A Guide to ToolBus Programming

P. Klint[1,2]

April 22, 2002

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

Abstract

The ToolBus is a new software architecture intended for building cooperating, distributed applications. This guide aims at providing a comprehensive but complete explanation of "ToolBus programming": writing ToolBus scripts (or **T** scripts for short) describing the overall architecture of an application and writing tools that actually implement the application's functionality.

Contents

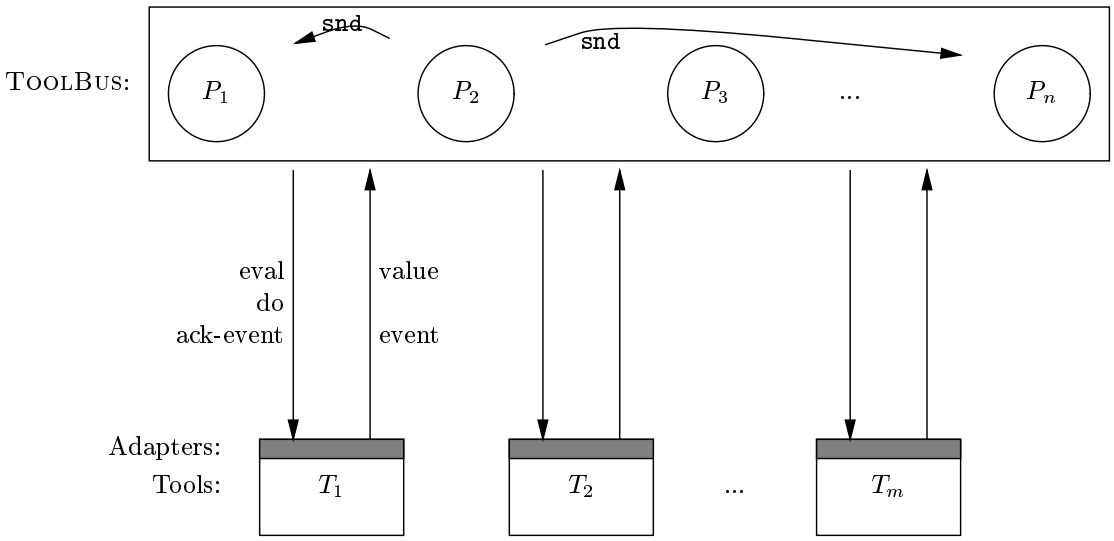

Figure 1: Global organization of the ToolBus

1 Introduction

1.1 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.

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".

1.2 The ToolBus architecture

The global architecture of the ToolBus is shown in Figure 1. The ToolBus serves the purpose of defining the cooperation of a 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.

The ToolBus itself consists of a variable number of processes[1] P_i $(i = 1, ..., n)$. 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 processes. They come into existence either by an execution command issued by the ToolBus or their execution is initiated

[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 processes".

```
/* hello1.tb -- Our first ToolBus script */

process HELLO is printf("Hello world, my first T script!\n")

toolbus(HELLO)
```

Figure 2: `hello1`: first script for the `hello` application.

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 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`) at a low priority. Notes are intended to notify others of state changes in the sending process. Sending notes amounts to *asynchronous selective broadcasting*. Processes will only receive notes to which they have *subscribed*.

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.

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.

1.3 Purpose of this guide

This guide is a companion to the various ToolBus papers[2] fully describing the motivation and overall architecture of the ToolBus and explaining the **T** scripts used to describe cooperating sets of tools. The reader is also referred to these reports for several examples of systems that have been described using the ToolBus approach.

Here, the main emphasis is on explaining *all* details needed to actually implement systems using the ToolBus. First, we will give a "hello world" example in the context of the ToolBus.

[2]The most comprehensive publication is J.A. Bergstra and P. Klint, "The discrete time ToolBus – a software coordination architecture", *Science of Computer Programming*, 31(2-3):205–229, July 1998.

Technical reports giving detailed descriptions of the semantics (using ASF+SDF specifications) and implementation of the ToolBus are: J.A. Bergstra and P. Klint, "The ToolBus—a component interconnection architecture", Report P9408, Programming Research Group, University of Amsterdam, 1994, and J.A. Bergstra and P. Klint, "The Discrete Time ToolBus", Report P9502, Programming Research Group, University of Amsterdam, 1995.

```
/* hello2.tb -- hello script using a separate hello tool */

process HELLO is
  let H : hello,               %% H will represent the hello tool
      S : str                  %% S is a string valued variable
  in
      execute(hello, H?) .     %% Execute hello, H gets a tool id as value
      snd-eval(H, get-text) .  %% Request a text from the hello tool
      rec-value(H, text(S?)).  %% Receive it, S gets the text as value
      printf(S)                %% Print it
  endlet

tool hello is {command = "hello"}

toolbus(HELLO)
```

Figure 3: `hello2`: second script for the `hello` application.

1.4 Hello world

The most simple program that is frequently used to learn a new programming language is a program which prints some string (e.g., "hello world") as proof of competence of its author to write, compile, and execute a program in the language in question. Clearly, it is the road to arrive at this result that counts and not the result itself (as the old proverb says).

The simplest hello program possible is shown in Figure 2. Typing the command

```
toolbus hello1.tb
```

will simply print the desired message.

Let's now be more ambitious. In the above example, the text to be printed appears as a literal string in the script. We complicate the example by introducing a "`hello`" tool that will provide the text to be printed. This results in the script given in Figure 3. But how do we implement the `hello` tool itself? We will explain in this guide the range of implementation languages that can be used (i.e., C, Tcl, Perl, ASF+SDF ...). For the sake of this example we only show what a C implementation will look like.

In Figure 4 a first, simple, version of the `hello` tool is shown. It consists of the following parts[3]:

- An include of a standard header file (`atb-tool.h`) that contains common definitions for all tools.
- A declaration of a function `hello_handler` that is called when there is input available from the ToolBus: its argument `inp` is the input term, and its result (either a term or `NULL`) will be sent back to the ToolBus.[4] The input is analyzed by using `ATmatch`, a library function for matching terms. For the `get-text` case, the term

  ```
  snd-value("Hello World, my first ToolBus tool in C!\n")
  ```

  is constructed and returned to the ToolBus.
- A main program that calls an initialization function and then enters an event loop.

Although it is not yet clear from the examples given so far, it turns out that there is much commonality among the handlers written for different tools. In particular, the code for analyzing terms coming from

[3] You are not yet supposed to understand every detail of these listings, but you *will* be able to do so after reading this guide!

[4] It is important to stress that the handler should **always** return a value: either a term or NULL.

the ToolBus is similar. This code also duplicates information in the script concerning the requests sent to each tool. For this reason, we also provide a *tool interface generator* that automatically generates tool interfaces from a given script.

This approach is shown in a second version of the `hello` tool (Figure 5). From the script `hello2.tb` we generate automatically[5] the following two files:

- `hello.tif.h`: this includes necessary header files, and declares prototypes for application functions `get_text`, `rec_terminate` and the ToolBus interfacing functions `hello_handler` (handles all requests coming from the ToolBus) and `hello_checker` (checks that the interface as expected by the ToolBus is compatible with the interface as provided by the tool).
- `hello.tif.c`: includes `hello.tif.h` and contains the declarations for `hello_handler` and `hello_checker`.

The actual program `hello.c` consists of the following parts:

- An include of `hello.tif.c`
- A declaration of the function `get_text` that handles the eval request coming from the ToolBus Note that `get_text` is called by `hello_handler` and that its return value will be sent back to the ToolBus.
- A declaration of the function `rec_terminate` that is always called on termination of a tool.
- A main program that calls an initialization function and then enters an event loop.

What we see in these examples is that building an application with the ToolBus requires the following steps:

- Design the overall behaviour of the application by writing a **T** script (`hello2.tb`).
- Write and compile the tools needed by the script (`hello.c`). The required interfacing code can be written by hand or be generated automatically from the **T** script (`hello.tif.c`).
- Execute the ToolBus interpreter with the script as input.

1.5 Further reading

If you have come this far, you may be interested to learn more about the details of ToolBus programming. In Section 2 the ways to execute the ToolBus interpreter and tools are described. Next follows an intermezzo explaining the overall structure of ToolBus adapters (Section 3). In Section 4 you will find a complete description of the library functions provided for writing tools in C.

In the sections that follow we explain how to write tools in various languages and systems: arbitrary Unix commands (Section 5), Java (Section **??**), Perl (Section 6), Python (Section 7), and Tcl/Tk (Section 8).

Four appendices with summaries conclude this guide.

2 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.

[5] In section 4.5 this is fully explained.

2.1 Common arguments

ToolBus and tools have the following optional arguments in common:

- `-help`: prints a description of all arguments of the toolbus or tool.
- `-verbose`: produces a log of steps taken by toolbus or tool that may be useful to debug your script or tool. The same effect may be obtained by setting the environment variable `TB_VERBOSE` to `true` and export it. In the Korn shell this can, for instance, be achieved by:

```
TB_VERBOSE=true
export TB_VERBOSE
```

- `-TB_PORT` *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.

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

2.2 ToolBus arguments

The *script_name* (see below) given as argument to the ToolBus is always preprocessed by the C preprocessor before it is parsed as a **T** script. In this way, directives like, e.g., `#define`, `#include` and `#ifdef` can be used freely in **T** scripts. 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`*macro*: define macro *macro* with the string "`1`" as its definition.
- `-D`*macro*`=`*defn*: define macro *macro* with *defn* as definition.

Other arguments specific for the `toolbus` command are:

- `-logger`: execute a logger tool that will be attached to all processes in the ToolBus. If the script contains a tool definition for a tool named "`logger`", that will be used for executing the logger. Otherwise a default tool definition is used.
- `-viewer`: similar as above, for a viewer tool. The default viewer is the "ToolBus viewer" (previously known as the ToolBus debugger).
- `-controller`: similar as above, for a controller tool. Currently, no default controller tool is provided.
- `-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.
- `-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.
- *script_name*: any other argument is the name of the ToolBus script to be interpreted.

As an example, consider first

```
toolbus hello.tb
```

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

```
toolbus -TB_PORT 4000 hello.tb
```

which interprets the same script, but uses socket `4000` to find the ToolBus. Next, consider,

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

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

```
toolbus -gentifs hello.tb
```

produces the tool interfaces file `hello.tifs`.

2.3 Tool arguments

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 **T** script (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).
- `-TB_SINGLE`: execute the tool stand-alone and do not connect it with the ToolBus.

The execution of a tool can start in two ways:

- The tool is started by an `execute` command in the **T** script.
- 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 Section 1.4. The `hello` tool will be executed by the ToolBus using the command

```
hello -TB_PORT 8998 -TB_HOST host1.institute.nl
```

when running on machine `host1.institute.nl`.

Suppose, we replace the explicit `execute` in Figure 3 by a `rec-connect` as shown in Figure 6. 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 -TB_HOST host1.institute.nl
```

3 Adapters for tools and languages

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 7. 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.
- 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.

Our experience so far is restricted to adapters of the first category. 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.

Common arguments of adapters. 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 Section 2.3.)

4 Writing tools in C

Although ToolBus tools can be implemented in many languages (including Java, C++, Perl, Tcl/Tk, Prolog, ASF+SDF, Cobol and others) we start explaining how tools can be written in C. In other languages identical notions will be used with only minor adjustements 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 ToolBus (Section 4.1).
- The global structure of a ToolBus tool (Section 4.2).
- The ToolBus Application Programmer's Interface (Section 4.3).
- Compiling ToolBus tools written in C (Section 4.4).
- Generating tool interfaces with `tifstoc` (Section 4.5).

4.1 ATerms: Composing and decomposing terms

We use a datatype called ATerms for creating, matching, reading and writing terms. ATerms are fully described elsewhere.[6] Here we only give a brief overview.

[6] H. A. de Jong and P. A. Olivier, "ATerm Library User Manual" and M.G.J. van den Brand, H.A. de Jong, P. Klint and P.A. Olivier, "Efficient Annotated Terms", *Software Practice and Experience*, 30:259–291, 2000.

4.1.1 Term patterns

Composition and decomposition of terms is *not* based on the direct manipulation of the underlying representation of terms. Instead, *term patterns* are used to guide composition and decomposition. Such term patterns play the same role as format strings in the printf/scanf paradigm in C. In first approximation, a term pattern is a literal string that would be obtained by a preorder traversal of a term. For instance, the term pattern `"or(true, false)"`, corresponds to a term whose root is labeled with the symbol `or`, and whose children are labeled with, respectively, `true` and `false`. In this way, term patterns can be used to construct and to match terms.

Term patterns become, however, much more useful if they can be parameterized with subterms that have been computed separately. To this end, we introduce the notion of *directives* as follows:

`<int>` : corresponds to an integer (in C: `int`);

`<str>` : corresponds to a string (in C: `char *`);

`<blob>` : corresponds to a binary string (in C: a (length, pointer) pair represented by two values of types, respectively, `int` and `void *`);

`<term>` : corresponds to an aterm (in C: `ATerm`);

`<appl>` : corresponds to one function application(in C: `char *pattern`, followed by arguments);

`<list>` : corresponds to a list of terms (in C: `ATerm`).

The precise interpretation of these directives depends on the context in which they are used. When constructing a term, directives indicate that a subterm should be obtained from some given variable. When matching a term, directives indicate the assignment of subterms to given variables. For the implications of these directives for memory management, see Section 4.1.8.

4.1.2 `ATmake`

The function

```
term *ATmake(char *Pattern, ...)
```

constructs a term according to `Pattern`, where occurrences of directives are replaced by the values of the variables occurring in `...`.

For instance, assuming the declarations

```
int n = 10;
char *fun = "pair", name = "any";
ATerm yellow = ATmake("yellow"), t;
```

the call

```
t = ATmake("exam(<appl(<term>,9)>,<int>,<str>)", fun, yellow, n, 10, name)
```

will construct the term `t` with value

```
exam(pair(yellow,9),10,10,"any")
```

Binary strings (*Binary Large OBjects* or *blob*s) are used to represent arbitrary length, binary data that cannot be represented by ordinary C strings because they may contain "null" characters. A binary string is represented by a character pointer and a length. For instance, given

```
char buf[12];
ATerm bstr;
buf[0] = 0; buf[1] = 1; buf[2] = 2;
```

the call

```
bstr = ATmake("exam(<blob>)", 3, buf);
```

will construct a term with function symbol "exam" and as single argument a binary string of length 3 consisting of the three values 0, 1, and 2.

4.1.3 ATmatch

Matching terms amounts to

- determining whether there is a match or not,
- selectively assigning matched subterms to given variables.

This is precisely what the function

```
ATbool ATmatch(ATerm Trm, const char *Pattern, ...)
```

does. It matches Trm against Pattern and, when a submatch is found that corresponds to a directive, it makes assignments to variables whose addresses appear in .... For most directives, the values assigned to these variables are pointers to subterms of Trm. Pattern should be a well-formed, textual representation of a term which may contain any of the directives described earlier. For instance, in the context

```
ATerm t = ATmake("exam(pair(yellow,9),10, \"any\")");
ATerm t1;
int n;
char *ex, *s;
```

the call

```
ATmatch(t, "appl(<term>,<int>,<str>)", &ex, &t1, &n, &s);
```

yields true and is equivalent to the following assignments:

```
ex = "exam";
t1 = ATmake("pair(yellow,9)");
n = 10;
s = "any";
```

As explained in full detail in Section 4.1.8, memory is managed automatically by the ATerm library. As a general rule, the values for ex, t1, and s are pointers into the original term t rather than newly created values. As a result, they have a life time that is equal to that of t.

Matching binary strings is the inverse of constructing them. Given the term bstr constructed at the end of the previous paragraph, its size and contents can be extracted as follows:

```
int n;
char *p;

ATmatch(bstr, "exam(<blob>)", &n, &p);
```

ATmatch will succeed and will assign 3 to the variable n and will assign a pointer to the character data in the binary string to the variable p.

Here, again, the value of p is a pointer into the term bstr rather than a newly allocated string.

Notes

- Double quotes (“"”) appearing *inside* the pattern argument of both `ATmake` and `ATmatch` have to be escaped using “\"”.
- The number and type of the variables whose addresses appear as arguments of `ATmatch` should correspond, otherwise disaster will strike (as usual when using C).
- Assignments are being made during matching. As a result, some assignments may be performed, even if the match as a whole fails.

4.1.4 Input and output of ATerms

We make a distinction between the “raw” input and output of terms as they are, for instance, being sent through communication channels between ToolBus and tools, versus formatted input and output of terms. Raw term i/o is provided by `TBwrite` and `TBread`. Formatted term output is provided by `TBprintf` and `TBsprintf`. There are currently no primitives for formatted term input.

4.1.5 Reading and writing ATerms

ATerms can be read from and written to strings and files. Two formats are supported: a human-readable but verbose textual format and a very concise binarry format. Here we only discuss the textual variant.

The function

```
void ATwriteToTextFile(ATerm Trm, FILE *File)
```

writes ATerm `Trm` to the file `File`. For instance, in the context:

```
FILE *f = fopen("foo", "wb");
ATerm Trm1 = ATmake("<appl(red,<int>)>", "freq", 17);
```

the statement

```
ATwriteToTextFile(Trm1, f);
```

will write the value of `Trm1` (i.e., `freq(red,17)`) to file “`foo`”.

The function

```
ATerm ATreadFromFile(FILE *File)
```

is the inverse of `ATwriteToFile`: it reads a term (either in textual format or in internal format) from a file and returns it as value. When end of file is encountered or the term could not be read, the operation is aborted.[7]

4.1.6 `ATfprintf`

The function

```
int ATfprintf(FILE *File, const char *Pattern, ...)
```

writes formatted output to `File`. `Pattern` is printed literally except for occurrences of directives which are replaced by the textual representation of the values appearing in `...`. For instance,

```
ATfprintf(stderr, "Wrong event \"%t\" ignored\n", ATmake("failure(<int>)", 13));
```

will print:

```
Wrong event "failure(13)" ignored
```

Note that `ATprintf` uses the normal `printf` conversion specifiers extended with aterm-specific specifiers. The most frequently used specifier is `%t` which stands for an aterm argument whose textual representation is to be inserted in the output stream.

[7] The user can redefine this behaviour using `ATsetAbortHandler`, which allows the definition of a user-defined abort handler. See the *ATerm Library User Manual* for further details.

4.1.7 Further ATerm manipulation functions

The ATerm library acutally provides two interfaces:

- The level 1 interface: a simple but expressive interface as just sketched.
- The level 2 interface: a more detailed interface that allows the very efficient coding of operations on ATerms. We will not further discuss here the level 2 interface.

4.1.8 Memory Management of ATerms

The functions in the ATerm library provide automatic memory management of terms. Terms that have been created but are no longer referenced are removed by a method called *garbage collection*. The global model is that there is a set of *protected* terms that are guaranteed to survive a garbage collection. Effectively, all protected terms (and their subterms) are conserved and all other terms are considered as garbage and can be collected.

It is guaranteed that no garbage collection takes place during the execution of an event handler, hence it is not necessary to protect temporary terms that are constructed during the execution of an event handler. However, terms that should have a longer life time must be protected in order to survive.

In order to protect terms from being collected, the function

```
void ATprotect(ATerm *TrmPtr)
```

can be used that has as single argument *a pointer to a variable with an ATerm as value*. The protection can be undone by the function

```
void ATunprotect(ATerm *TrmPtr)
```

The interplay between garbage collection and program variables is subtle. The following points are therefore worth mentioning:

- Functions that return a term as value (e.g., `TBreadTermfromFile`) do not explicitly protect it but the result may, of course, be protected because it is a subterm of an already protected term.
- The function `ATmake` uses strings and terms and includes them into a new term T. The implications for memory management are:
  - All string arguments (using `<str>`, `<blob>` or `<appl>`) are copied before they are included into T. They can thus safely be deallocated (e.g., using `free`) by the C program.
  - All term arguments (using `<term>`) are included into T by means of a pointer. They thus become reachable from T and their life time becomes at least as large as that of T; it is unnecessary to explicitly protect them.
- The function `ATmatch` assigns strings and terms to program variables by extracting them from an existing term T. The general rule here is that extracted values have a life time that is equal to that of T. The implications for memory management are:
  - All string values (obtained using `<str>`, `<blob>` or `<appl>`) should be copied if they should survive T.
  - All term values (obtained using `<term>`) should be explicitly protected if they should survive T.

4.1.9 Initializing and using the ATerm library

Using the ATerm library requires the following:

- Include the header file `aterm1.h` (or `aterm2.h` if you want to use the level 2 interface). `aterm1.h` defines:
  - `ATbool`: the boolean data type defined by

    ```
    typedef enum ATbool {ATfalse=0, ATtrue} ATbool;
    ```

    It is mainly used as the return value of library functions.
  - `ATerm`: the type definition of ATerms. The ATerm library has been designed in such a way that only pointers to terms must be passed to or are returned by library functions. The primitives that are provided for constructing and decomposing terms are of such a high level that it is unnecessary to know the internal representation of terms. When necessary, you can access the internal structure of ATerms using the level 2 interface.
- Declare in your `main` program a local ATerm variable that will be used to determine the bottom of C's runtime stack.
- Call `ATinit` to initialize the ATerm library.
- Link the ATerm library `libATerm.a` when compiling your application. This is achieved using the `-lATerm` option of the C compiler.

A typical usage pattern is as follows:

```
#include <aterm1.h>
int main(int argc, char *argv[])
{
  ATerm bottomOfStack;
  ATinit(argc, argv, &bottomOfStack);
 /* ... code that uses ATerms ... */
}
```

4.2 The global structure of a ToolBus tool

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 8. 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).

At the level of the source code, the global structure of a purely reactive tool without additional term or character ports has already been illustrated in Figure 4.

4.2.1 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

4.2.2 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[8]:

- `ATBinit`: tool initialization (Sections 4.3.1).
- `ATBconnect`: to connect the tool to the ToolBus (Sections 4.3.2).
- `ATBdisconnect`: to disconnect the tool from the ToolBus (Sections 4.3.2).
- `ATBeventloop`: a standard event loop for a tool (Section 4.3.3).
- `ATBreadTerm`: process one input event on a port (Section 4.3.4).
- `ATBwriteTerm`: send a term to the ToolBus (Section 4.3.4).

In the following section, we will describe these functions.

[8] For an exhaustive description, see H. A. de Jong and P. A. Olivier, "ATerm Library User Manual"

4.3 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 **T** script.
- 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.

4.3.1 `ATBinit`

The initialization of a tool is achieved by

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

The standard program arguments that are passed (via `argc` and `argv`) are fully described in Section 2. 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 **T** script. 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 Section 4.1.9).

4.3.2 `ATBconnect` and `ATBdisconnect`

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 descriptior 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`. 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 retruned 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 `int` is a previously created connection.

4.3.3 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, giving up!\n");
  ATBeventloop();
  return 0;
}
```

4.3.4 ATBwriteTerm and ATBreadTerm

So far, we have seen primitives for tools that only receive terms from the ToolBus. In the case of, for instance, 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 ATBwriteTerm(int conn, ATerm term)
```

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

```
ATBwriteTerm(conn, ATmake(snd-event(button("ok")).
```

Symmetrically, a term can be read from a ToolBus connection as follows:

```
ATerm ATBreadTerm(int conn).
```

4.3.5 Advanved 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. This is achieved by the function

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

which returns `ATtrue` if incoming data from the 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

```
void ATBhandleAny(void)
```

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 **T** script being used should, of course, be able to receive such events.

4.4 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`?
- 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.

4.5 Generating C tool interfaces with `tifstoc`

The interface code for each tool depends on the particulars of the **T** script 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.

As already mentioned in Section 1.4, 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 **T** script. This generation process is shown in Figure 9 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`[9] 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
  ```

In Figure 9 it is also shown how tool interface generators for *other* languages (e.g., Java, Cobol) fit into this scheme. In addition to `tifstoc`, we alo support the generation of Java interfaces by way of `javatif` (See Section **??**).

5 Using arbitrary Unix commands as tool

Using arbitrary Unix commands as ToolBus tool is achieved by the `gen-adapter` to be explained in Section 5.1. An example of its is use is given in Section 5.2.

5.1 gen-adapter

Synopsis. Execute an arbitrary Unix command as tool.

Example. `gen-adapter -cmd ls -l`

Specific arguments.

- `-addnewline`: always add a newline character to the standard input for the command.
- `-keepnewline`: keep the last newline character in the output generated by the command. Without this argument, the last newline character is always removed.
- `-string-output`: return the output of the command as string (type: `<str>`). When no output format options has been specified, `-string-output` is used.
- `-binary-output`: return the output of the command as a binary string (type: `<bstr>`).
- `-term-output`: return the output of the command as term (type: `<term>`).

Communication.[10]

- `snd-eval(`*Tid*`, cmd(`*Cmd*`, input(`*Str*`))`: execute the Unix command

  Cmd `<` *Str*

  i.e., execute *Cmd* with *Str* as standard input. The output of this command execution is captured and will be returned by default as string value (see below). Other output formats can be selected using command line options. *Tid* is a tool identifier (as produced by `execute` or `rec-connect`) for an instance of the `gen-adapter`.

[9] This is an ATerm-compatible version of a similar generator called `ctif`. `ctif` is still in use for some older tools but will be gradually phased out.

[10] Communication is described from the point of view of the ToolBus, i.e., `snd-` and `rec-` mean, respectively, send by ToolBus and receive by ToolBus.

- `snd-eval(`*Tid*`, cmd(`*Cmd*`, input(`*Bstr*`))`: same as above, except that a binary string is used as standard input for *Cmd*.
- `rec-value(`*Tid*`,output(`*Res*`))`: the return value for a previous evaluation request. *Res* is a string containing the output produced by the command execution. By default `gen-adapter` returns the output of the executed command as ordinary string. Other output formats can be specified using command line options. **Note:** in the current implementation of `gen-adapter` there is an arbitrary limit (10000) on the *size* of the output produced by the command.
- `snd-terminate(`*Tid*`, `A_1`)`: terminate execution of `gen-adapter`.

5.2 Example: pipe communication between two Unix commands

Suppose we want to count how many words there are in a listing of the current file directory. At the Unix level, this can be achieved by

```
ls -l | wc
```

where "`ls -l`" produces the directory listing and "`wc -w`" counts the number of words in this listing. The same effect is achieved by the script given in Figure 10.

6 Writing tools in Perl

Writing ToolBus tools in Perl is greatly simplified by the `perl-adapter` to be explained in Section 6.1. Next, a small set of predefined Perl functions is described that are always loaded by the `perl-adapter` and can be used in any Perl script (Section 6.2). Finally, we present in Section 6.3 the Perl version of the `hello` tool.

6.1 `perl-adapter`

Synopsis. Execute a Perl script as tool.

Example. `perl-adapter -script hello.perl`

Specific arguments.

- `-script`: The Perl script to be executed.

Communication.[11]

- `snd-eval(`*Tid*`, `*Fun*`(`A_1`, ..., `A_n`)`: perform the Perl subroutine call `do` *Fun*`(`A_1`, ..., `A_n`)`. Here *Tid* is tool identifier (as produced by `execute` or `rec-connect`) for an instance of the `perl-adapter`.
- `rec-value(`*Tid*`,`*Res*`)`: the return value for a previous evaluation request.
- `rec-event(`*Tid*`, `A_1`, ..., `A_n`)`: event generated by Perl.
- `snd-ack-event(`*Tid*`, `A_1`)`: acknowledgement of a previously generated event.
- `snd-terminate(`*Tid*`, `A_1`)`: terminate execution of perl-adapter.

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

[11] Communication is described from the point of view of the ToolBus, i.e., `snd-` and `rec-` mean, respectively, send by ToolBus and receive by ToolBus.

6.2 Predefined Perl functions

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

- `TBstring` *Str*: converts a Perl string to a ToolBus string by surrounding it with double quotes and escaping double quotes occurring inside *Str*.
- `PERLstring` *Str*: converts a ToolBus string into a Perl string by removing surrounding double quotes.
- `TBerror` *Msg*: constructs an error message that can be send back to the ToolBus.
- `TBsend` *Trm*: send *Trm* back to the ToolBus.

6.3 The hello example in Perl: `hello.perl`

Writing the hello tool in Perl requires two steps:

- Write the required Perl code `hello.perl`. The result is shown in Figure 11.
- Replace `hello`'s tool definition in `hello2.tb` by:

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

7 Writing tools in Python

You can write ToolBus tools in Python using a ToolBus-aware Python interpreter. How to build such a ToolBus-aware Python interpreter is explained in Section 7.1. Section 7.2 explains how to connect your python scripts that are executed by a ToolBus-aware Python interpreter to the ToolBus.

7.1 Building a ToolBus-aware Python interpreter.

Before adding ToolBus support to Python, you first have to retrieve and install Python, version 1.3. If you need more information about Python or more specific information about installing Python, you can visit the *Python home page* at `http://www.python.org/`. In this document we assume you have succesfully installed python and are only interested in adding ToolBus support.

The first thing to do, is to copy the file `adapters/python-adapter/TBmodule.c`, located in this ToolBus distribution to the Python `Modules` directory:

```
cp ToolBus/python-adapter/TBmodule.c Python-1.3/Modules
```

Now you have to add the two lines to the file `Setup` in the Python `Modules` directory. If you have enabled the `tkinter` module in the `Setup` file, these lines are:

```
TBBASE=<your ToolBus location>
TB TBmodule.c -I$(TBBASE)/include -I/home/olivierp/include $(TBBASE)/lib/libtb.a
```

If you do not have the `tkinter` module enabled in the `Setup` file, you have to add the following two lines instead:

```
TBBASE=<your ToolBus location>
TB TBmodule.c -DNO_TK -I$(TBBASE)/include TBBASE)/lib/libtb.a
```

Now type `make` and keep your fingers crossed. If all goes well, a new ToolBus-aware python interpreter will be build, which you can install using the command `make install`.

The script `python-adapter`, located in `adapters/python-adapter`, is used by your ToolBus scripts to start the ToolBus-aware python interpreter with the right arguments. This script is automatically moved to the ToolBus `bin` directory during the ToolBus installation.

7.2 Using the ToolBus-aware python interpreter

Synopsis. Start a python script as a tool.

Example. `python-adapter -script hello.py`

Specific arguments.

- `-program` *Name*: Use *Name* rather than `python` as the (ToolBus-aware) python interpreter.
- `-trace-calls`: Trace the calls made by the python-adapter. a list of function calls is printed to `stderr`.
- `-script` *Script*: Execute the python script *Script*.
- `-arg` *Arg*: Pass the argument *Arg* as a command line option to the python interpreter. This option can be repeated multiple times.
- `-script-args` *Arg1, Arg2, ...*: This must be the last option. The arguments *Arg1, Arg2, ...* are passed to the python script in the variable *TB.argv*.

Communication.[12]

- `snd-do`(*Tid*, *Fun*(*A*$_1$, ..., *A*$_n$)): perform the Python function call *Fun*(*cid*,*A*$_1$, ..., *A*$_n$). Here *Tid* is a tool identifier (as produced by `execute` or `rec-connect`) for an instance of the `python-adapter`, and *cid* is the connection id for this tool instance as returned by `TB.parseArgs` or `TB.newConnection`.
- `snd-eval`(*Tid*, *Fun*(*A*$_1$, ..., *A*$_n$)): perform the Python function call *Fun*(*Tid*, *A*$_1$, ..., *A*$_n$. *Tid* and *cid* as above. Note that the function *Fun* must send an anser back to the ToolBus using `return TB.make("snd-value(...)", ...)`.
- `rec-value`(*Tid*,*Res*): the return value for a previous evaluation request.
- `rec-event`(*Tid*, *A*$_1$, ..., *A*$_n$): event generated by python.
- `snd-ack-event`(*Tid*, *A*$_1$): acknowledgement of a previously generated event. Perform the Python function call `rec_ack_event`(*cid*, *A*$_1$).
- `snd-terminate`(*Tid*, *A*$_1$): terminate execution of the python tool. Perform the Python function call `rec_terminate`(*cid*, *A*$_1$).

7.3 Predefined Python functions

The following Python functions are predefined and can be used freely in Python scripts executed by a ToolBus-aware Python interpreter.

- `TB.parseArgs`(*Args*, *Module*): Parse the commandline options in *Args*, and create a new tool instance. *Module* is the Python module associated with this tool, for instance `"__main__"`. The tool is not yet connected to the ToolBus, until `TB.connect` is called. This function returns the new tool instance id, or raises an exception.
- `TB.newConnection`(*Tool*, *Host*, *Port*, *Module*): An alternate way to create a new tool instance. *Tool*, *Host*, and *Module* are strings, *Port* is an integer. *Host* can also be `None`, in which case the local host is always used. This function returns the new tool instance id, or raises an exception.

[12] Communication is described from the point of view of the ToolBus, i.e., `snd-` and `rec-` mean, respectively, send by ToolBus and receive by ToolBus.

- `TB.connect(`*Cid*`)`: Create the actual connection with the ToolBus, or raise an exception.
- `TB.eventloop()`: Start the ToolBus eventloop. Please do not use this function when you use the `Tkinter` module. If you use `Tkinter`, use `TB.enableTk` in combination with the `Tkinter` eventloop.
- `TB.enableTk()`: Instruct the `TB` module to register its callback functions with `Tkinter`, so the `Tkinter` eventloop can be used instead of the ToolBus eventloop.
- `TB.send(`*cid*`,` *Term*`)`: Send a term to the ToolBus. *Term* must be a Python term object.
- `TB.make(`*Fmt*`, ...)`: Build a new *term* object. You can use the following format directives in *Fmt*:
  - `<bool>`: Build a boolean term. This directive consumes one Python object from the argument list. If this object happens to be `None`, the created term is `false`, else the created term is `true`.
  - `<int>`: Build an integer term. This directive consumes one Python integer object from the argument list.
  - `<str>`: Build a string term. This directive consumes one Python string object from the argument list.
  - `<bstr>`: Build a binary string term. This directive consumes one Python string object from the argument list.
  - `<real>`: Build a real term. This directive consumes one Python float object from the argument list.
  - `<appl>`: Build a term application. This directive consumes two arguments: a string defining the function symbol, and a Python list of terms giving the arguments.
  - `<list>`: Build a list of terms. This directive consumes one argument: a Python list of terms.
  - `<term>`: Build a term. This directive consumes one argument: a Python term object.

7.4 Methods of the class `term`

The module `TB` introduces a new Python class called `term`. This class supports the following methods:

- *Term*`.kind()`: Returns the type of the term. This type is represented by one of the following strings:
  - `bool`
  - `int`
  - `real`
  - `str`
  - `bstr`
  - `bstr`
  - `appl`
  - `list`
- Value retrieval functions. Strictly speaking, these are not functions, but rather 'context sensitive attributes'. These attributes are only valid when the term is of the appropriate type.
  - *Term*`.bool`: *Term* must be a boolean. Returns 1 if the term is `true`, `None` otherwise.
  - *Term*`.int`: *Term* must be a integer. Returns a Python integer object representing the same value.

  - *Term*.`real`: *Term* must be a real. Returns a Python float object representing the same value.
  - *Term*.`str`: *Term* must be a string. Returns a Python string object representing the same string.
  - *Term*.`bstr`: *Term* must be a binary string. Returns a Python string object representing the same string.
  - *Term*.`appl`: *Term* must be a application. Returns a tuple containing the function symbol as a string and the arguments as a Python list of terms.
  - *Term*.`list`: *Term* must be a list. Returns a Python list of terms.

- *Term*.`simplify()`: Maps the term onto a native Python object. For instance, a term of type `<str>` is translated into a Python string object.

- *T*.`match(`*Fmt*`)`: Check if a term matches with a certain string. The string *Fmt* is first parsed into a term and then the two terms are matched. The function returns `None` when the two terms do not match. When the two terms do match, a list is returned that contains the subterms of *T* matching with the placeholders of *Fmt*.

- *T*.`matchTerm(`*Fmt*`)`: As *T*.`match`, but *Fmt* is now a term containing placeholders, and does not have to be parsed before matching.

7.5 term example

```
Python 1.3 (Jun  6 1996)  [GCC 2.6.3]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import TB
>>> T = TB.make("[1,2,3]")
>>> print T
<term, [1,2,3]>
>>> print T.list
[<term, 1>, <term, 2>, <term, 3>]
>>> L = T.match("[<int>,2,<term>]")
>>> print L
[1, <term, 3>]
>>> print L[1].str
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TB.error: term is not of type str
>>> print L[1].int
3
```

7.6 The hello example in Python: `hello.py`

Writing the hello tool in Python requires two steps:

- Write the required Python code `hello.py`. The result is shown in Figure 12.

- Replace `hello`'s tool definition in `hello2.tb` by:

```
tool hello is {command = "python-adapter -script hello.py"}
```

8 Writing tools in Tcl/Tk

There are two ways to connect tools written in Tcl/Tk (see Figure 13):

- Connect `wish`, Tcl/Tk's windowing shell, to the ToolBus via an adapter sends commands to `wish` via a pipe and receives the outpt of `wish` via another pipe. This strategy is used in the `wish-adapter` (available, but not further described in this guide).
- Use an adapter that is completely integrated with a Tcl/Tk interpreter. This is more efficient, but less flexible than the previous approach.

When the ToolBus distribution has been configured using the option `--with-tcltk=<tcl/tk-basepath>`, the `tcltk-adapter` is build.

8.1 `tcltk-adapter`

Synopsis. Execute Tcl/Tk's windowing shell `wish` as a tool.

Example. `tcltk-adapter -script calculator.tcl`

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`.

Communication. [13]

- `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-eval(`*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 anser 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.
- `snd-monitor(`*Trm*`)`: in this case the Tcl function `monitor_atom` {*ProcId AtFun, Src Blino Bpos Elino Epos*} is called where *ProcId* is the process-id of the process to which this atom belongs, *AtFun* is the action function name of the atom (printf, tau, snd-do, snd-eval, rec-msg etc.), *Src* the source file where the atom is defined, *Blino* the number of the line where the atom starts, *Bpos* the column of the line where the atom starts, *Elino* the number of the line where the atom ends and *Epos* is the column of the line where the atom ends (this information can be used for example for highlightning a piece of source code). After calling this function the term is further analyzed, possibly resulting in (several) other Tcl function calls. The following situations are considered:

[13] Communication is described from the point of view of the ToolBus, i.e., `snd-` and `rec-` mean, respectively, send by ToolBus and receive by ToolBus.

- *process creation*: `create_proc` {*ProcId ProcName*} is called.
- *tool creation*: `create_tool` {*ToolId ToolName*} is called.
- *process to tool communication*: `proc_tool_comm` {*ToolId ProcId*} is called.
- *tool to process communication*: `tool_proc_comm` {*ProcId ToolId*} is called.
- *process to process communication*: `proc_proc_comm` {*ProcId1 ProcId2*} is called.
- *update the value of a variable in a process*: `update_var` {*ProcId VarName NewValue*} is called.
- *update the list of subscribtions of a process*: `update_subs` {*ProcId Subs*} is called.
- *update the list of notes of a process*: `update_notes` {*ProcId Notes*} is called.

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).

8.2 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.

Note. 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.

8.3 The hello example in tcl: `hello.tcl`

Writing the hello tool in Tcl requires two steps:

- Write the required Tcl code `hello.tcl`. The result is shown in Figure 14.
- Replace `hello`'s tool definition in `hello2.tb` by:

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

Acknowledgements

Hayco de Jong and Pieter Olivier made major contributions to the documentation of the ToolBus in their "Aterm Library User Manual". Information from that manual has been used in this guide as well. Simon Gray and Mark van den Brand commented on drafts of this guide.

A Incompatibilities with older ToolBus versions

A.1 Include files, Libraries and API's

In older versions, tools had to include the file `TB.h`. Currently, this is `atb-tool.h`.

In older versions, the ToolBus API was provided by `libTB`. Currently, this is split over to libraries: `libATerm.a` (the ATerm functions) and `libATB.a` (the ToolBus API). As a consequence, older tools have to be compiled with the compiler flag `-lTB`.

In the older API's, all functions begin with the prefix `TB`. Currently, functions begin with either `AT` (ATerms) or `ATB` (ToolBus API).

The old and the new API's can be compared as follows:

Old	New
`TBinit`	`ATinit`
`TBmake`	`ATmake`
`TBmatch`	`ATmatch`
`TBwrite`	`ATwriteTerm`
`TBread`	`ATreadTerm`
`TBreadTerm`	—
`TBprintf`	`ATprintf`
`TBsprintf`	`ATsprintf`
`TBprotect`	`ATprotect`
`TBunprotect`	`ATunprotect`
`TBcollect`	—
`TBinit`	`ATBinit`
—	`ATBconnect`
—	`ATBdisconnect`
`TBaddTermPort`	—
`TBaddCharPort`	—
`TBreceive`	`ATBreadTerm`
`TBsend`	`ATBwriteTerm`
`TBeventloop`	`ATBeventloop`
`TBpeek`	`ATBpeekAny`
	`ATBpeekOne`
—	`ATBhandleOne`
—	`ATBhandleAny`
—	`ATBgetDescriptors`

In addition, the new API's provide other functions taht are not listed in this table.

B Limitations/extensions current implementation

The current implementation is a faithful implementation of the system described in "The Discrete Time ToolBus". There are some minor differences that are summarized here.

Extensions

- The types `<bstr>` and `<real>`.
- The atomic actions `printf` and `read`.

Limitations

- Certain functions in expressions have not yet been implemented (see Appendix D).
- The atomic actions `attach-monitor`, `detach-monitor`, and `reconfigure` have not yet been implemented.

C The syntax of T scripts

C.1 Preprocessor directives

The *script_name* given as argument to the ToolBus is always preprocessed by the C preprocessor before it is parsed as **T** script. In this way, directives like, e.g., `#define`, `#include` and `#ifdef` can be used freely in **T** scripts. We summarize the most frequently used directives:

- `#define` *`identifier token-sequence`* causes the preprocessor to replace all occurrences of *identifier* by *token-sequence*.
- `#define` *`identifier`* `(` *`identifier-list`* `)` *`token-sequence`* is a macro definition with parameters given by *identifier-list*. Textual occurrences of the identifier followed by an argument list containing an appropriate number of tokens separated by comma's will be replaced by *token-sequence* after parameter substitution.
- `#include "`*`filename`*`"` will be replaced by the entire contents of the named file.
- `#if`, `#ifdef`, and `#ifndef` can be used for the conditional incorporation or exclusion of parts of a script.

We refer to any ANSII C manual for a detailed description of these directives.

See Section 2.2 for a description of the preprocessor related arguments `-I`*dir*, `-D`*macro*, and `-D`*macro*`=`*defn* of the `toolbus` command.

C.2 Context-free syntax

```
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
```

D Expressions in T scripts

D.1 Boolean and arithmetic 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>$_1$ $\equiv$ <term>$_2$; for lists multi-set equality
not-equal(<term>$_1$, <term>$_2$)	<bool>	not(equal(<term>$_1$, <term>$_2$))
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>	absolute value $\mid$ <int>$_1$ $\mid$
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>$_2$
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>$_2$
rabs(<real>$_1$)	<real>	absolute value $\mid$<real>$_1\mid$
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>$_2$
sin(<real>$_1$)	<real>	sin(<real>$_1$)
cos(<real>$_1$)	<real>	cos(<real>$_1$)
atan(<real>$_1$)	<real>	tan^{-1}(<real>$_1$) in range $[-\pi/2, \pi/2]$
atan2(<real>$_1$, <real>$_2$)	<real>	tan^{-1}(<real>$_1$/<real>$_2$) in range $[-\pi, \pi]$
exp(<real>$_1$)	<real>	exponential function $e^{\texttt{<real>}_1}$
log(<real>$_1$)	<real>	natural logarithm ln(<real>$_1$), <real>$_1 > 0$
log10(<real>$_1$)	<real>	base 10 logarithm log_{10}(<real>$_1$), <real>$_1 > 0$
sqrt(<real>$_1$)	<real>	$\sqrt{\texttt{<real>}_1}$, <real>$_1 \geq 0$

D.2 Functions on lists and multi-sets

Function	Result type	Description
`first(<list>`$_1$`)`	`<term>`	first element of `<list>`$_1$; `[]` for non-lists
`next(<list>`$_1$`)`	`<list>`	remaining elements of `<list>`$_1$; `[]` for non-lists
`join(<term>`$_1$`,<term>`$_2$`)`	`<list>`	concatenation of `<term>`$_1$ and `<term>`$_2$; for a list argument `<term>`$_i$ ($i = 1, 2$), the list elements are spliced into the new list; non-list arguments are included as single element of the new list.
`size(<list>`$_1$`)`	`<int>`	$\lvert$`<list>`$_1\rvert$ (number of elements in list)
`index(<list>`$_1$`,<int>`$_1$`)`	`<term>`	If $\lvert$`<list>`$_1\rvert \leq$ `<int>`$_1$ return the `<int>`$_1$th element from `<list>`$_1$; otherwise `[]` and give a warning.
`replace(<list>`$_1$`,<int`$_1$`>,<term>`$_1$`)`	`<list>`	If $\lvert$`<list>`$_1\rvert \leq$ `<int>`$_1$ replace the `<int>`$_1$th element of `<list>`$_1$ by `<term>`$_1$ and return the modified (and partially copied) version of `<list>`$_1$; otherwise return `<list>`$_1$ and give a warning.
`get(<list>`$_1$`,<term>`$_1$`)`	`<term>`	If `<list>`$_1$ contains a pair `[<term>`$_1$`, <term>`$'_1$`]` then `<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>`$_1$.
`member(<term>`$_1$`,<list>`$_2$`)`	`<bool>`	`<term>`$_1 \in$ `<list>`$_2$ (membership in multi-set)
`subset(<list>`$_1$`, <list>`$_2$`)`	`<bool>`	`<list>`$_1 \subseteq$ `<list>`$_2$ (subset on multi-sets)
`diff(<list>`$_1$`, <list>`$_2$`)`	`<list>`	`<list>`$_1 -$ `<list>`$_2$ (difference on multi-sets)
`inter(<list>`$_1$`, <list>`$_2$`)`	`<list>`	`<list>`$_1 \cap$ `<list>`$_2$ (intersection on multi-sets)

D.3 Predicates and functions on terms

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>` equals `[]` 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 an application then its function symbol; otherwise `""`.
`args(<term>)`	`<list>`	If `<term>` is an application then its argument list; otherwise `[]`.

D.4 Miscellaneous functions

Function	Result type	Description
`process-id`	`<int>`	id of current process
`process-name`	`<str>`	name of current process
`quote(<term>)`	`<term>`	quoted (unevaluated) term, only variables are replaced by their value
`functions`	`<list>`	list of built-in functions
`current-time`	`<list>`	six-tuple describing current absolute time
`sec(<int>`$_1$`)`	`<int>`	convert `<int>`$_1$ in seconds
`msec(<int>`$_1$`)`†	`<int>`	convert `<int>`$_1$ in milli-seconds

†Not yet implemented in the current version

E Synopsis of primitives available in T scripts

Primitive	Description
`delta`	inaction (deadlock)
`tau`	internal step
P_1`+`P_2	choice
P_1`.`P_2	sequential composition
P_1`||`P_2	parallel composition
P_1`*`P_2	iteration
`if` T `then` P `fi`	guarded command
`if` T `then` P_1 `else` P_2 `fi`	conditional
`create(`Pnm`(`T`,...), `Pid`?)`	process creation[1]
V `:=` T	assignment, T expression (see D)
`snd-msg(`T`,...)`	send a message (binary, synchronous)
`rec-msg(`T`,...)`	receive a message (binary, synchronous)
`snd-note(`T`)`	send a note (broadcast, asynchronous)
`rec-note(`T`)`	receive a note (asynchronous)
`no-note(`T`)`	no notes available for process
`subscribe(`T`)`	subscribe to notes
`unsubscribe(`T`)`	unsubscribe from notes
`delay(`T`)`	relative time delay of atom
`abs-delay(`T`,...)`	absolute time delay of atom[2]
`timeout(`T`)`	relative timeout of atom
`abs-timeout(`T`,...)`	absolute timeout of atom[2]
`rec-connect(`Tid`?)`	receive a connection request from a tool
`rec-disconnect(`Tid`?)`	receive a disconnection request form a tool
`execute(`Tnm`(`T`,...), `Tid`?)`	execute a tool[1]
`snd-terminate(`Tid`, `T`)`	terminate the execution of a tool
`shutdown(`T`)`	terminate ToolBus
`reconfigure`	reconfigure ToolBus†
`attach-monitor`	attach a monitoring tool to a process†
`detach-monitor`	detach a monitoring tool from a process†
`snd-eval(`Tid`, `T`)`	send evaluation request to tool
`snd-cancel(`Tid`)`	cancel an evaluation request to tool†
`rec-value(`Tid`, `T`)`	receive a value from a tool
`snd-do(`Tid`, `T`)`	send request to tool (no return value)
`rec-event(`Tid`, `T`, ...)`	receive event from tool
`snd-ack-event(`Tid`, `T`)`	acknowledge a previous event from a tool
`printf(`S`, `T`, ...)`	print terms (after variable replacement) according to format S
`read(`T_1`, `T_2`)`	give prompt T_1, read term, should match with T_2
`process` Pnm`(`F`, ...) is` P	process definition[3]
`let` F`, ... in` P `endlet`	declare variables in P
`tool` Tnm`(`F`,...) is {` $Feat$`, ... }`	tool definition[3]
`host =` Str	host feature in tool definition
`command =` Str	command feature in tool definition
`details = <<` $Lines$ `>>`	details feature in tool definition
`toolbus(`Pnm`(`T`,...), ...)`	ToolBus configuration

Notes

[1]	`(`T`, ...)` is optional
[2]	Absolute time described by a 6-tuple ($year$, $month$, day, $hour$, $minutes$, $seconds$) with $year \geq 95$, $1 \leq month \leq 12$, $1 \leq day \leq 31$, $0 \leq hour \leq 23$, $0 \leq minutes \leq 59$, and $0 \leq seconds \leq 61$ (seconds can be greater than 59 to allow leap seconds). Absolute time may be abbreviated, by omitting, at most, the first three elements of the 6-tuple. Omitted elements default to their current value.
[3]	`(`F`, ...)` is optional
†	Not yet implemented

Legendum

T	term
T`, ...`	list of terms separated by comma's
V	variable
F	declaration of formal or local variable of the form $V{:}Type$
P, P_1, P_2	process expression
Tid	tool identifier, a variable of type Tnm (with Tnm declared as tool name)
Tnm	tool name
Pnm	process name
Pid	process identifier, a variable of type `int`
Str	a string constant
$Lines$	list of lines

```
/*  hello.c -- hello tool in C */

#include <stdio.h>
#include <stdlib.h>
#include <aterm1.h>                       /* ATerms, level 1 interface */
#include <atb-tool.h>                     /* ToolBus tool interface */

ATerm hello_handler(int conn, ATerm inp)  /* Handle input from ToolBus */
{ ATerm arg, isig, osig;

  if(ATmatch(inp, "rec-eval(get-text)"))
    return ATmake("snd-value(text(\"Hello World, my first ToolBus tool in C!\n\"))");
  if(ATmatch(inp, "rec-terminate(<term>)", &arg))
    exit(0);
  if(ATmatch(inp, "rec-do(signature(<term>,<term>))", &isig, &osig)){
    return NULL;                          /* we don't do a signature check */
  }
  ATerror("hello: wrong input %t received\n", inp);
  return NULL;
}

int main(int argc, char *argv[])          /* main program of hello tool */
{ ATerm bottomOfStack;                    /* marks stack bottom for ATerms */

  ATBinit(argc, argv, &bottomOfStack);  /* initialize ToolBus library */
  if(ATBconnect(NULL, NULL, -1, hello_handler) >= 0){
    ATBeventloop();
  } else {
    fprintf(stderr, "hello: Could not connect to the ToolBus, giving up!\n");
    return -1;
  }
  return 0;
}
```

Figure 4: `hello.c`: simple C code for the `hello` tool.

```
/*  hello-gen.c  -- hello tool in C using generated interface hello.tif.c */

#include <stdlib.h>
#include "hello.tif.h"                    /* Include generated tool interface */

ATerm get_text(int conn)                  /* Generate a hello text */
{
  return ATmake("snd-value(text(\"Hello World, my first ToolBus tool in C!\n\"))");
}

void rec_terminate(int conn, ATerm msg)  /* Mandatory function to terminate tool */
{
  exit(0);
}

int main(int argc, char *argv[])          /* main program of hello tool */
{
  ATerm bottomOfStack;

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

}
```

Figure 5: `hello-gen.c`: C code for the `hello` tool using a generated tool interface.

```
/* hello3.tb -- hello script with explicit rec-connect */

process HELLO is
  let H : hello,               %% H will represent the hello tool
      S : str                  %% S is a string valued variable
  in
      rec-connect(H?) .        %% Connect to a hello tool, H gets a tool id as value
      snd-eval(H, get-text) .  %% Request a text from the hello tool
      rec-value(H, text(S?)) . %% Receive it, S gets the text as value
      printf(S)                %% Print it
  endlet

tool hello is {command = "hello"}

toolbus(HELLO)
```

Figure 6: `hello3`: `hello` application with `rec-connect`.

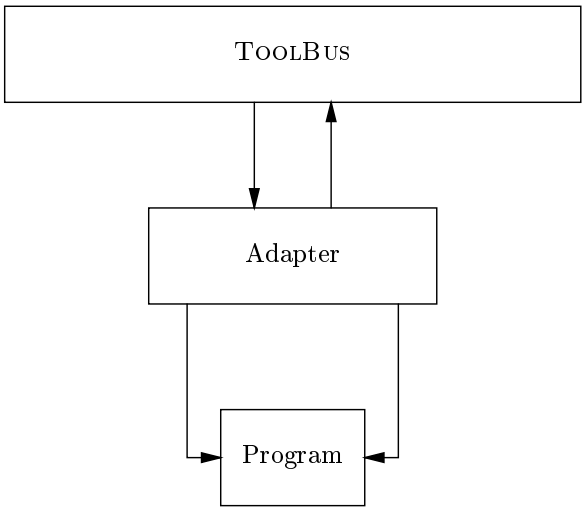

Figure 7: General structure of a tool adapter

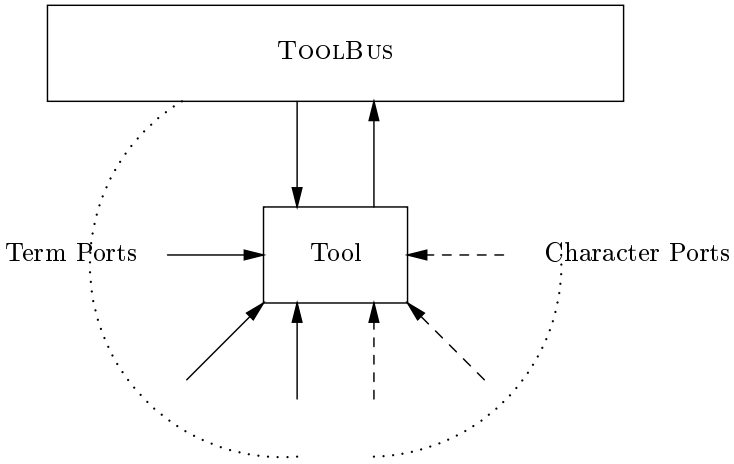

Figure 8: Global organization of a tool

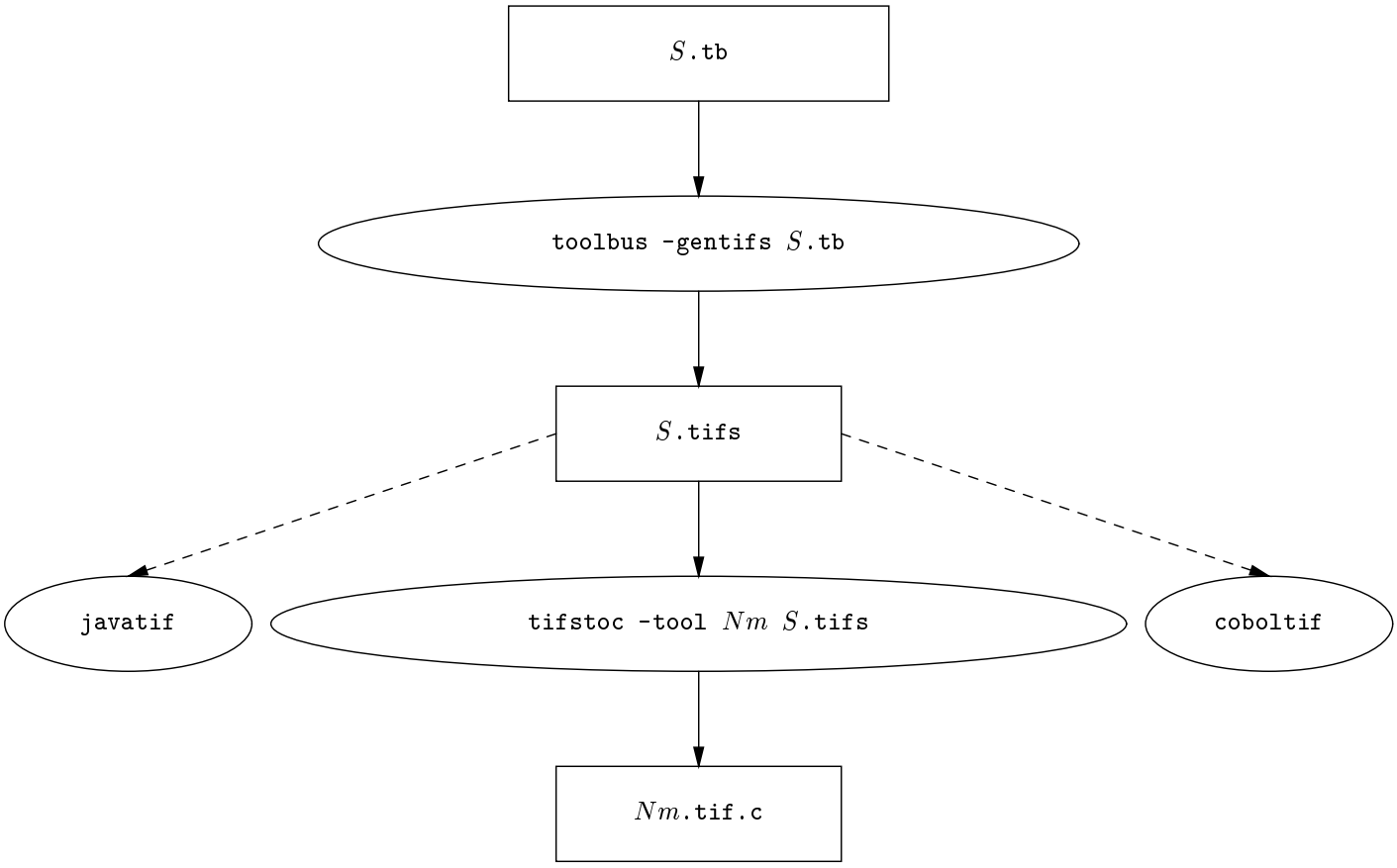

Figure 9: Automatic generation of tool interfaces.

```
/* pipe.tb -- Unix pipes simulated in a ToolBus script */

process PIPE(Tid : gen, Cmd1 : str, Inp : str, Cmd2 : str, Res : str?) is
  let Out1 : str
  in
     snd-eval(Tid, cmd(Cmd1, input(Inp))) .
     rec-value(Tid, output(Out1?)) .
     snd-eval(Tid, cmd(Cmd2, input(Out1))) .
     rec-value(Tid, output(Res?))
  endlet

process A is
  let Tid : gen, R : str
  in
     execute(gen, Tid?) .
     PIPE(Tid, "ls -l", "", "wc -w", R?) .
     printf(R)
  endlet

tool gen is {command = "gen-adapter"}
toolbus(A)
```

Figure 10: `pipe.tb`: Executing the pipe line `ls -l | wc` in a script.

```
# hello.perl -- hello tool in Perl

sub get_text {
   do TBsend("snd-value(\"Hello World, my first ToolBus tool in Perl!\n\")");
}

sub rec_terminate {
    local($n) = @_;

    exit(0);
}
```

Figure 11: `hello.perl`: the hello tool in Perl

```
# hello.py -- hello tool in Python

import TB
import sys
import __main__                 # Don't forget this one!

def rec_terminate(cid, A):
        sys.exit(0)

def get_text(cid):
        return TB.make("snd-value(text(<str>))",
           "Hello World, my first ToolBus tool in Python!\n")

cid = TB.parseArgs(sys.argv, __main__)
TB.connect(cid)
TB.eventloop()
```

Figure 12: `hello.py`: the hello tool in Python

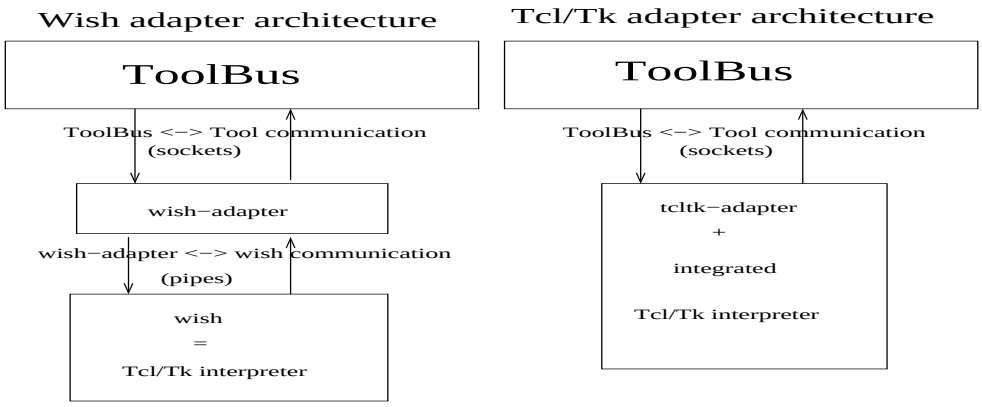

Figure 13: The wish-adapter and the tcltk-adapter

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

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

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

Figure 14: `hello.tcl`: the hello tool in Tcl