello2.tb
  ```

  will create a file `hello2.tifs` containing the tool interfaces.

- Use the language independent interface description to generate a tool interface for a specific tool in a specific implementation language. The generator **tifstoc** exists for generating C tool interfaces. It is called as follows:

  ```
  tifstoc -tool Name TifsFile
  ```

  and generates a file named `Name.tif.c`. For the hello example, we would have, for instance:

  ```
  tifstoc -tool hello hello2.tifs
  ```

  The resulting file `hello.tif.c` is shown in Example 1.30, “The generated file `hello.tif.c`” [44].

In Figure 1.17, “Automatic generation of tool interfaces” [36] it is also shown how tool interface generators for *other* languages (e.g., Java, Cobol) fit into this scheme. In addition to **tifstoc**, we used to support the generation of Java interfaces by way of **tifstojava**. In the current ToolBus implementation, this is no longer necessary, see the section called “*Writing ToolBus tools in Java*” [46].

Writing ToolBus tools in C

Although ToolBus tools can be implemented in many languages (including Java, C++, Tcl/Tk, ASF+SDF, and others) we start explaining how tools can be written in C. In other languages identical notions will be used with only minor adjustments to language-specific features and limitations. Writing tools in C amounts to:

- ATerms: the essential data type that is used to exchange information between tool and TB

  Note

  Add ref to ATerms

- The global structure of a ToolBus tool, see the section called “*The global structure of a ToolBus tool*” [34].
- The ToolBus Application Programmer's Interface , see the section called “*The ToolBus API*” [38].
- Compiling ToolBus tools written in C, the section called “*Compiling ToolBus tools written in C*” [41].
- Generating tool interfaces with {\tt tifstoc}, the section called “*Automatic generation of tool interfaces*” [36].

The include file `atb-tool.h`

Each tool needs to include the file `atb-tool.h` which defines some basic types as well as the set of library functions available. It consists of

- An include of `<aterm1.h>`.
- Defines `ATBhandler`: the type of event handlers.
- Defines the prototypes of all library functions

The tool library `libATB.a`

When compiling tools, the library `libATB.a` must be specified in order to make the tool library available (using the `-lATB` option of the C compiler). It provides the following functions:

- `ATBinit`: tool initialization, see the section called “*`ATBinit`*” [38].
- `ATBconnect`: to connect the tool to the ToolBus, the section called “*`ATBconnect`*” [38].
- `ATBdisconnect`: to disconnect the tool from the ToolBus, the section called “*`ATBconnect`*” [38].
- `ATBeventloop`: a standard event loop for a tool, see the section called “*ATBeventloop*” [39].
- `ATB`postEvent send an event to the ToolBus, see the section called “*ATBpostEvent*” [39].

In the following section, we will describe these functions.

The ToolBus API

During the initialization of each tool, some preparations have to made before the tool can be properly connected to the ToolBus. These preparations include

- Defining the *name* of the tool as it is known from a tool declaration in a Tscript.
- Parsing standard program arguments that are passed to the tool when it is started.
- Creating a pair of socket connections with a ToolBus interpreter.
- Starting an event loop.

During execution of the event loop, the tool can either *receive* terms from the ToolBus or it can take the initiative to *send* terms to the ToolBus. It is thus possible for a tool to both respond to ToolBus requests and *asynchronously* send terms to the ToolBus.

`ATBinit`

The initialization of the ToolBus API is achieved by

```
int ATBinit(int argc, char *argv[], ATerm *bottomOfStack).
```

This initializes the ToolBus API as well as the ATerm library that is used by it.

The standard program arguments that are passed (via *`argc`* and *`argv`*) are fully described in the section called “*Executing ToolBus and tools*” [32]. Particularly important is that the tool is initialized with a proper name. It should be literally equal (including the case of letters) to a tool name as appearing in a tool declaration in the Tscript. This is important since the tool name will be used when the tool is connected to the ToolBus. Note that `ATBinit` also initializes the ATerm library (hence the *`bottomOfStack`* argument, see

Note

Ref to Section \ref{ATinit}) in ATerm manual.

The return value indicates whether or not the ToolBus host could be found: 0 indicates that all is well, and -1 indicates an error, in which case the standard variable `errno` of the C run-time system is set to indicate which error.

`ATBconnect`

A tool can be connected to the ToolBus using *`term ports`* that can be using for sending and receiving data in the form of complete terms. Two aspects of term ports are important: the input channel used for the actual data transfer and the *handler* that takes care of processing input terms when they arrive. The connection is established as follows:

```
int ATBconnect(char *toolname, char *host,
               int port, ATBhandler h);
```

Here, *`toolname`* is the tool name to be used, *`host`* is the machine where the ToolBus is executing, *`port`* is the file descriptor of the channel to be used, and *`h`* is the handler to associated with this connection. If value `NULL` is passed as *`toolname`* or *`host`*, default values are used that are taken from *`argv`* that was passed to `ATBinit`. The same is true when `-1` is passed as value for *`port`*. The return value of `ATBconnect` is either `-1` (failure) or a positive number (the connection succeeded and the result is the file descriptor of the resulting socket connection with the ToolBus). Handlers for term ports are functions from ATerm to ATerm and have the type:

```
ATerm some_handler(int conn, ATerm input)
```

The argument *`conn`* is the connection along which the input term was received and *`input`* is the actual term received. The term returned by the handler is the reply to be sent to the ToolBus in response to this input event, or `NULL` if no reply is needed. In this fashion, an arbitrary number of term input ports can be set up which will be read in parallel: as soon as a term arrives at one of the ports the associated handler is activated. A connection can be terminated as follows:

```
void ATBdisconnect(int conn)
```

where *`conn`* is a connection that has been created earlier using `ATBconnect`.

ATBeventloop

Many tools first establish a number of term ports and then enter an infinite loop that processes input events. The function

```
int ATBeventloop(void)
```

captures this idea. It never returns, unless something goes wrong. We can now give a skeleton that many tools have in common:

```
#include "my_tool.tif.c"

ATerm my_tool_handler(int conn, ATerm input)
{ ... handle input and return a term or NULL ...  }

int main(int argc, char *argv[]){
  ATerm bottomOfStack;

  ATBinit(argc, argv, &bottomOfStack);
  if(ATBconnect(NULL, NULL, -1,  my_tool_handler) >= 0){
     ATBeventloop();
  }else{
     fprintf(stderr, "my_tool: Could not connect to the ToolBus\n");
  }
  ATBeventloop();
  return 0;
}
```

ATBpostEvent

So far, we have seen primitives for tools that only receive terms from the ToolBus. In the case of events that are generated by a tool, a term needs to be sent from the tool to the ToolBus. This can be achieved using

```
int ATBpostEvent(int conn, ATerm term)
```

which sends *term* along the port *conn*. Failure is indicated by the return value `-1`. A typical usage is:

```
ATBpostEvent(conn, ATmake(button("ok"))).
```

ATBpostRequest

Tools can also send synchroneous requests to the ToolBus with:

```
ATerm ATBpostRequest(int conn, ATerm request)
```

This call will send the given request to the ToolBus and return the response as soon as it arrives.

Primitives for advanced control flow

Tool programming amounts, in essence, to event driven programming: most of the time a tool is awaiting the arrival of data on one of its ports and when the data are there, a reply is sent to the ToolBus by the handler associated with that port. In computation-intensive tools, the need may arise to check for the availability of incoming data from the ToolBus during computations. In those cases, ATBeventloop may not offer enough flexibility. More customized control flow can be achieved using the following functions. Observe that these function are parameterized with a specific ToolBus connection (as returned by `ATBconnect`) and can be used to handle situations where a single tool is connected with *more than one* ToolBus.

Checking if there is input awaiting on a ToolBus connection is done by:

```
ATbool ATBpeekOne(int conn)
```

This function returns `ATtrue` if incoming data from a ToolBus are available on the connection *conn*.

Similarly, the availability of data on *any* connection may be checked by:

```
int ATBpeekAny(void)
```

If input is waiting, the appropriate connection is returned. Otherwise `-1` is returned. The sequence of activities needed for handling (once) the data available from a specific connection is captured by the function

```
void ATBhandleOne(int conn)
```

This amounts to calling the handler associated with connection *conn* with the available data as input term. Similarly, the data from *any* connection is handled by

```
int ATBhandleAny(void)
```

which returns -1 if anything goes wrong.

Finally, the function

```
int ATBgetDescriptors(fd_set *set)
```

Gathers all ToolBus connection file descriptors in a single descriptor set. The return value indicates the maximum value of any descriptor in the set.

Control flow patterns

Given, the control flow primitives in the previous section, we can express various common control flow patterns.

The function `ATBeventloop` can be expressed with the primitives just introduced:

```
int ATBeventloop(void)
{ int conn;
  while(ATtrue)
  {
    n  = ATBhandleAny();
    if(n < 0)
      return -1;
  }
}
```

Another style mixes the handling of input from the ToolBus, with other computations:

```
   while(ATtrue)
   {
     if(n = ATBpeekAny() >= 0) /* if there is an incoming event */
        ATBhandleOne(n);       /* handle it                    */
     else {
        ...                    /* perform other computation    */
     }
   }
```

In some tools, a mixture of passively awaiting input and actively sending terms to the ToolBus can be seen. Using `ATBwriteTerm`, the most general global event loop of a tool becomes:

```
while(ATtrue)
{
... ATBwriteTerm(c1,e1); ...; ATBwriteTerm(cn,en); ...
  ATBhandleAny();
}
```

In other words, each iteration starts by sending zero or more terms to the ToolBus (using `ATBwriteTerm`) and ends with processing one event coming from some port (using `ATBhandleAny`). The Tscript being used should, of course, be able to receive such events.

Compiling ToolBus tools written in C

When compiling a tool written in C the following questions should be answered:

- Where is the include file `aterm1.h` (or `aterm2.h` if you use the more sophisticated parts of the ATerm library)?
- Where is the include file `atb-tool.h`?
- Where is the ATerm library `libATerm.a`?
- Where is the ToolBus API library `libATB.a`?
- Which other libraries are needed to compile the tool?

The answers to these questions are clearly system dependent. There are two strategies to answer them. Strategy 1: find the desired locations on your system and hard code them in the compilation command. This will lead to a call to the C compiler with the following arguments:

- `-I`*`dir-where-aterm1.h-is`*
- `-I`*`dir-where-ATB-tool.h-is`*
- `hello.c -o hello`
- `-L`*`dir-where-libATerm.a-is`*

- `-lATerm`
- `-L`*`dir-where-libATB.a-is`*
- `-lATB`
- other libraries.

Strategy 2: write a make file that encodes this information. As a result, the location information is hardwired in the make file rather than in a command that has to be repeated over and over again.

Automatic generation of C tool interfaces

As already explained in the section called “*Automatic generation of tool interfaces*”[36] tool interfaces can be generated from a given Tscript for a given tool name. The ToolBus can generate a language-independent `.tifs` file, when it is started with the `-gentifs` option. In the case of C, the command **tifstoc** generates a tool interface in C for use with the ATerm library. The generated interface consists of two files:

- a C source file (`hello2.tif.c` in the example below), and
- a C header file (`hello2.tif.h` in the example below).

In the header file a number of interface functions is declared, one for each element in the input signature of the tool. It is up to the writer of the tool to provide an implementation for these functions. The generated C file contains a handler function that analyzes incoming terms from the ToolBus, and delegates actual processing to the appropriate interface function.

We will use the Tscript `hello2.tb` shown earlier in Example 1.2, “hello2.tb”[10] and describe all the steps needed to write and compile the hello tool.

Step 1: generate tifs

Using the command:

```
toolbus -gentifs hello2.tb
```

we generate a file called `hello2.tifs`. It contains information amount the interfaces for all tools that are used in a given Tscript.

Warning

The -gentifs flag is not yet implemented.

Step 2: generate C tool interface

Using the command:

```
tifstoc -tool hello test.tifs
```

we generate two files:

- the header file hello.tif.h, see Example 1.29, “The generated header file `hello.tif.h`” [43].
- the source file `hello.tif.c`, see Example 1.30, “The generated file `hello.tif.c`” [44].

Example 1.29. The generated header file `hello.tif.h`

```
**
 * This file is generated by tifstoc. Do not edit!
 * Generated from tifs for tool 'hello' (prefix='')
 */

#ifndef _HELLO_H
#define _HELLO_H

#include <atb-tool.h>

/* Prototypes for functions called from the event handler */
ATerm get_text(int conn);
void rec_terminate(int conn, ATerm);
extern ATerm hello_handler(int conn, ATerm term);
extern ATerm hello_checker(int conn, ATerm sigs);

#endif
```

Only the functions `get_text` and `rec_terminate` together with a simple `main` function have to be implemented to build a fully functional ToolBus tool.

Example 1.30. The generated file `hello.tif.c`

```
/**
 * This file is generated by tifstoc. Do not edit!
 * Generated from tifs for tool 'hello' (prefix='')
 */

#include "hello.tif.h"

#define NR_SIG_ENTRIES  2


static char *signature[NR_SIG_ENTRIES] = {      ❶
  "rec-eval(<hello>,get_text)",
  "rec-terminate(<hello>,<term>)",
};

/* Event handler for tool 'hello' */
ATerm hello_handler(int conn, ATerm term)       ❷
{
  ATerm in, out;
  /* We need some temporary variables during matching */
  ATerm t0;

  if(ATmatch(term, "rec-eval(get_text)")) {
    return get_text(conn);
  }
  if(ATmatch(term, "rec-terminate(<term>)", &t0)) {
    rec_terminate(conn, t0);
    return NULL;
  }
  if(ATmatch(term,
             "rec-do(signature(<term>,<term>))", &in, &out)) {
    ATerm result = hello_checker(conn, in);
    if(!ATmatch(result, "[]"))
      ATfprintf(stderr,
                "warning: not in input signature:\n\t%\n\tl\n",
                result);
    return NULL;
  }

  ATerror("tool hello cannot handle term %t", term);
  return NULL; /* Silence the compiler */
}

/* Check the signature of the tool 'hello' */
ATerm hello_checker(int conn, ATerm siglist)    ❸
{
  return ATBcheckSignature(siglist, signature, NR_SIG_ENTRIES);
}
```

Notes:

❶ An array of signature definitions (`signature`) that contains the argument and return types of each interface function.

❷ The handler function (`hello_handler`), that differentiates between the different possible input terms coming from the ToolBus, and delegates the actual work to the appropriate function.

❸ The signature checker `hello_checker` uses the assembled signature information in the array `signature` and compares it with `siglist`, an ATerm that encodes the tool's signature as expected by the ToolBus.

Step 3: write main

As mentioned earlier, the only thing needed to implement the actual hello tool, is the implementation of the two interface functions `get_txt` and `rec_terminate`, and the implementation of `main` to get things going. We will first take a look at the initialization stuff that the `main` function of the hello tool has to do, see Example 1.31, “`main` function of hello tool” [45].

Example 1.31. `main` function of hello tool

```
#include <stdlib.h>
#include "hello.tif.h"

int main(int argc, char *argv[])
{
  ATerm bottomOfStack;                                       ❶

  ATBinit(argc, argv, &bottomOfStack);                       ❷
  if(ATBconnect(NULL, NULL, -1, testing_handler) >= 0) {     ❸
    ATBeventloop();                                          ❹
  } else {
    fprintf(stderr,
            "Could not connect to the ToolBus, giving up!\n");
    return -1;
  }
  return 0;
}
```

Notes:

❶ The variable `bottomOfStack` is needed by the ATerm library to determine where to look for the stack. It is passed as argument to `ATBinit`.

❷ The variables `argc` and `argv` are passed unchanged to `ATBinit`, so the ToolBus library can look for default values for things like the ToolBus' well-known socket address and the ToolBus host name.

❸ The call to `ATBconnect` connects to a running ToolBus, and requires four arguments: a character string representing the tool name, a character string representing the host name of the ToolBus to connect to, the port number of the ToolBus to connect to, and a handler function. Passing `NULL`, `NULL`, and -1 respectively as the tool name, the host name, and the port number cause the defaults for these values to be used instead.

❹ When all goes well, the call to `ATBeventloop` starts the main ToolBus eventloop and the tool will be ready to receive requests from the ToolBus.

Step 4: implement interface functions

Finally, we only need the implementation of the two interface functions `get_txt` and `rec_terminate`, see Example 1.32, “Implementation of interface functions of hello tool” [46]

Example 1.32. Implementation of interface functions of hello tool

```
ATerm get_text(int conn)                              ❶
{
  return
    ATmake(
     "snd-value(text(\"Hello World, my first tool in C!\n\"))"
    );
}

void rec_terminate(int conn, ATerm msg){              ❷
  exit(0);
}
```

Notes:

❶ get_text: generate the greeting text. The `conn` argument identifies the ToolBus connection, making it possible to distinguish *which* ToolBus made the request. This enables connecting to more than one ToolBus at the same time.

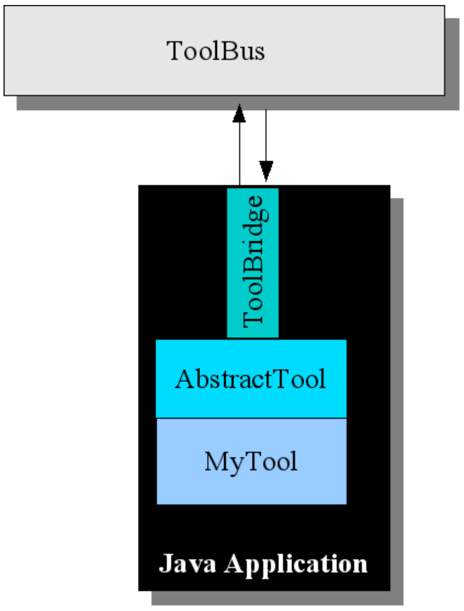
❷ Mandatory function that is called to terminate the tool.

Writing ToolBus tools in Java

Now we will show how ToolBus tools can be implemented in Java. The overall organization is shown in Figure 1.18, “Global organization of a tool implemented in Java”[46]. The actual communication between ToolBus and tool is taken care of by an instance of the class `ToolBridge` that takes care of low-level communication details. The ToolBridge is used by `AbstractTool`, an abstract class that defines the possible interactions between ToolBus and tool. The actual tool, in the figure `MyTool`, extends `AbstractTool`, gives implementations for its abstract methods, and implements tool-specific behaviour.

Compared to writing a tool in C, using Java is simpler because there is no need for generating a tool interface using a tif file.[2]

Figure 1.18. Global organization of a tool implemented in Java



[2]This is due to the use of Java reflection in the class ToolBridge. In older versions, a generation step was also needed for Java tools.

Before showing the Java implementation of the hello tool, we will first explain the AbstractJavaTool class.

The `AbstractJavaTool` class

```
public abstract class AbstractJavaTool implements IOperations{

public AbstractJavaTool(){ ... } ❶

public void connect(String[] args) throws Exception{ ... } ❷

public void connectDirectly(ToolBus toolbus, ClassLoader toolClassLoader, Strin

public ToolBridge getToolBridge(){ ... } ❹

public PureFactory getFactory(){ ... } ❺

public void sendEvent(ATerm aTerm){ ... }  ❻

public ATerm sendRequest(ATerm request){ ... } ❼

public void disconnect(ATerm aTerm){ ... } ❽

public abstract void receiveAckEvent(ATerm aTerm); ❾

public abstract void receiveTerminate(ATerm aTerm); 🅐
}
```

Notes:

❶ The default constructor of AbstractJavaTool.

❷ Connect to the ToolBus through a TCP/IP socket. The argument `args` contains the required information for running a tool (name, id, host and port of the ToolBus). An exception is thrown when something goes wrong during parsing of the arguments or while establishing the connection.

❸ Connects to the ToolBus directly. In this case all communication will go through method calls. Note however that this functionality is only available for tools that connect to the ToolBus on their own initiative; executed tools will always run in a seperate process.

❹ Returns a reference to the tool bridge that this tool instance is using.

❺ Returns a reference to the ATerm factory that is being used.

❻ Send an event to the ToolBus.

❼ Send a request to the ToolBus and returns the response as soon as it arrives.

❽ Send a disconnect request to the ToolBus. The argument `aTerm` gives additional information about the request.

❾ Receive an acknowledgement (in response to a previous event generated by this tool instance by way of `sendEvent`). The argument `aTerm` gives further details about the acknowledgement.

🅐 Receive a request from the ToolBus to terminate the execution of this tool instance.

Tool definition in the hello script

The hello script from Example 1.2, “hello2.tb” [10] can be used as is, except that the definition of the hello tool has to be changed to reflect the Java implementation:

```
tool hello is { kind = "javaNG" class = "toolbus.tool.java.hello.HelloTool"}
```

The hello example in Java

Example 1.33. Hello tool implemented in Java

```
package toolbus.tool.java.hello;

import toolbus.adapter.java.AbstractJavaTool;
import aterm.ATerm;
import aterm.ATermFactory;

public class HelloTool extends AbstractJavaTool{ ❶

  public HelloTool(){
    super();
  }

  public static void main(String[] args) throws Exception{
    HelloTool helloTool = new HelloTool();
    helloTool.connect(args); ❷
  }

  protected ATerm getText(){ ❸
    ATermFactory factory = getFactory();
    return factory.make("text(<str>)", "Hello world in Java!\n");
  }

  public void receiveAckEvent(ATerm aTerm){
    // Left blank intentionally.
  }

  public void receiveTerminate(ATerm msg){
    System.out.print("rec-terminate received: " + msg);
  }
}
```

Notes:

❶ The class `HelloTool` extends `AbstractJavaTool` and provides an implementation for the hello tool.

❷ The`HelloTool` then attempts to make a connection. The arguments of the constructor are passed to the `connect` call.

❸ The evaluation request `snd-eval(H, getText)` in the Tscript is implemented by the method `getText`. It constructs the required ATerm that `text("Hello world in Java!\n")` and returns it as result. This result will passed to the ToolBus and will be accepted by the atom `rec-value(H, text(S?))` in the Tscript. Note that in this example `H` is the tool identifier of the hello tool.

Writing ToolBus tools in other languages

Writing tools in Tcl/TK

Writing ToolBus tools in Tcl is greatly simplified by the **wish-adapter** to be explained in the section called "*wish-adapter*" [49]. Next, a small set of predefined Tcl functions is described that are always loaded by the **wish-adapter** and can be used in any Tcl script, see the section called "*Predefined Tcl functions*" [49]. Finally, we present the Tcl version of the hello tool in the section called "*The hello example in Tcl*" [50].

wish-adapter

The purpose of the wish-adapter is to execute Tcl/Tk's windowing shell **wish** as a tool. For instance,

```
wish-adapter -script calculator.tcl
```

executes **wish** as a TooBus tool and executes the Tcl script `calculator.tcl`.

In addition to the common tool arguments, wish-adapter has the following specific arguments:

- `-wish` *`Name`*: Use *`Name`* rather than **wish** as Tcl/Tk's windowing shell.
- `-lazy-exec`: Postpone execution of **wish** until needed.
- `-script`: The Tcl script to be executed.
- `-script-args`: The arguments for the Tcl script to be executed. These arguments are available to the Tcl script throught the variables *`argc`* and *`argv`*.

Various communication patterns are supported by wish-adapter. Communication is described here from the point of view of the ToolBus, i.e., `snd-` and `rec-` mean, respectively, send by ToolBus and receive by ToolBus. The communication patterns are:

- `snd-do(`*`Tid`*`, `*`Fun`*`(`A_1`, ..., `A_n`))` perform the Tcl function call *`Fun`*`(`A_1`, ...,`A_n`)`. Here *`Tid`* is a tool identifier (as produced by `execute` or `rec-connect`) for an instance of the **wish-adapter**.
- `snd-do(`*`Tid`*`, `*`Fun`*`(`A_1`, ...,`A_n`))`: perform the Tcl function call *`Fun`*`(`A_1`, ...,`A_n`)`. Here *`Tid`* is a tool identifier (as produced by `execute` or `rec-connect`) for an instance of the **wish-adapter**. Note that the function *`Fun`* must send an answer back to the ToolBus (using `TBsend "snd-eval(...)"`).
- `rec-value(`*`Tid`*`, `*`Res`*`)`: the return value for a previous evaluation request.
- rec-event(Tid, A_1`, ...,`A_n): event generated by wish.
- snd-ack-event(*`Tid`*, A_1): acknowledgement of a previously generated event.
- snd-terminate(*`Tid`*, A_1): terminate execution of wish-adapter.

The command **wish** is executed once, an initial Tcl script is read, and all further requests are directed to this incarnation of **wish**. A small set of Tcl procedures is available for unpacking and packing ToolBus terms (see below).

Predefined Tcl functions

The following Tcl functions are predefined and can be used freely in Tcl script executed via the wish-adapter:

- `TBstring` *`Str`*: converts a Tcl string to a ToolBus string by surrounding it with double quotes and escaping double quotes occurring inside *`Str`*.

- `TCLstring` *`Str`*: converts a ToolBus string into a Tcl string by removing surrounding double quotes.
- `TBlist` *`List`*: converts a Tcl list to a ToolBus list by separating the elements with commas and surrounding the list by curly braces.
- `TBerror` *`Msg`*: constructs an error message that can be sent to the ToolBus.
- `TBsend` *`Trm`*: send *`Trm`* back to the ToolBus.
- `TBevent` *`Event`*: send event `Event` to the ToolBus.
- `TBrequire` *`ToolName ProcName Nargs`*: check that the Tcl code for *`ToolName`* contains a procedure declaration for *`ProcName`* with *`Nargs`* formal parameters. This function is mainly used by the **wish-adapter** to check compatibility of the Tcl code with the expected input signature of the tool.

Warning

All communication between **wish-adapter** and a tool written in Tcl is done via standard input/output. Only use the standard error stream for print statements in the Tcl script, since using standard output will disrupt the communication with the ToolBus.

The hello example in Tcl

Writing the hello tool in Tcl requires two steps:

- Write the required Tcl code `hello.tcl`. The result is shown in Example 1.34, “hello.tcl: the hello tool in Tcl” [50].
- Replace hello's tool definition in `hello2.tb` (see Example 1.2, “hello2.tb” [10]) by:

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

Example 1.34. hello.tcl: the hello tool in Tcl

```
# hello.tcl -- hello tool in Tcl/Tk

proc get-text {} {
   TBsend "snd-value(text(\"Hello World tool in Tcl!\n\"))"
}

proc rec-terminate { n } {
   exit
}
```

Python

A Python adapter is only available for older versions of the ToolBus. It is currently not supported.

Perl

A Perl adapter is only available for older versions of the ToolBus. It is currently not supported.

Reference Information

The syntax of Tscripts

A Tscript may contain directives like, e.g., `#define`, `#include` and `#ifdef` that are replaced by a preprocessor similar to the C preprocessor. We summarize the most frequently used directives:

- `#define` *`Identifier Token-sequence`* causes the preprocessor to replace all occurrences of *`Identifier`* by *`Token-sequence`*.
- `#include "`*`Filename`*`"` will be replaced by the entire contents of the named file.
- `#ifdef` and `#ifndef` can be used for the conditional incorporation or exclusion of parts of a script.

The syntax of Tscripts (without preprocessor directives) is as follows:

Warning

This definition is slightly out-of-date.

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

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

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

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

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

        TERM                                  -> TYPE

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

        NAME                                     -> VNAME

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

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

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

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

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

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

Built-in functions

Tscripts provide a limited form of built-in functions that are summarized here. Recall that built-in functions are only evaluated at the following syntactic positions in a Tscript:

- The right-hand side of an assignment.
- The test in an if-then or if-then-else construct.
- The expression in time-related constructs.

Boolean functions

Table 1.1. Boolean functions

Function	Result type	Description
`not(<bool>`$_1$`)`	`<bool>`	$\neg$ `<bool>`$_1$
`and(<bool>`$_1$`, <bool>`$_2$`)`	`<bool>`	`<bool>`$_1 \wedge$ `<bool>`$_2$
`or(<bool>`$_1$`, <bool>`$_2$`)`	`<bool>`	`<bool>`$_1 \vee$ `<bool>`$_2$
`equal(<term>`$_1$`, <term>`$_2$`)`	`<bool>`	`<term>`$_1 =$ `<term>`$_2$
`not-equal(<term>`$_1$`, <term>`$_2$`)`	`<bool>`	`<term>`$_1 \neq$ `<term>`$_2$

Integer functions

Table 1.2. Integer functions

Function	Result type	Description
`add(<int>`$_1$`, <int>`$_2$`)`	`<int>`	`<int>`$_1 +$ `<int>`$_2$
`sub(<int>`$_1$`, <int>`$_2$`)`	`<int>`	`<int>`$_1 -$ `<int>`$_2$
`mul(<int>`$_1$`, <int>`$_2$`)`	`<int>`	`<int>`$_1 \times$ `<int>`$_2$
`div(<int>`$_1$`, <int>`$_2$`)`	`<int>`	`<int>`$_1 /$ `<int>`$_2$
`mod(<int>`$_1$`, <int>`$_2$`)`	`<int>`	`<int>`$_1$ **mod** `<int>`$_2$
`abs(<int>`$_1$`)`	`<int>`	$\vert$`<int>`$_1\vert$
`less(<int>`$_1$`, <int>`$_2$`)`	`<bool>`	`<int>`$_1 <$ `<int>`$_2$
`less-equal(<int>`$_1$`, <int>`$_2$`)`	`<bool>`	`<int>`$_1 \leq$ `<int>`$_2$
`greater(<int>`$_1$`, <int>`$_2$`)`	`<bool>`	`<int>`$_1 >$ `<int>`$_2$
`greater-equal(<int>`$_1$`, <int>`$_2$`)`	`<bool>`	`<int>`$_1 \geq$ `<int>`$_2$

Real functions

Table 1.3. Real functions

Function	Result type	Description
`radd(<real>`$_1$`, <real>`$_2$`)`	`<real>`	`<real>`$_1 +$ `<real>`$_2$
`rsub(<real>`$_1$`, <real>`$_2$`)`	`<real>`	`<real>`$_1 -$ `<real>`$_2$
`rmul(<real>`$_1$`, <real>`$_2$`)`	`<real>`	`<real>`$_1 \times$ `<real>`$_2$
`rdiv(<real>`$_1$`, <real>`$_2$`)`	`<real>`	`<real>`$_1 /$ `<real>`$_2$
`mod(<real>`$_1$`, <real>`$_2$`)`	`<real>`	`<real>`$_1$ **mod** `<real>`$_2$
`rabs(<real>`$_1$`)`	`<real>`	$\vert$`<real>`$_1\vert$
`rless(<real>`$_1$`, <real>`$_2$`)`	`<bool>`	`<real>`$_1 <$ `<real>`$_2$
`rless-equal(<real>`$_1$`, <real>`$_2$`)`	`<bool>`	`<real>`$_1 \leq$ `<real>`$_2$
`rgreater(<real>`$_1$`, <real>`$_2$`)`	`<bool>`	`<real>`$_1 >$ `<real>`$_2$
`rgreater-equal(<real>`$_1$`, <real>`$_2$`)`	`<bool>`	`<real>`$_1 \geq$ `<real>`$_2$

Goniometric functions

Table 1.4. Goniometric functions

Function	Result type	Description
`sin(<real>`$_1$`)`	`<real>`	*sin*(`<real>`$_1$)
`cos(<real>`$_1$`)`	`<real>`	*cos*(`<real>`$_1$)
`atan(<real>`$_1$`)`	`<real>`	*tan*$^{-1}$(`<real>`$_1$) in the range [-π/2, π/2]
`atan2(<real>`$_1$`, <real>`$_2$`)`	`<real>`	*tan*$^{-1}$(`<real>`$_1$/`<real>`$_2$) in the range [-π, π]
`exp(<real>`$_1$`)`	`<real>`	$e^{\text{<real>}_1}$
`log(<real>`$_1$`)`	`<real>`	natural logarithm *ln*(`<real>`$_1$), with `<real>`$_1$ > 0
`log10(<real>`$_1$`)`	`<real>`	base 10 logarithm *log*$_{10}$(`<real>`$_1$), with `<real>`$_1$ > 0
`sqrt(<real>`$_1$`)`	`<real>`	√`<real>`$_1$, with `<real>`$_1$ ≥ 0

Functions on lists

Table 1.5. Functions on lists

Function	Result type	Description
`first(<list>`$_1$`)`	`<term>`	First element of `<list>`$_1$; The empty list `[]` when applied to non-list or empty list.
`next(<list>`$_1$`)`	`<list>`	Remaining elements of `<list>`$_1$.
`join(<term>`$_1$`, <term>`$_2$`)`	`<list>`	Concatenation of `<term>`$_1$ and `<term>`$_2$. When both arguments are lists their elements are spliced into a new list. A non-list argument is included as single element in the new list.
`size(<list>`$_1$`)`	`<int>`	The number of elements in `<list>`$_1$.

Table 1.6. Functions on lists as arrays

Function	Result type	Description
`index(<list>`$_1$`, <int>`$_1$`)`	`<term>`	The `<int>`$_1$-th element of `<list>`$_1$, if it exists; otherwise `[]`.
`replace(<list>`$_1$`,<int`$_1$`>,<term>`$_1$`)`	`<list>`	If the `<int>`$_1$-the element exists, replace it by `<term>`$_1$ and returned the modified list; otherwise return `<list>`$_1$ unmodified.

Table 1.7. Functions on lists as symbol tables

Function	Result type	Description
`get(<list>`$_1$`, <term>`$_1$`)`	`<term>`	If `<list>`$_1$contains a pair `[<term>`$_1$`, <term>`$_1$`']` then return `<term>`$_1$`'`; otherwise [].
`put(<list>`$_1$`,<term>`$_1$`,<term>`$_2$`)`	`<list>`	If `<list>`$_1$contains a pair `[<term>`$_1$`, <term>`$_1$`']` then replace it by `[<term>`$_1$`, <term>`$_2$`]`; otherwise add a new pair [<term>$_1$, <term>$_2$] to <list>$_1$.

Table 1.8. Functions on lists as multi-sets

Function	Result type	Description
`member(<term>`$_1$`, <list>`$_1$`)`	`<bool>`	`<term>`$_1$ $\in$ `<list>`$_2$ (membership in multi-set)
`subset(<list>`$_1$`,<list>`$_2$`)`	`<bool>`	`<list>`$_1$ $\subseteq$ `<list>`$_2$ (subset on multi-set)
`diff(<list>`$_1$`,<list>`$_2$`)`	`<list>`	`<list>`$_1$ $-$ `<list>`$_2$ (difference on multi-set)
`inter(<list>`$_1$`,<list>`$_2$`)`	`<list>`	`<list>`$_1$ $\cap$ `<list>`$_2$ (intersection on multi-set)

Functions on terms

Table 1.9. Functions on ATerms

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

Time-related functions

Table 1.10. Time-related functions

Function	Result type	Description
`current-time`	`<list>`	Six-tuple describing the current absolute time
`sec(<int>)`	`<int>`	Convert <int> into seconds

Miscellaneous functions

Table 1.11. Miscellaneous functions

Function	Result type	Description
`process-id`	`<int>`	Process id of the current process
`process-name`	`<str>`	Name of the current process
`quote(<term>)`	`<term>`	Quoted (unevaluated) term; only variables are replaced by their values
`functions`	`<list>`	List of all built-in functions

Synopsis of ToolBus primitives

In the following two sections all primitives are summarized that can occur in a Tscript.

Process-related primitives in Tscripts

Table 1.12. Process-related primitives in Tscripts

Primitive	**Synopsis**	**See**
delta	Inaction (deadlock)	
tau	Internal step	
$P_1 + P_2$	Choice between P_1 and P_2	
$P_1 . P_2$	P_1 followed by P_2	
$P_1 \parallel P_2$	P_1 parallel with P_2	
$P_1 * P_2$	Repeat P_1 until P_2	
`if` *T* `then` *P* `fi`	Guarded command	
`if` *T* `then` P_1 `else` P_2 `fi`	Conditional	
`create(`*Pnm*`(`*T*`, ...),` *Pid*`?)`	Create new process	
V `:=` *T*	Assign *T* (seen as expression) to *V*	
`snd-msg(`*T*`)`	Send synchronous message	
`rec-msg(`*T*`)`	Receive a synchronous message	
`snd-note(`*T*`)`	Broadcast an asynchronous note	
`rec-note(`*T*`)`	Receive an asynchronous note	
`no-note(`*T*`)`	No note available	
`subscribe(`*T*`)`	Subscribe to notes	
A `delay(`*T*`)`	Relative delay of atom execution	
A `abs-delay(`*T*`)`	Absolute delay of atom execution	
A `timeout(`*T*`)`	Relative timeout of atom execution	
A `abs-timeout(`*T*`)`	Absolute timeout of atom execution	
`shutdown(`*T*`)`	Terminate ToolBus application	
`printf(`*S*`,` *T*`, ...)`	Print terms according to format string S	
`read(`T_1`,` T_2`)`	Give prompt $T_1$and read term that should match with T_2	
`process` *Pnm*`(`*F*`, ...) is` *P*	Define process *Pnm*	
`let` *F*`, ... in` *P* `endlet`	Declare local variables in *P*	
`ToolBus(`*Pnm*`(`*T*`,...), ...)`	Define initial ToolBus process configuration	

Tool-related primitives in Tscripts

Table 1.13. Tool-related primitives in Tscripts

Primitive	Synopsis	See
`rec-connect(`*`Tid`*`?)`	Receive connection request from tool	
`rec-disconnect(`*`Tid`*`?)`	Receive disconnection request from tool	
`execute(`*`Tnm`*`(`*`T`*`,...),` *`Tid`*`?)`	Execute a tool	
`snd-terminate(`*`Tid`*`,` *`T`*`)`	Terminate execution of a tool	
`snd-eval(`*`Tid`*`,` *`T`*`)`	Send evaluation request to tool	
`snd-cancel(`*`Tid`*`)`	Cancel previous evaluation request	
`rec-value(`*`Tid`*`,` *`T`*`)`	Receive answer to evaluation request	
`snd-do(`*`Tid`*`,` *`T`*`)`	Send evaluation request to tool (no return value)	
`rec-event(`*`Tid`*`,` *`T`*`, ...)`	Receive event from tool	
`snd-ack-event(`*`Tid`*`,` *`T`*`)`	Acknowledge previous event from tool	
`tool` *`Tnm`* `is {` *`Feat`*`, ... }`	Define tool *`Tnm`*	
`host =` *`Str`*	Host feature in tool definition	
`command =` *`Str`*	Command feature in tool definition	

Historical notes

The first generation ToolBus is described in [BK94] [59]. In addition to the design, the complete C implementation is discussed in detail. The second generation ("discrete time") ToolBus includes timing primitives as well as built-in functions. It has been formally described using ASF+SDF, see [BK95] [59] and [BK98] [59] In [Oli00] [60] a framework for the debugging of ToolBus applications is presented. Initial thoughts about a next generation ToolBus were published in [dJK03] [59]. [dJ07] [59] describes architectural aspects of ToolBus-based applications.

Warning

Add: theses of Peter Heibrink, Arnold Lankamp, Dennis Hendriks.

Bibliography

[BK94] J.A. Bergstra and P. Klint. *The toolbus: a component interconnection architecture*. Technical ReportP9408. University of Amsterdam, Programming Research Group. 1994.

[BK95] J.A. Bergstra and P. Klint. *The discrete time toolbus*. Technical ReportP9502. University of Amsterdam, Programming Research Group. 1995.

[BK98] J.A. Bergstra and P. Klint. *The discrete time ToolBus -- a software coordination architecture*. 205--229. *Science of Computer Programming*. 31. 2-3. July 1998.

[dJK03] H.A. de Jong and P. Klint. *Toolbus: the next generation*. 220--241. Formal Methods for Components and Objects. . F.S. de Boer, M. Bonsangue, S. Graf, and W.P de Roever. Lecture Notes in Computer Science.
<seriesvolnum>2852</seriesvolnum>
2003. Springer.

[dJ07] H.A. de Jong. *Flexible Heterogeneous Software Systems*. PhD thesis. University of Amsterdam. 2007.

[Oli00] P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis. University of Amsterdam. 2000.

To Do

- What do we do with the other adapters?
- Describe current viewer.
- Describe console commands.
- Do we describe the global structure of the Java implementation (or partially refer to online docs)?