# The Discrete Time ToolBus

J.A. Bergstra[1,2] P. Klint[3,1]

[1] Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands
[2] Department of Philosophy, Utrecht University
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
[3] Department of Software Technology
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

## Abstract

Building large, heterogeneous, distributed software systems poses serious problems for the software engineer; achieving interoperability of software systems is still a major challenge.

In a previous paper, we have proposed 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".

Based on the experience with our previous proposal, we extend ToolBus scripts with features like conditionals and simple operations on the built-in data type of terms. More significantly, we introduce discrete time and give detailed descriptions of the protocol between ToolBus and tools. As a result, we can completely define the dynamic connection and disconnection of tools as well as notions like "monitoring" the execution of the ToolBus and "dynamic reconfiguration" of the ToolBus during execution.

These extensions are defined in a generic framework intended for the experimentation with new ToolBus features.

*1991 CR Categories*: C.2.4 [**Computer-communication networks**]: Distributed systems; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Tools and techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors; D.4.1 [**Operating Systems**]: Process management; D.4.4 [**Operating Systems**]: Communications management; H.5.2 [**Information interfaces and presentation**]: User interfaces.

*1991 Mathematics Subject Classification*: 68N15 [*Software*]: Programming Languages; 68N20 [*Software*]: Compilers and generators; 68Q65 [*Theory of computing*]: Abstract data types; algebraic specification.

*Key Words & Phrases*: interoperability of software tools, modularity, control integration, client/server architectures, distributed systems, process interpreter, prototyping, software development methodology, randomized execution.

# Contents

# Chapter 1

# Introduction

Building large, heterogeneous, distributed software systems poses serious problems for the software engineer. Systems grow *larger* because the complexity of the tasks we want to automate increases. They become *heterogeneous* because large systems may be constructed by re-using existing software as components. It is more than likely that these components have been developed using different implementation languages and run on different hardware platforms. Systems become *distributed* because they have to operate in the context of local area networks.

It is fair to say that the *interoperability* of software components is essential to solve these problems. The question how to connect a number of independent, interactive, tools and integrate them into a well-defined, cooperating whole has already received substantial attention in the literature (see, for instance, [SvdB93]), and it is easy to understand why:

- by connecting existing tools we can reuse their implementation and build new systems with lower costs;
- by decomposing a single monolithic system into a number of cooperating components, the modularity and flexibility of the systems' implementation can be improved.

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

In a previous paper [BK94], we have proposed the use of process-oriented "ToolBus scripts" for describing tool interactions. This paper has two aims:

- Introducing a number of new features based on the experience with our previous proposal.
- Establishing a framework for the design of and experimentation with such new features.

Where [BK94] used a script interpreter in which all language features were built-in, we propose here an extensible approach that allows the arbitrary combination of language features thus yielding a "family" of component interconnection languages.

The plan for the paper is as follows. In the following sections[1] we first summarize current research in tool integration and relate it to our work. In Chapter 2 we will briefly describe the ToolBus architecture and all the features used in ToolBus scripts (or **T** scripts for short). Next, we give several annotated examples in Chapter 3. A number of common notions like terms, matching and substitution are defined in Chapter 4. The global definition framework is presented in Chapter 5, where also two examples of ToolBus interpreters are given. A list of ToolBus features for processes and tools are described in Chapter 6, respectively, Chapter 7. A discussion in Chapter 8 concludes the paper. In Appendices A–D and the Index we provide cross-reference information to facilitate reading the specifications in this paper. In Appendix E we describe a discrete time process algebra for the ToolBus.

[1]These sections are updated versions of Sections 1.2 and 1.3 of [BK94].

## 1.1 Current research in tool integration

### 1.1.1 Data integration

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

Instead of providing a general mechanism for representing the data in arbitrary applications, we will use a single, uniform, data representation based on term structures. A consequence of this approach is that *existing* tools will have to be encapsulated by a small layer of software that acts as an "adapter" between the tool's internal dataformats and conventions and those of the ToolBus.

### 1.1.2 Control integration

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

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

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

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

See, for instance, [TvR85, BJ94] for an overview of these and related issues in the context of distributed operating systems.

**General message passing.** The most advanced tool integration can be achieved in systems based on general message passing. In SunMicrosystems' ToolTalk [TOO92], data integration as well as generic message passing are available. For each tool the names and types of the incoming and outgoing messages are declared. However, a description of the message interactions between between tools is not possible.

Another system in this category is Polygen, described in [WP92], where a separate description is used of the permitted interactions between tools. From this description, *stubs*[2] are generated to perform the actual communication. The major advantage of this approach is that the tool interactions can be described independently from the actual, underlying, communication mechanisms. The major disadvantage of this particular approach is that the interactions are defined in an ad hoc manner, that precludes further analysis of the interaction patterns like, for instance, the study of the dead lock behaviour of the cooperating tools.

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

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

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

**Control integration in the ToolBus.** The control integration between tools is achieved by using process-oriented "**T** scripts" that model the possible interactions between tools. The major difference with other approaches is that we use one, formal, description of *all* tool interactions.

### 1.1.3 User-interface integration

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

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

[2] Small pieces of interfacing software.

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

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

## 1.2 The relation with Module Interconnection Languages

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

The approach to component interconnection to be presented in this paper, *concentrates* on these dynamic, behavioural, aspects of modules. It shares many of the objectives of the work on "formal connectors" [AG94], where (untimed) CSP is used to describe software architectures. Their work is more ambitious than ours, since it aims at describing *arbitrary* software architectures, while we use a fixed (bus-oriented) architecture. The mechanisms we use to configure our bus architecture are, however, more powerful than the ones described in [AG94] (i.e., dynamic process creation, dynamic connection and disconnection of components, time).

## 1.3 Our approach

### 1.3.1 Requirements and points of departure

Before starting a more detailed analysis of component integration, it is useful to make a list of our requirements and state our points of departure.

To get control over the possible interactions between software components ("tools") we forbid direct inter-tool communication. Instead, all interactions are controlled by a "script" that formalizes all the desired interactions among tools. This leads to a communication architecture resembling a hardware communication bus, and therefore we will call it a "TOOLBUS". Ideally speaking, each individual tool can be replaced by another one, provided that it implements the same protocol as expected by other tools. The resulting software architecture should thus lead to a situation in which tools can be combined with each other in many fashions. We replace the classical procedure interface (a named procedure with typed arguments and a typed result) by a more general *behaviour description*.

A "TOOLBUS script" should satisfy a number of requirements:

- It has a formal basis and can be formally analysed.
- It is simple, i.e., it only contains information directly related to the objective of tool integration.
- It exploits a number of predefined communication primitives, tailored towards our specific needs. These primitives are such, that the common cases of deadlock can be avoided by adhering to certain styles of writing specifications.
- The manipulation of *data* should be completely transparent, i.e., data can only be received from and sent to tools, but inside the TOOLBUS there are no operations on them.

- There should be no bias towards any implementation language for the tools to be connected. We are at least interested in the use of C, Lisp, Tcl, and ASF+SDF for constructing tools.
- It can be mapped onto an efficient implementation.

### 1.3.2 The TOOLBUS

Compared with other approaches, the most distinguishing features of the TOOLBUS approach are:

- The prominent role of primitives for process control in the setting of tool integration. The major advantage being that complete control over tool communication can be achieved.
- The use of time primitives.
- The absence of user-defined datatypes. Compare this with the abstract datatypes in, for instance, LOTOS [Bri87], PSF [MV90, MV93], and $\mu$CRL [GP90]. We only depend on a free algebra of terms and use matching to manipulate data. Only a small set of built-in operations on terms is provided. Transformations on data can only be performed by tools, giving opportunities for efficient implementation.

# Chapter 2

# Overview of the ToolBus architecture

## 2.1 Global architecture

The global architecture of the ToolBus is shown in figure 2.1. 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 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 usually informs 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

[1]By "processes" we mean here computational activities *inside* the ToolBus as opposed to, for instance, processes at the operating system level. When confusion might arize, we will call the former ToolBus processes" and the latter "operating system level processes". Typically, the whole ToolBus will be implemented as a single operating system level process. This is also the case for each tool connected to the ToolBus.

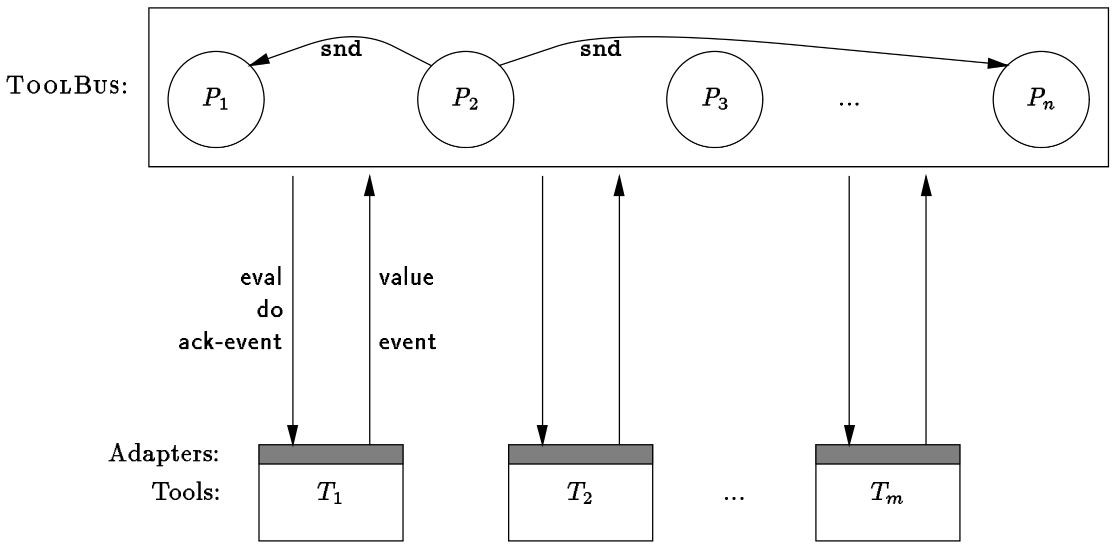

Figure 2.1: Global organization of the ToolBus

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.

## 2.2 Types and terms

The only values that can be exchanged between the ToolBus and tools or can be manipulated inside the ToolBus are *terms*: prefix expressions like, for example, `true`, `add(3, mul(4,5))`, and `pair("eva", age(7))`. In some cases, the full generality of these terms can be used, but there are many cases that it is better to formulate constraints that characterize the more specific forms of terms that are expected. For instance, when maintaining a counter we know in advance that its only permitted values are integers (and not arbitrary terms). As in many programming languages, we introduce a notion of *type* to express such constraints.

Variables appearing in ToolBus scripts will have to be declared to be of some *type* and it will be enforced that only terms of the appropriate type will be assigned to variables.

We will use the following *types*:

- `bool` is a type and represents Boolean values.
- `int` is a type and represents integer values.
- `str` is a type and represents string values (strings of characters).
- `list` is a type and represents lists whose elements may have arbitrary types.
- `list(`*Type*`)` is a type and represents lists whose elements are of type *Type*.
- `term` is a type and represents arbitrary terms.
- A single identifier *Id* is a type and represents any constant term with function symbol *Id*.
- *Id*`(`$Type_1$`, `$Type_2$`, ...)` is a type, provided that $Type_1$, $Type_2$, ... are also types. It represents terms of the form *Id*`(`$Term_1$`, `$Term_2$`, ...)` such that $Term_i$ is $Type_i$.

- `[`*Type*$_1$`,` *Type*$_2$`, ...]` is a type, provided that *Type*$_1$, *Type*$_2$, ... are also types. It represents terms of the form `[`*Term*$_1$`,` *Term*$_2$`, ...]` such that *Term*$_i$ is *Type*$_i$.

We define *terms* as follows:

- A Boolean value *Bool* is a term.
- An integer value *Int* is a term.
- A string value *String* is a term.
- A variable *Var* is a term. We enforce the convention that variables start with an uppercase letter.
- A result variable *Var*? is a term.
- A single identifier *Id* is a term. We enforce the convention that identifiers start with a lowercase letter.
- An application *Id*`(`*Term*$_1$`,` *Term*$_2$`, ...)` is a term, provided that *Term*$_1$, *Term*$_2$, ... are also terms.
- A list `[`*Term*$_1$`,` *Term*$_2$`, ...]` is a term, provided that *Term*$_1$, *Term*$_2$, ... are also terms.
- A placeholder <*Type*> is a term.

The *basic data types* Boolean, integer, and string are standard, and we will not elaborate on them in this document.

We distinguish two kinds of *occurrences of variables*:

- *Value occurrences* of the form *V* whose value is obtained from the context in which they are used.
- *Result occurrences* of the form *V*? who get a value assigned depending on the context in which they occur; this may be either as a result of a successful match with another term, or as a result of an assignment.

For instance, in a context where variable `X` has value `3`, the term `f(X)` is equivalent to `f(3)`. When, on the other hand, the terms `f(X?)` and `f(3)` are matched, the value `3` will be assigned to variable `X` as a result of this successful match.

*Placeholders* are intended to define *term patterns* in which certain positions are marked with the required type at that position. For instance, `add(<int>,<int>)` defines the type of a function `add` with two arguments of type `int`.

## 2.3 T scripts

A "ToolBus script" (or **T** script, for short) describes the complete behaviour of a system. A script consists of the parallel composition of a number of process names, each defined by a process expression. We start by defining "minimal **T** scripts" to which various extensions will be made later on.

### 2.3.1 Minimal T scripts

Minimal **T** scripts consist of the notions of atomic process, process expression, process definition, and tool definition. The only *atomic processes* available are:

- `delta`: the atomic process corresponding to inaction,[2] mainly used for representing process termination.
- `tau`: an internal step of a process.

[2]Also know as "deadlock", this explains the use of the (greek) letter "d".

*Composite process expressions* may have the following form:

- An atomic process.
- $P_1$ `.` $P_2$: the *sequential composition* of process expression $P_1$ and process expression $P_2$, i.e., $P_1$ followed by $P_2$.
- $P_1$ `+` $P_2$: the *choice* between process expression $P_1$ and process expression $P_2$. Note that "+" has lower precedence than "`.`".
- A *process invocation* has the form *Pname*(*Actuals*). *Pname* will be replaced by its definition in which formal parameter names are first replaced by their corresponding actual values. The actual values should correspond with the declared formal parameters in number and type. Recursive invocations are not allowed.

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

`process` *Pname* (*Formals*) `is` *P*

*Formals* are optional and contain a list of formal parameter names (and their types).
A *tool definition* can define a tool as follows:

`tool` *name* (*Formals*) `is` { ... }

A tool has a *name*, formal parameters, and is characterized by a number of features: a list of (identifier, string) pairs. Before a tool is executed, occurrences of formal parameter names in the strings defining features are replaced by their actual value.

Our general approach here is that a tool definition should contain all information needed to execute an instance of the tool, but we do not specify how a tool instance comes into existence. The interpretation of the names of features is therefore not fixed here, but in the examples we will assume the following feature names:

- *command*: the command needed to start the execution of a tool at the operating system level;
- *host*: the computer on which the tool will be executing.

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

`toolbus(`$Pname_1$`(`$Formals_1$`), ...,` $Pname_n$`(`$Formals_n$`))`

A complete **T** *script* consists of a list of process and tool definitions followed by a single TOOLBUS configuration.

### 2.3.2 Other features available in T scripts

The following primitives will be defined as orthogonal extensions to minimal **T** scripts:

**Iteration** (Section 6.1)

- $P_1$ `*` $P_2$: zero or more *repetitions* of process expression $P_1$ followed by process expression $P_2$ (binary Kleene star). Note that "`*`" has a higher precedence than "`.`".

**Free merge** (Section 6.2)

- $P_1$ `||` $P_2$: the parallel composition of process expressions $P_1$ and $P_2$ inside one TOOLBUS process. Note that no communication is possible between $P_1$ and $P_2$ since this is only permitted between process expressions appearing in *different* TOOLBUS processes. This operator has a lower precedence than `+`.

**Introduction of variables** (Section 6.3)

- `let` $Var_1$`:`$Type_1$`, ... in` $P$ `endlet`: introduce new variables and their required type in process expression $P$. Variables are initialized to themselves, i.e., the initial value of $Var_i$ is the term $Var_i$`:`$Type_i$.

**Expressions** (Section 6.5)

- $V$ `:=` *Term*: assigns the result of evaluating *Term* to variable $V$. Variables occurring in *Term* are replaced by their current value. As a matter of principle, the number and meaning of function symbols that can be used in *Term* is extensible rather than being fixed here. Each ToolBus implementation may provide its own set of functions and we only require a function `functions` that gives a list of all available functions and their signature. In a minimal system, the value of `functions` will only contain its own signature, i.e., a function with name "`functions`", zero arguments, and a result of type `list`:

  ```
  [function(functions,<list>)]
  ```

  However, to be able to give meaningful examples the following function symbols will be given a predefined meaning in this specification:

  - Functions on Booleans: `not`, `and`, and `or`.
  - Functions on Integers: `add`, `sub`, `mul`, `mod`, `less`, `less-equal`, `greater`, `greater-equal`, `sec` (convert to seconds), and `msec` (convert to mill-seconds).
  - Functions on lists: `first`, `next`, `join`, `member`, `subset`, `diff`, `inter`, and `size`.
  - Miscellaneous: `equal`, `not-equal`, `process-id` (the identification of the current process), `process-name` (the name of the current process), `current-time` (the current absolute time), `quote` (literal term that is not evaluated, only the variables appearing in it are replaced by their value), and `functions` (gives a list of all function symbols, and their signature, that have a meaning in expressions).

  For this collection of functions the value of `functions` will be:

  ```
  [function(not(<bool>), <bool>),
   function(and(<bool>,<bool>), <bool>),
   function(or(<bool>,<bool>), <bool>),
   function(equal(<term>, <term>), <bool>),
   function(not-equal(<term>, <term>), <bool>),
   function(add(<int>,<int>), <int>),
   function(add(<int>,<int>), <int>),
   function(sub(<int>,<int>), <int>),
   function(mul(<int>,<int>), <int>),
   function(less(<int>,<int>), <bool>),
   function(less-equal(<int>,<int>), <bool>),
   function(greater(<int>,<int>), <bool>),
   function(greater-equal(<int>,<int>), <bool>),
   function(first(<list>), <term>),
   function(next(<list>), <list>),
   function(join(<list>,<list>), <list>),
   function(member(<term>,<list>), <bool>),
   function(subset(<list>, <list>), <bool>),
   function(diff(<list>, <list>), <list>),
   function(inter(<list>, <list>), <list>),
   function(size(<list>), <int>),
  ```

```
    function(process-id, <int>),
    function(process-name, <str>),
    function(quote(<term>), <term>),
    function(functions, <list>)]
```

**Conditionals** (Section 6.6)

- `if` *Term* `then` $P_1$ `else` $P_2$ `fi`: *Term* should evaluate to a Boolean value. If it evaluates to `true`, $P_1$ is executed, otherwise $P_2$ is executed.
- `if` *Term* `then` $P$ `fi`: *Term* should evaluate to a Boolean value and if it evaluates to `true`, $P_1$ is executed. Otherwise, this construct reduces to `delta` meaning that the process in which this conditional occurs can not further proceed and becomes inactive.

**Dynamic process creation** (Section 6.7)

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

**Discrete time, delay and timeout** (Sections 6.9 and 6.10) The following attributes can be attached to atomic processes, in order to define their behaviour in time:

- `delay`: relative execution delay.
- `abs-delay`: absolute execution delay.
- `timeout`: relative timeout for execution.
- `abs-timeout`: absolute timeout for execution.

We only permit the following combinations of these attributes:

- relative time: `delay`, `delay/timeout`, `timeout`.
- absolute time: `abs-delay`, `abs-delay/abs-timeout`, `abs-timeout`.

Other combinations, e.g., mixtures of relative and absolute time are forbidden. Note that time is determined by the actual clock time of the ToolBus and not by the clocks of the tools, since these may be executing on different computers and their clocks are likely to be in conflict with each other.

**Communication between ToolBus processes.** We make a distinction between *messages* (for binary, synchronous communication) and *notes* (for asynchronous broadcasting).

The primitives for messages are (Section 6.4):

- `snd-msg` and `rec-msg`: used for sending and receiving messages between two processes using synchronous communication. A `snd-msg` can communicate with exactly one `rec-msg` that matches the `snd-msg`'s argument list. Both atoms will assign values to result variables (marked with ?) appearing in their argument lists; these can be used later on in the process expression in which these atoms occur.

The primitives for notes are (Section 6.8):

- `subscribe` and `unsubscribe`: subscribe, respectively unsubscribe, to notes of a given form. A process will only receive notes to which it has subscribed.

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

**Primitives for communication with tools** (Section 7) The first group of primitives deals with the explicit execution and connection of tools. They require a *tool definition* in the script in order to define a mapping between the name of a tool as used in the script and the command needed to execute it.

- `execute`: start the execution of a tool.
- `rec-connect`: receive a request to establish a connection with a tool already executing outside the ToolBus.

The second group of primitives deals with the communication between ToolBus and tools:

- `snd-eval`: request a tool to evaluate a term. The first argument serves as the identification of the tool, while the second argument is the term to be evaluated.
- `rec-value`: receive from a tool the result of a previous evaluation request.
- `snd-cancel`: cancel a previous `snd-eval`.
- `snd-do`: request a tool to evaluate a term and ignore the resulting value.
- `rec-event`: receive an event from a tool. The first argument of `rec-event` is a tool identification. The second argument serves as an identification of the source of the event. The remaining, optional, arguments give the details of the event in question.
- `snd-ack-event`: send an acknowledgement to a previous event received from a source. The assumption is made (and enforced) that the next event from that particular source will not be sent before the previous one has been acknowledged. Since one tool can generate events with different sources, a certain internal concurrency in tools can be supported.

The third group of primitives deals with the monitoring of ToolBus processes by tools. We make a distinction between three kinds of monitors:

- *logger*s are intended for the non-interactive recording of the behaviour of the processes being monitored. Typical examples are system logging, generation of play back scripts, and the gathering of performance information and statistics.
- *viewer*s are intended for interactive, but *non-intrusive*, viewing of processes. The monitored processes wait for a continue message from the viewer before they proceed. Typical example is a non-intrusive tracer/debugger.
- *controller*s are intended for the interactive, intrusive, control over processes. The monitored processes wait for a continue message from the controller before they proceed. This continue message may contain modifications to be made to the internal state of the processes or even to their process expressions. Typical applications are intrusive debuggers, and applications that perform arbitrary computations that want to use (parts of) the facilities of the ToolBus during their computations.

The monitoring primitives are:

- **`attach-monitor`**: attach a monitoring tool to a process. Note that *any* tool can act as a monitor for any ToolBus process. As a result, information will be sent to the tool allowing the detailed monitoring of the execution of the process.
- **`detach-monitor`**: detach a monitoring tool from a process.

The fourth and last group of primitives deals with the termination and disconnection of tools:

- **`snd-terminate`**: terminate a currently executing tool.
- **`rec-disconnect`**: receive a request to disconnect a tool from the ToolBus(without terminating its execution).
- **`shutdown`**: terminate *all* currently executing tools as well as all ToolBus processes.
- **`reconfigure`**: reconfigure the ToolBus by reading a new script, selectively deleting processes and tools, and restarting the bus using the new script.

# Chapter 3

# Examples

## 3.1 A calculator

### 3.1.1 Informal description

Consider a calculator capable of evaluating expressions, showing a log of all previous computations, and displaying the current time. Concurrent with the interactions of the user with the calculator, a batch process is reading expressions from file, requests their computation, and writes the resulting value back to file.

The calculator is defined as the cooperation of five processes:

- The user-interface process `UI` can receive the external events `button(calc)`, `button(showLog)` and `button(showTime)`.

  After receiving the "calc" button, the user-interface is requested to provide an expression (probably by asked the user via a dialogue window). This may have two outcomes: `cancel` to abort the requested calculation or the expression to be evaluated. After receiving the "showLog" button all previous calculations are displayed.

  The external event `button(showTime)` leads to the display of the current time. The user-interface has the property that the "showTime" button can be pushed at any time, i.e. even while a calculation is in progress. This is reflected in the use of the merge operator (||) in the process definition.

- The calculation process `CALC` which depends on a tool `calc` for performing the actual calculations.
- A process `BATCH` that reads expressions from file (by way of a tool `batch`) calculates their value, and writes the result back on file.
- A process `LOG` that maintains a log of all calculations performed. Observe that `LOG` explicitly subscribes to "`compute`" notes.
- A process `CLOCK` that can provide the current time on request (by way of a tool `clock`).

This example illustrates the, not completely trivial, connection of a user-interface and various tools.

### 3.1.2 T script for calculator

We present a complete, annotated, listing of the **T** script for the calculator. The actual script is presented in a `typewriter` font, comments appear as ordinary (roman) text.

```
process CALC is
  let Tid : calc, E : str, V : int
  in
     execute(calc, Tid?) .
```

```
    ( rec-msg(compute, E?) . snd-eval(Tid, expr(E)) .
      rec-value(Tid, V?) .
      snd-msg(compute, E, V) . snd-note(compute(E, V))
    ) * delta
  endlet
```

We take a closer look at the definition of the `CALC` process. First, three typed variables are introduced: `Tid` (of type `calc`, a tool identifier representing the `calc`-tool, see below), `E` (a string variable representing the expression whose value is to be computed), and `V` (an integer variable representing the computed value of expressions). The first atom,

```
execute(calc, Tid?)
```

executes the `calc`-tool using the command (and optionally also the desired host computer) as defined in `calc`'s tool definition. The result variable `Tid` gets as value a descriptor of this particular execution of the `calc`-tool. All subsequent atoms (e.g., `snd-eval`, `rec-event`) that communicate with this tool instance will use this descriptor as first argument. Next, we encounter a construct of the form

( rec-msg(compute, E?) ... ) * delta

describing an infinite repetition of all steps inside the parentheses. Note that inaction (`delta`) will be avoided as long as there are other steps possible. Next, we see the atom

```
rec-msg(compute, E?)
```

for receiving a computation request from another process. Here, `compute` is a constant, and the variable `E` will get as value a string representing the expression to be computed. Next, an evaluation request goes to the `calc`-tool as a result of

```
snd-eval(Tid, expr(E))
```

The resulting value is received by

```
rec-value(Tid, V?)
```

Observe the combination of an ordinary variable `Tid` and a result variable `V`. Clearly, this atom should *only* match with a value event coming from the `calc`-tool that was executed at the beginning of the `CALC` process. It is also clear that `V` should get a value as a result of the match. A reply to the original request `rec-msg(compute, E?)` is then given by

```
snd-msg(compute, E, V)
```

and this is followed by the notification

```
snd-note(compute(E, V))
```

that will be used by the `LOG` process.
The definition for the `calc` tool is:

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

The string value given for `command` is the operating system level command needed to execute the tool. It may contain additional arguments as can be seen in the definition of the `ui`-tool below.

The user-interface is defined by the process `UI`. First, it executes the `ui`-tool and then it handles three kinds of buttons. Note that the buttons "calc" and "log" exclude each other: either the "calc" button or the "log" button may be pushed but not both at the same time. The "time" button is independent of the other two buttons: it remains enabled while any of the other two buttons has been pushed.

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

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

The treatment of each button is defined in a separate, auxiliary, process definition. They have a common structure:

- Receive an event from the user-interface.
- Handle the event (either by doing a local computation or by communicating with other ToolBus processes that may communicate with other tools).
- Send an acknowledgement to the user-interface that the handling of the event is complete.

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

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

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

The `BATCH` process executes the `batch` tool, reads expressions from file, computes their value by exchanging messages with process `CALC` and writes an (expression, value) pair back to a file.

```
process BATCH is
  let Tid : batch, E : str, V : int
  in
```

```
     execute(batch, Tid?) .
     ( snd-eval(Tid, fromFile). rec-value(Tid, expr(E?)) .
       snd-msg(compute, E). rec-msg(compute, E, V?).
       snd-do(Tid, toFile(E, V))
     ) * delta
  endlet

tool batch is {command = "batch"}
```

The `LOG` process subscribes to notes of the form `compute(<str>,<int>)`, i.e., a function `compute` with a string and an integer as arguments.

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

tool log is {command = "log"}
```

There are alternatives for the way in which the process definitions in this example can be defined. The `LOG` process can, for instance, be defined without resorting to a tool in the following manner:

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

Instead of storing the log in a tool we can use a variable (`TheLog`) for this purpose in which we maintain a list of pairs. We use the function "`join`" (list concatenation) to append a new pair to the list. Note that `join` operates on lists, hence we concatenate a singleton list consisting of the pair as single element. The process `CLOCK` executes the `clock` tool and answers requests for the current time.

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

tool clock is {command = "clock"}
```

Finally, we define one of the possible ToolBus configurations that can be defined using the above definitions:

```
toolbus(UI, CALC, LOG1, CLOCK, BATCH)
```

## 3.2 A distributed auction

### 3.2.1 Informal description

Consider a completely distributed auction in which the auction master (auctioneer) and the bidders are cooperating via a workstation in their own office. The problem is how to synchronize bids, how to inform bidders about higher bids, and how to decide when the bidding is over. In addition, bidders may connect and disconnect from the auction whenever they want.[1]

The auction is defined by the following processes:

- The auction is initiated by the process `Auction` which executes the "master" tool (the user-interface used by the auction master) and then handles connections and disconnections of new bidders, introduction of a new item for sale to the auction, and the actual bidding process. A `delay` is used to determine the end of the bidding activity per item.
- A `Bidder` process is created for each new bidder that connects to the auction; it describes the possible behaviour of the bidder.

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

### 3.2.2 T script for auction

The overall steps performed during an auction are described by the process `Auction`.

```
process Auction is
  let Mid : master, Bid : bidder
  in
     execute(master, Mid?) .       %% execute the master tool
     ( ConnectBidder(Mid, Bid?)    %% repeat: add new bidder between sales
     +                             %%         or
       OneSale(Mid)                %%         perform one sale
     ) *
     rec-disconnect(Mid) .         %% until auction master quits
     shutdown("Auction is closed") %% close the auction
  endlet

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

The auxiliary process `ConnectBidder` handles the connection of a new bidder to the auction. It takes the following steps:

- Receive a connection request from some bidder. This may occur when someone executes a bidder tool outside the ToolBus (may be even on another computer). As part of its initialization, the bidder tool will attempt to make a connection with some ToolBus (the particular ToolBus is given as a parameter when executing the bidder tool).
- Create an instance of the process `Bidder` that defines the behaviour of this particular bidder.

[1]This example is an extension of the example given in [Yel94], where it was used in the context of protocol conversion and the generation of protocol adapters. We have added certain features, e.g., dynamic connection and disconnection of bidders and time considerations, to approximate the behaviour of a "real" auction.

- Ask the bidder for its name and send that to the auction master.

```
process ConnectBidder(Mid : master, Bid : bidder?) is
  let Pid : int, Name : str
  in
     rec-connect(Bid?) .            %% receive connection request from new bidder
     create(Bidder(Bid), Pid?) .    %% create a new Bidder process
     snd-eval(Bid, get-name) .      %% ask bidder for its name, and send it
     rec-value(Bid, Name?) .        %% to the master tool
     snd-do(Mid, new-bidder(Bid, Name))
  endlet
```

The auxiliary process `OneSale` handles all steps needed for the sale of one item:

- Receive an event from the master tool announcing a new item for sale.
- Broadcast this event to all connected bidders and perform one of the following steps as long as the item is not sold:
  - receive a new bid;
  - connect a new bidder;
  - ask for a final bid if no bids were received during the last 10 seconds;
  - declare the item sold if no new bids arrive within 10 seconds after asking for a final bid.

The process definition is:

```
process OneSale(Mid : master) is
  let Descr : str,                  %% Description of current item for sale
      InAmount : int,               %% Initial amount for item
      Amount : int,                 %% Current amount
      HighestBid : int,             %% Highest bid so far
      Final : bool,                 %% Did we already issue a final call for bids?
      Sold : bool,                  %% Is the item sold?
      Bid : bidder                  %% New bidder tool connected during sale
  in
      rec-event(Mid, new-item(Descr?, InAmount?)) .
      HighestBid := InAmount .
      snd-note(new-item(Descr, InAmount)) .
      Final := false . Sold := false .
      ( if not(Sold) then
           rec-msg(bid(Bid?, Amount?)) .
           snd-do(Mid, new-bid(Bid, Amount)) .
           if less-equal(Amount, HighestBid) then
              snd-msg(Bid, rejected)
           else
              HighestBid := Amount .
              snd-msg(Bid, accepted) .
              snd-note(update-bid(Amount)) .
              snd-do(Mid, update-highest-bid(Bid, Amount)) .
              Final := false
           fi
        fi
      +
        if not(or(Final, Sold)) then
           snd-note(any-higher-bid) delay(sec(10)) .
           Final := true
        fi
      +
        if and(Final, not(Sold)) then
           snd-note(sold(HighestBid)) delay(sec(10)) .
           Sold := true
        fi
      +
        ConnectBidder(Mid, Bid?) .  %% add new bidder during a sale
        snd-msg(Bid, new-item(Descr, HighestBid)) .
        Final := false
      ) *
      if Sold then snd-ack-event(Mid, new-item(Descr, InAmount)) fi
  endlet
```

The `Bidder` process defines the behaviour of one bidder.

```
process Bidder(Bid : bidder) is
  let Descr : str,                  %% Description of current item for sale
      Amount : int,                 %% Current amount
      Acceptance : term             %% Acceptance/rejection of our last bid
  in
     subscribe(new-item(<str>, <int>)) . subscribe(update-bid(<int>)) .
     subscribe(sold(<int>)) . subscribe(any-higher-bid) .
     ( ( rec-msg(Bid, new-item(Descr?, Amount?))
       +
         rec-note(new-item(Descr?, Amount?))
       ) .
       snd-do(Bid, new-item(Descr, Amount)) .
       ( rec-event(Bid, bid(Amount?)) .
         snd-msg(bid(Bid, Amount)) . rec-msg(Bid, Acceptance?) .
         snd-do(Bid, accept(Acceptance)) . snd-ack-event(Bid, bid(Amount))
       +
         rec-note(update-bid(Amount?)) . snd-do(Bid, update-bid(Amount))
       +
         rec-note(any-higher-bid) . snd-do(Bid, any-higher-bid)
       +
         rec-disconnect(Bid) . delta
       ) *
       rec-note(sold(Amount?)) . snd-do(Bid, sold(Amount))
     )
     * delta
  endlet

tool bidder(Name : str) is
  { command = "wish-adapter -script bidder.tcl -script-args -name Name" }
```

The complete auction is, finally, defined by the ToolBus configuration:

```
toolbus(Auction)
```

## 3.3 Simulation of the one-dimensional wave equation

### 3.3.1 Informal description

In [KPvW95], an algorithm is described for computing the one-dimensional wave equation that models, for instance, vibrations in a string. It computes the wave amplitudes $y_i(t)$ at sample point $i$ $(1 \leq i \leq N-1)$

on the $x$-axis and sample moment $t$ by introducing a processor per sample point using the following definitions:

$$y_i(t + \Delta t) = F(y_i(t), y_i(t - \Delta t), y_{i-1}(t), y_{i+1}(t)) \tag{3.1}$$

and

$$F(z_1, z_2, z_3, z_4) = 2z_1 - z2 + (c\frac{\Delta t}{\Delta x})^2 (z_3 - 2z_1 + z_4), \tag{3.2}$$

where $\Delta x$ is the (small) interval between the sampling points and $c$ is a constant representing the propagation velocity of the wave in transversal direction.

We present a **T** script that performs precisely this computation. It consists of the following processes and tools:

- The auxiliary process `F` is used to compute function $F$ defined above. We assume, in this example, that terms may also be floating point numbers and that the functions `radd` (real addition), `rsub` (real subtraction), and `rmul` (real multiplication) are available in expressions.
- The process `P` models a processor per sample point. It holds two values `D` and `E` representing, respectively, the amplitude in the sample point at time $t - \Delta t$ and $t$.
- The process `Pend` models the end points of the string.
- The process `MakeWave` constructs $N$ connected instances of `P` and two end points.
- A tool `display` is used to visualize the progress of the simulation.

This example illustrates the use of the ToolBus for simulation purposes.

### 3.3.2 T script for wave equation

Compute the function $F$. Note the use of the result parameter `Res`.

```
process F(Z1 : real, Z2 : real, Z3 : real, Z4 : real, Res : real?) is
  let CdTdX2 : real
  in
     CdTdX2 := 0.01 .                                %% arbitrary value for (c dt/dx)^2
     Res := radd(rsub(rmul(2.0, Z1), Z2),            %% 2z1 - z2 +
                 rmul(CdTdX2,                        %%     (c dt/dx)^2 *
                    radd(rsub(Z3, rmul(2.0, Z1)), Z4))) %%     (z3 - 2z1 + z4)
  endlet
```

Process `P` describes the behaviour of sample point `I` with left neighbour `L` and right neighbour `R`. The amplitude in point `I` at time $t - \Delta t$ and $t$ is, respectively `D` and `E`. The current amplitude in point `I` is written to display tool `Tid`. The global behaviour of `P` is:

- Receive the amplitudes of both neighbours.
- Send the amplitude `E` to both neighbours.
- Compute the new amplitude `E` at $t + \Delta t$ using auxiliary process `F` defined above.
- Repeat these steps.

```
process P(Tid : display, L : int, I : int, R : int, D : real, E : real) is
  let AL : real, AR : real, D1 : real
  in
     ( (  rec-msg(L, I, AL?)        %% receive amplitude of left neighbour
       || rec-msg(R, I, AR?)        %% receive amplitude of right neighbour
       || snd-msg(I, L, E)          %% send own amplitude to left neighbour
       || snd-msg(I, R, E)          %% send own amplitude to right neighbour
       || snd-do(Tid, update(I, E)) %% update own amplitude on the display
       ) .
       D1 := E .
       F(E, D, AL, AR, E?) .
       D := D1
     ) * delta
  endlet
```

Define the processes at the end points. `I` is the index of the end point, `NB` is its immediate neighbour.

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

Construct the processes $\mathtt{Pend}_0$, $\mathtt{P}_1$, ..., $\mathtt{P}_{N-1}$, $\mathtt{Pend}_N$.

```
process MakeWave(N : int) is
  let Tid : display, Id : int, I : int, L : int, R : int
  in
    execute(display, Tid?) .          %% create the display
    snd-do(Tid, mk-wave(N)) .         %% make an N point wave
    create(Pend(Tid, 0, 1), Id?).     %% create left end point
    L := sub(N,1) .
    create(Pend(Tid, N, L), Id?) .    %% create right end point
    I := 1 .                          %% create the P's in between
    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 *
    delta
  endlet
```

Define the `display` tool.

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

Define the initial ToolBus configuration.

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

# Chapter 4

# Elementary notions used in the ToolBus specification

As a preparation for the definition of the specification to be presented later on, we need the following common notions:

- *Terms*: the basic term structures used (Section 4.1) and their typed variant (Section 4.2). Several utility functions on terms are defined in Section 4.3.
- *Environments*: for associating values with variables (Section 4.4).
- *Matching and substitution*: for determining the match (or mismatch) between terms, and for replacing, in terms, variables by their value (Section 4.5).

## 4.1 Terms

The sort TERM will represent terms constructed from Booleans, integers, strings, prefix functions (with or without arguments), lists of terms, variables, and term patterns as already explained in Section 2.2. We assume the existence of appropriate modules Booleans and Integers; they are not further presented here.

```
Module Terms
imports Integers(4.1)
exports
  sorts ID VNAME STRING TERM TERM-LIST VAR GEN-VAR
  lexical syntax
    [␣\t\n]               → LAYOUT
    "%%"~[\n]*            → LAYOUT
    [a-z][A-Za-z0-9\-]*   → ID
    "\""~[\"]*"\""        → STRING
  context-free syntax
    BOOL                     → TERM
    INT                      → TERM
    STRING                   → TERM

    VNAME                    → VAR

    VAR                      → GEN-VAR
    VAR "?"                  → GEN-VAR
    GEN-VAR                  → TERM
    "<" TERM ">"           → TERM
    ID                     → TERM
    ID "(" TERM-LIST ")"   → TERM
    {TERM ","}*            → TERM-LIST
    "[" TERM-LIST "]"      → TERM
  variables
    T [0-9']*         → TERM
    Ts [0-9']*        → {TERM ","} +
    OptTs [0-9']*     → {TERM ","}*
    Vname [0-9']*     → VNAME
    Var [0-9']*       → VAR
    Vars [0-9']*      → {VAR ","}*
    GenVar [0-9']*    → GEN-VAR
    GenVars [0-9']*→  {GEN-VAR ","}*
    Id [0-9']*        → ID
    Int               → INT
    String [0-9']*    → STRING
```

Note that the sort VNAME is defined here, but we postpone a definition of the actual syntactic form of variables until Section 5.1. The sort VNAME may therefore be considered as a parameter as the definition of terms.

## 4.2 Typed terms

The purpose of types is to control the use of and the assignment to variables (see Section 2.2). Technically, types and terms have exactly the same syntactic structure, except that types may not contain variables. For simplicity, we will represent types as terms and enforce this additional constraint.

```
Module TypedTerms
imports  Terms(4.1)
exports
  sorts TYPE
  context-free syntax
    TERM                        → TYPE
    VNAME ":" TYPE              → VAR
    outermost-type-of(TERM)     → TYPE
    has-no-vars(TYPE)           → BOOL
    require-type(TYPE, TERM)    → BOOL
  variables
    Type [0-9']*→ TYPE
equations
```

*Determine the outermost type of a term, i.e., the outermost function symbol of its type.*

| | | | |
|---|---|---|---|
| outermost-type-of(*Bool*) | = | bool | [out-type-of-1] |
| outermost-type-of(*Int*) | = | int | [out-type-of-2] |
| outermost-type-of(*String*) | = | str | [out-type-of-3] |
| outermost-type-of(*Vname*) | = | term | [out-type-of-4] |
| outermost-type-of(*Vname* : *Type*) | = | *Type* | [out-type-of-4] |
| outermost-type-of(*Vname* : *Type* ?) | = | *Type* | [out-type-of-5] |
| outermost-type-of(*Id*) | = | *Id* | [out-type-of-6] |
| outermost-type-of(*Id*(*OptTs*)) | = | *Id* | [out-type-of-7] |

$\text{outermost-type-of}([\mathit{OptTs}]) = \texttt{list}$ [out-type-of-7]

$\text{outermost-type-of}(< T >) = \texttt{out-type-of}(T)$ [out-type-of-8]

*Check that a term does not contain variables.*

$\text{has-no-vars}(\mathit{Bool}) = \text{true}$ [has-no-vars-1]

$\text{has-no-vars}(\mathit{Int}) = \text{true}$ [has-no-vars-2]

$\text{has-no-vars}(\mathit{String}) = \text{true}$ [has-no-vars-3]

$\text{has-no-vars}(\mathit{Var}) = \text{false}$ [has-no-vars-4]

$\text{has-no-vars}(\mathit{Id}) = \text{true}$ [has-no-vars-5]

$\text{has-no-vars}(\mathit{Id}(\mathit{OptTs})) = \text{has-no-vars}([\mathit{OptTs}])$ [has-no-vars-6]

$\text{has-no-vars}([T, \mathit{OptTs}]) = \text{has-no-vars}(T) \wedge \text{has-no-vars}([\mathit{OptTs}])$ [has-no-vars-7]

$\text{has-no-vars}([]) = \text{true}$ [has-no-vars-8]

$\text{has-no-vars}(< T >) = \text{has-no-vars}(T)$ [has-no-vars-9]

*Check that a term has a certain required type.*

$\text{require-type}(\texttt{bool}, \mathit{Bool}) = \text{true}$ [require-type-1]

$\text{require-type}(\texttt{int}, \mathit{Int}) = \text{true}$ [require-type-2]

$\text{require-type}(\texttt{str}, \mathit{String}) = \text{true}$ [require-type-3]

$\text{require-type}(\texttt{list}, [\mathit{OptTs}]) = \text{true}$ [require-type-4]

$\text{require-type}(\texttt{list}(T_1), [T_2, \mathit{OptTs}]) = \text{require-type}(T_1, T_2) \wedge \text{require-type}(T_1, [\mathit{OptTs}])$ [require-type-5]

$\text{require-type}(\texttt{list}(T), []) = \text{true}$ [require-type-6]

$\text{require-type}(\texttt{term}, T) = \text{true}$ [require-type-7]

$$\frac{\mathit{Id} = \text{outermost-type-of}(T)}{\text{require-type}(\mathit{Id}, T) = \text{true}}$$ [require-type-8]

$\text{require-type}(\mathit{Id}(\mathit{OptTs}), \mathit{Id}(\mathit{OptTs}')) = \text{require-type}([\mathit{OptTs}], [\mathit{OptTs}])$ [require-type-9]

$\text{require-type}([T_1, \mathit{OptTs}_1], [T_2, \mathit{OptTs}_2]) =$ [require-type-10]
$\text{require-type}(T_1, T_2) \wedge \text{require-type}([\mathit{OptTs}_1], [\mathit{OptTs}_2])$

$\text{require-type}([], []) = \text{true}$ [require-type-11]

$\text{require-type}(< T_1 >, T_2) = \text{require-type}(T_1, T_2)$ [require-type-12]

$\text{require-type}(T_1, T_2) = \text{false}$ **otherwise** [require-type-13]

## 4.3 Term utilities

**Module** TermUtils
**imports** Terms$^{(4.1)}$
**exports**
**context-free syntax**
ms-eq "(" TERM ";" TERM ")" $\rightarrow$ BOOL

```
append "(" TERM-LIST ";" TERM-LIST ")"                    → TERM-LIST
is-elem "(" TERM ";" TERM-LIST ")"                        → BOOL
ms-subset "(" TERM-LIST ";" TERM-LIST ")"                 → BOOL

ms-diff "(" TERM-LIST ";" TERM-LIST ")"                   → TERM-LIST
ms-inter "(" TERM-LIST ";" TERM-LIST ";" TERM-LIST ")"    → TERM-LIST
ms-size "(" TERM-LIST ")"                                 → INT
mk-term-list(TERM)                                        → TERM-LIST
```

**equations**

The sort TERM-LIST will represent lists of terms. Several of the operations on lists below (i.e., ms-eq, ms-subset, ms-diff, mk-inter) treat their argument list(s) as multi-set(s).

*Equality predicate.*

| | | | | |
|---|---|---|---|---|
| ms-eq($T$; $T$) | = | true | | [ms-eq-1] |
| ms-eq([$T$, $OptTs$]; [$OptTs_1$, $T$, $OptTs_2$]) | = | ms-eq([$OptTs$]; [$OptTs_1$, $OptTs_2$]) | | [ms-eq-2] |
| ms-eq([]; []) | = | true | | [ms-eq-3] |
| ms-eq($T_1$; $T_2$) | = | false | **otherwise** | [ms-eq-4] |

*Append.*

append($OptTs_1$; $OptTs_2$) = $OptTs_1$, $OptTs_2$ [append-1]

*Is-element-of predicate.*

| | | | |
|---|---|---|---|
| | is-elem($T$; $T$, $OptTs$) = | true | [is-elem-1] |
| $T \neq T'$ ⇒ | is-elem($T$; $T'$, $OptTs$) = | is-elem($T$; $OptTs$) | [is-elem-2] |
| | is-elem($T$; ) = | false | [is-elem-3] |

*Subset predicate.*

| | | | | |
|---|---|---|---|---|
| ms-subset(; $OptTs$) | = | true | | [ms-subs-1] |
| ms-subset($T$, $OptTs$; $OptTs_1$, $T$, $OptTs_2$) | = | ms-subset($OptTs$; $OptTs_1$, $OptTs_2$) | | [ms-subs-2] |
| ms-subset($OptTs_1$; $OptTs_2$) | = | false | **otherwise** | [ms-subs-3] |

*Set difference.*

| | | | | |
|---|---|---|---|---|
| ms-diff($OptTs_1$, $T$, $OptTs_2$; $T$, $OptTs_3$) | = | ms-diff($OptTs_1$, $OptTs_2$; $OptTs_3$) | | [ms-diff-1] |
| ms-diff($OptTs_1$; ) | = | $OptTs_1$ | | [ms-diff-2] |
| ms-diff($OptTs_1$; $T$, $OptTs_2$) | = | ms-diff($OptTs_1$; $OptTs_2$) | **otherwise** | [ms-diff-3] |

*Set intersection*

ms-inter($OptTs_1$, $T$, $OptTs_2$; $OptTs_3$, $T$, $OptTs_4$; $OptTs_5$) = [ms-inter-1]
ms-inter($OptTs_1$, $OptTs_2$; $OptTs_3$, $OptTs_4$; $OptTs_5$, $T$)

ms-inter($OptTs_1$; $OptTs_2$; $OptTs_3$) = $OptTs_3$ **otherwise** [ms-inter-2]

*Size (number of elements).*

| | | | |
|---|---|---|---|
| ms-size() | = | 0 | [ms-size-1] |
| ms-size($T$, $OptTs$) | = | ms-size($OptTs$) + 1 | [ms-size-2] |

Finally, we define a conversion operation that converts a term, if necessary, into a term list.

| | | | | |
|---|---|---|---|---|
| mk-term-list([$OptTs$]) | = | $OptTs$ | | [mk-tl-1] |
| mk-term-list($T$) | = | $T$ | **otherwise** | [mk-tl-2] |

## 4.4 Environments

*Environments* are needed for representing the values of variables. They are also used for representing variable bindings during the matching of terms.

```
Module Environments
imports  TypedTerms(4.2)
exports
  sorts  ENV ENTRY ENV-PAIR
  context-free syntax
    VAR "↦" TERM                       → ENTRY
    "[" {ENTRY ","}* "]"               → ENV
    assign(VAR, TERM, ENV)             → ENV
    assign1(VAR, TERM, TERM, ENV)      → ENV
    value(VAR, ENV)                    → TERM
    value1(VAR, TERM, ENV)             → TERM
    update(ENV, ENV)                   → ENV
    delete(TERM, ENV)                  → ENV

    "(" ENV "," ENV ")"                → ENV-PAIR
    nullEnvP                           → ENV-PAIR
    env1(ENV-PAIR)                     → ENV
    env2(ENV-PAIR)                     → ENV
    declared-type(VNAME, ENV)          → TYPE
    env2term(ENV)                      → TERM
    term2env(TERM)                     → ENV
  variables
    Entry [0-9']*   → ENTRY
    Entries [0-9']* → {ENTRY ","}*
    Env [0-9']*     → ENV
    EnvP [0-9']*    → ENV-PAIR
equations
```

Environments (ENV) consist of zero or more (VAR, TERM) pairs, on which the operations assign and value are defined yielding, respectively, an environment reflecting an assignment to a variable, and the current value of a variable.

$$\mathsf{assign}(\mathit{Var},\ T,\ [\mathit{Var} \mapsto T',\ \mathit{Entries}]) = \mathsf{assign1}(\mathit{Var},\ T,\ T',\ [\mathit{Entries}]) \qquad \textbf{[assign-1]}$$

$$\mathsf{assign}(\mathit{Var},\ T,\ []) = [\mathit{Var} \mapsto T] \qquad \textbf{[assign-2]}$$

$$\frac{\mathsf{assign}(\mathit{Var},\ T,\ [\mathit{Entries}]) = [\mathit{Entries}']}{\mathsf{assign}(\mathit{Var},\ T,\ [\mathit{Var}' \mapsto T',\ \mathit{Entries}]) = [\mathit{Var}' \mapsto T',\ \mathit{Entries}']} \quad \textbf{otherwise} \qquad \textbf{[assign-3]}$$

The actual modification of the environment is performed by the auxiliary function assign1 in which essentially two cases are distinguished. If the old value of the variable Var whose value is to be updated is *not* a result variable, then just replace the old value by the new one. If the old value is a result variable, then perform the assignment to that result variable in the remaining environment. In this manner, a mechanism resembling call-by-reference is defined that permits assignment to formal (result) parameters of processes that are visible in the invoking process.

$$\mathsf{assign1}(\mathit{Var},\ T,\ \mathit{Var}',\ [\mathit{Entries}]) \ = \ [\mathit{Var} \mapsto T,\ \mathit{Entries}] \qquad \textbf{[assign1-1]}$$

$$\frac{\mathsf{assign}(\mathit{Var}',\ T,\ [\mathit{Entries}]) = [\mathit{Entries}']}{\mathsf{assign1}(\mathit{Var},\ T,\ \mathit{Var}'\ ?\ ,\ [\mathit{Entries}]) = [\mathit{Var} \mapsto \mathit{Var}'\ ?,\ \mathit{Entries}']} \qquad \textbf{[assign1-2]}$$

$$\mathsf{assign1}(\mathit{Var},\ T,\ T',\ [\mathit{Entries}]) \ = \ [\mathit{Var} \mapsto T',\ \mathit{Entries}] \quad \textbf{otherwise} \qquad \textbf{[assign1-3]}$$

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

$\text{value}(Var, [Var \mapsto T, Entries]) = \text{value1}(Var, T, [Entries])$ **[value-1]**

$\text{value}(Var, []) = Var$ **[value-2]**

$\text{value}(Var, [Var' \mapsto T', Entries]) = \text{value}(Var, [Entries])$ **otherwise** **[value-3]**

$\text{value1}(Var, Var', Env) = \text{value}(Var', Env)$ **[value1-1]**

$\text{value1}(Var, Var'\ ?\ , Env) = \text{value}(Var', Env)$ **[value1-2]**

$\text{value1}(Var, T, Env) = T$ **otherwise** **[value1]**

The declared type of a variable can be retrieved from an environment in the following manner:

$\text{declared-type}(Vname, [Vname : Type \mapsto T, Entries]) = Type$ **[decl-type-1]**

$\text{declared-type}(Vname, []) = \texttt{term}$ **[decl-type-2]**

$\text{declared-type}(Vname, [Vname' : Type' \mapsto T, Entries]) =$ **[decl-type-3]**
$\text{declared-type}(Vname, [Entries])$ **otherwise**

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

$$\frac{\text{assign}(Var, T, [Entries']) = [Entries'']}{\text{update}([Var \mapsto T, Entries], [Entries']) = \text{update}([Entries], [Entries''])}$$ **[update-1]**

$\text{update}([], [Entries]) = [Entries]$ **[update-2]**

Delete a list of variables from an environment.

$\text{delete}([OptTs_1, Var, OptTs_2], [Entries_1, Var \mapsto T, Entries_2]) =$ **[delete-1]**
$\text{delete}([OptTs_1, OptTs_2], [Entries_1, Entries_2])$

$\text{delete}([], Env) = Env$ **[delete-2]**

$\text{delete}(T, Env) = Env$ **otherwise** **[delete-3]**

Introduce the notion of *environment pairs* to represent pairs of variable-bindings. The empty pair is defined as a useful value. In addition, two projection functions are defined on environment pairs.

$\text{nullEnvP} = ([], [])$ **[nullEnvP-1]**

$\text{env1}(([Entries_1], [Entries_2])) = [Entries_1]$ **[env1-1]**

$\text{env2}(([Entries_1], [Entries_2])) = [Entries_2]$ **[env2-1]**

For reasons that will only become clear in Section 7.7, we will define a two way mapping between environments and terms.

$$\frac{\text{env2term}([Entries]) = [OptTs]}{\text{env2term}([Var \mapsto T, Entries]) = [\texttt{entry}(Var, T), OptTs]}$$ **[e2t-1]**

$\text{env2term}([]) = []$ **[e2t-2]**

$$\frac{\text{term2env}([OptTs]) = [Entries]}{\text{term2env}([\texttt{entry}(Var, T), OptTs]) = [Var \mapsto T, Entries]}$$ **[t2e-1]**

$\text{term2env}([]) = []$ **[t2e-2]**

## 4.5 Matching and substitution

### Module Match

**imports** Environments$^{(4.4)}$ TermUtils$^{(4.3)}$ TypedTerms$^{(4.2)}$
**exports**
**context-free syntax**

```
substitute(TERM, ENV)          → TERM

nomatch                        → ENV-PAIR
match(TERM, ENV, TERM, ENV)    → ENV-PAIR
match1(TERM, TERM, ENV-PAIR)   → ENV-PAIR

cmatchp(TERM, TERM)            → BOOL
```

**equations**

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

$$\text{substitute}(\mathit{Var}, \mathit{Env}) = \text{value}(\mathit{Var}, \mathit{Env}) \qquad \textbf{[substitute-1]}$$

$$\frac{\text{substitute}([\mathit{OptTs}], \mathit{Env}) = [\mathit{OptTs}']}{\text{substitute}(\mathit{Id}(\mathit{OptTs}), \mathit{Env}) = \mathit{Id}(\mathit{OptTs}')} \qquad \textbf{[substitute-2]}$$

$$\frac{\text{substitute}(T, \mathit{Env}) = T',\ \text{substitute}([\mathit{OptTs}], \mathit{Env}) = [\mathit{OptTs}']}{\text{substitute}([T, \mathit{OptTs}], \mathit{Env}) = [T', \mathit{OptTs}']} \qquad \textbf{[substitute-3]}$$

$$\text{substitute}([\,], \mathit{Env}) = [\,] \qquad \textbf{[substitute-4]}$$

$$\text{substitute}(T, \mathit{Env}) = T \quad \textbf{otherwise} \qquad \textbf{[substitute-5]}$$

Given two terms, and two environments, first replace all variables by their respective values and then apply the auxiliary function match1. Observe that the case of multiple occurrences of the same variable in one of the terms is handled automatically by this preliminary substitution operation. The function match1 has as arguments the two terms resulting from the substitution just mentioned and two lists of variable bindings under construction (represented by an environment pair). match1 will fail if one of the terms contains an uninstantiated variable. When successful, match1 yields an environment pair representing the variable bindings needed to match the two terms. A failing match is represented by the (new) constant nomatch.

$$\frac{\begin{array}{c} T_1' = \text{substitute}(T_1, \mathit{Env}_1), \\ T_2' = \text{substitute}(T_2, \mathit{Env}_2) \end{array}}{\text{match}(T_1, \mathit{Env}_1, T_2, \mathit{Env}_2) = \text{match1}(T_1', T_2', \text{nullEnvP})} \qquad \textbf{[match-1]}$$

$$\text{match1}(\mathit{Bool}, \mathit{Bool}, \mathit{EnvP}) = \mathit{EnvP} \qquad \textbf{[match1-1]}$$

$$\text{match1}(\mathit{Int}, \mathit{Int}, \mathit{EnvP}) = \mathit{EnvP} \qquad \textbf{[match1-2]}$$

$$\text{match1}(\mathit{String}, \mathit{String}, \mathit{EnvP}) = \mathit{EnvP} \qquad \textbf{[match1-3]}$$

$$\frac{\begin{array}{c} [\mathit{Entries}_1] = \text{env1}(\mathit{EnvP}), \\ \text{value}(\mathit{Var}, [\mathit{Entries}_1]) = \mathit{Var}, \\ \mathit{Var} = \mathit{Vname} : \mathit{Type}, \\ \text{require-type}(\mathit{Type}, T_2) = \text{true} \end{array}}{\text{match1}(\mathit{Var}\,?\,, T_2, \mathit{EnvP}) = ([\mathit{Var} \mapsto T_2, \mathit{Entries}_1], \text{env2}(\mathit{EnvP}))} \qquad \textbf{[match1-4]}$$

$$\frac{\mathsf{value}(\mathit{Var},\ \mathsf{env1}(\mathit{EnvP})) = T_2}{\mathsf{match1}(\mathit{Var}\,?\ ,\ T_2,\ \mathit{EnvP}) = \mathit{EnvP}} \qquad \text{[match1-5]}$$

$$\frac{\begin{array}{c}[\mathit{Entries}_2] = \mathsf{env2}(\mathit{EnvP}),\\ \mathsf{value}(\mathit{Var},\ [\mathit{Entries}_2]) = \mathit{Var},\\ \mathit{Var} = \mathit{Vname} : \mathit{Type},\\ \mathsf{require\text{-}type}(\mathit{Type},\ T_1) = \mathsf{true}\end{array}}{\mathsf{match1}(T_1,\ \mathit{Var}\,?\ ,\ \mathit{EnvP}) = (\mathsf{env1}(\mathit{EnvP}),\ [\mathit{Var} \mapsto T_1,\ \mathit{Entries}_2])} \qquad \text{[match1-6]}$$

$$\frac{\mathsf{value}(\mathit{Var},\ \mathsf{env2}(\mathit{EnvP})) = T_1}{\mathsf{match1}(T_1,\ \mathit{Var}\,?\ ,\ \mathit{EnvP}) = \mathit{EnvP}} \qquad \text{[match1-7]}$$

$$\mathsf{match1}(\mathit{Id}(\mathit{OptTs}_1),\ \mathit{Id}(\mathit{OptTs}_2),\ \mathit{EnvP}) \ = \ \mathsf{match1}([\mathit{OptTs}_1],\ [\mathit{OptTs}_2],\ \mathit{EnvP}) \qquad \text{[match1-8]}$$

$$\frac{\begin{array}{c}\mathsf{match1}(T_1,\ T_2,\ \mathit{EnvP}) = \mathit{EnvP}_1,\\ \mathsf{match1}([\mathit{Ts}_1],\ [\mathit{Ts}_2],\ \mathit{EnvP}_1) = \mathit{EnvP}_2\end{array}}{\mathsf{match1}([T_1,\ \mathit{Ts}_1],\ [T_2,\ \mathit{Ts}_2],\ \mathit{EnvP}) = \mathit{EnvP}_2} \qquad \text{[match1-9]}$$

$$\begin{array}{lcll}\mathsf{match1}([T_1],\ [T_2],\ \mathit{EnvP}) & = & \mathsf{match1}(T_1,\ T_2,\ \mathit{EnvP}) & \text{[match1-10]}\\ \mathsf{match1}([],\ [],\ \mathit{EnvP}) & = & \mathit{EnvP} & \text{[match1-11]}\\ \mathsf{match1}(T,\ T,\ \mathit{EnvP}) & = & \mathit{EnvP} & \text{[match1-12]}\end{array}$$

$$\frac{\mathsf{require\text{-}type}(T_1,\ T_2) = \mathsf{true}}{\mathsf{match1}(<T_1>\ ,\ T_2,\ \mathit{EnvP}) = \mathit{EnvP}} \qquad \text{[match1-13]}$$

$$\frac{\mathsf{require\text{-}type}(T_2,\ T_1) = \mathsf{true}}{\mathsf{match1}(T_1,\ <T_2>\ ,\ \mathit{EnvP}) = \mathit{EnvP}} \qquad \text{[match1-13]}$$

$$\mathsf{match1}(T_1,\ T_2,\ \mathit{EnvP}) = \mathsf{nomatch} \qquad \textbf{otherwise} \qquad \text{[match1-13]}$$

The function cmatchp matches two closed terms, i.e., terms not containing any variables. Observe that match1 will always return nomatch when any of the terms does contain variables. The result is Boolean value.

$$\frac{\mathsf{match1}(T_1,\ T_2,\ \mathsf{nullEnvP}) = ([\mathit{Entries}_1],\ [\mathit{Entries}_2])}{\mathsf{cmatchp}(T_1,\ T_2) = \mathsf{true}} \qquad \text{[cmatchp-1]}$$

$$\mathsf{cmatchp}(T_1,\ T_2) = \mathsf{false} \qquad \textbf{otherwise} \qquad \text{[cmatchp]}$$

# Chapter 5

# A framework for the interpretation of T scripts

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

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

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

This is similar to the situation where a programming language contains equationally specified data types and randomized execution is needed to guarantee that the data type axioms can be used as correctness preserving transformations.

**Representing a ToolBus.** Our overall strategy is as follows. At any moment during interpretation each process, say process $k$, is represented as $\rho_k(\lambda_{Env}(<AP_1 + ... + AP_n>))$. Each $AP_i$ is an *action-prefix form*, i.e., a process expression starting with an action, and represents a possible choice in the process. $AP_i$ does not itself contain any $+$-operators. The operator $\lambda_{Env}$ represents the local state of $AP_k$ where $Env$ is a mapping from variables to their respective values. The operator $\rho$ represents a *renaming* that identifies all atoms as belonging to process $k$. All other information related to a process is maintained in a global *bus state* to be described in a moment.

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

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

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

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

For descriptive purposes, we also model the *tools* connected to the ToolBus as processes. The interpreter as a whole thus captures the input/output behaviour of the system described by the script: given a **T** script and events coming from the tools connected to the ToolBus, it computes responses modeled by messages to the connected tools.

We should—once more—emphasize that interpretation implies scheduling, which excludes certain possible execution paths. However, the path selected is among all possible paths as defined by its Process Algebra semantics.

**Specifying a ToolBus.** Our main concern is now to guarantee the *extensibility* of the interpretation method sketched above as well as of its specification. We have taken the following measures to ensure this:

- New atomic or composite processes can always be added by extending the appropriate sorts in the definition of **T** scripts (see Section 5.1: sorts ATOM, ATOMIC-FUN, and PROC).
- We use extensible record structures for representing the bus state (BUS-STATE). In this way, new information can be added when the interpretation of new features requires this.
- Two functions are defined on the bus state:
  - simple-atomic-step: for defining an atomic action in a single ToolBus process.
  - atomic-steps: for defining primitives that involve more than one process or tool.

  By extending the definitions of these functions, new features can be added.

The kernel of our interpretation framework consists of the following notions:

- *Minimal* **T** *scripts*: Section 5.1.
- *Prepare processes and terms*: a general preparation phase whose primary aim is to resolve names, i.e., postfix all variable names in process definitions with the name of the process definition in which they occur. This phase can, however, be used to define local preprocessing and transformation of terms, atoms and processes. (Section 5.2).
- *The bus state*: the representation of the complete ToolBus state (Section 5.3).
- *Action prefix form*: transform a process expression into a form that is well-suited for interpretation (Section 5.4).
- *The actual* ToolBus: represent the process behaviour of the ToolBus and define fundamental interpretation functions on it (Section 5.5).

Several ToolBus interpreters using this framework are described separately in Section 5.6.

**Structure of the specification.** An overview of the import structure of the specification is shown in Figure 5.1. The interpretation framework itself is represented by the solid boxes in the figure. All dashed boxes represent language features described in Sections 6 and 7. The dashed arrows entering the box labeled "interpreter(s)" represent imports of modules defining specific ToolBus features. By including or excluding such imports, interpreters for a complete family of ToolBus languages can be defined.

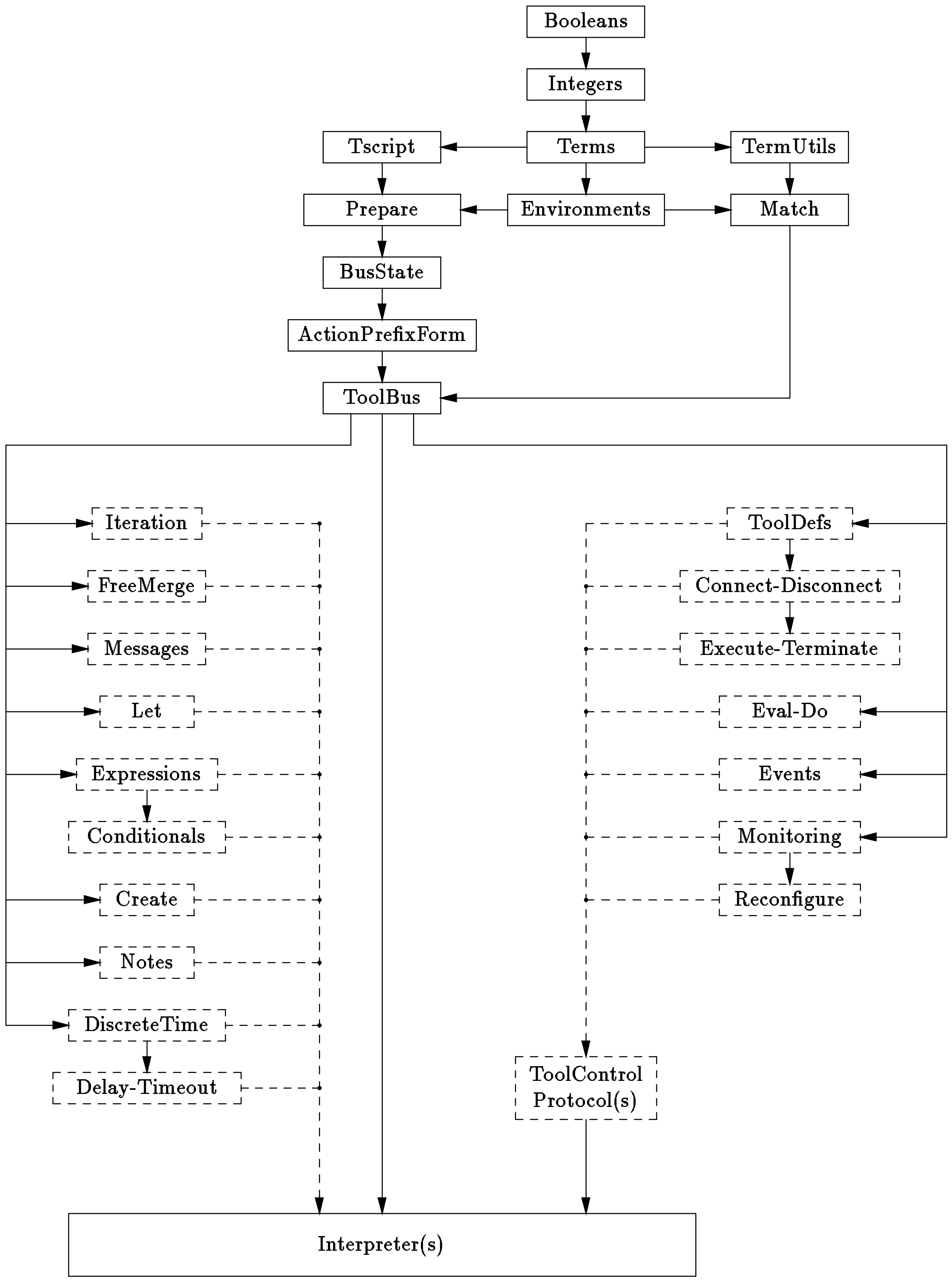

Figure 5.1: (Slightly simplified) import graph for ToolBus specification

**Rationale for the style of specification.** The specification presented here has the following characteristics:

- An attempt has been made to stay as close as possible to the concepts and notations of Process Algebra. When defining operations on process expressions, the standard approach is to *normalize* them, i.e., replace all operators by simpler ones thus obtaining a normal form containing a limited set of operators. The major advantage of this approach is simplicity, since more complex operators can be defined axiomatically in terms of simpler ones. Operationally, however, this approach is less suitable, since the resulting normal forms may become very large and their computation may be very expensive.

  In this specification we will use a fixed format of process expressions (as described above) and manipulate them directly, without normalisation. This approach can be characterized as "lazy" as opposed to "eager" normalisation before interpretation.

- A modular structure has been designed in which one feature is defined per module and modules can be combined in (nearly) arbitrary fashions. The result is an extensible framework, but certain artifacts of this modularisation are visible in the specification. For instance, we are forced to use a representation of the global state of the ToolBus that permits the addition of new fields later on. Without this modular structure, we could have used a simpler representation that contains all elements of the global state as fixed fields. Another artifact is the necessity to spread the definition of certain functions (e.g., atomic-step) across several modules rather than defining it in one module.

- The specification should also give a very clear guidance for an implementation in C. Although not immediately visible from the specification text itself, there is an obvious mapping from process algebra operators to record constructors. From that perspective they play the role of containers of information that is needed for the interpretation function.

**The process algebra primitives used in the ToolBus.** Process Algebra (ACP) is an algebraic approach to the description of parallel, communicating, processes originally proposed in [BK84]. We will use the axiom systems $BPA_\delta$, PA and ACP as well as operators for renaming [Vaa90], iteration [BBP94], process creation [Ber90], and state manipulation [BB88]. A summary of these axiom systems and operator definitions is given in Appendix E. We refer to [BW90] for an elaborate description of Process Algebra.

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

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

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

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

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

## 5.1 The syntax of minimal T scripts

Here we define the syntax of atoms (ATOM) and processes (PROC), process definitions (PROC-DEF), and ToolBus configurations (TB-CONFIG), and **T** scripts (T-SCRIPT).

Observe that we introduce here a sort for script variables (SVAR) that is used to extend the sort VAR of variables in terms (see Section 4.1).

This definition of **T** scripts is "minimal" in the sense that several of the sorts introduced here (i.e., **ATOM**, **ATOMIC-FUN**, **PROC**, **TOOL-DEF**) are either empty, or are only minimally defined here. They will all be extended later on.

A **T** script should satisfy the following static constraints:

- All names of defined processes should be different (even if they differ in number of formal parameters).
- All process names appearing in a ToolBus configuration or in any process definition should have been defined in some process definition.
- The number and type of actual parameters following a process name should be equal to the number and type of formal parameters in the corresponding process definition.
- Recursive process invocations are not allowed.

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

```
Module Tscript
imports  Terms(4.1)
exports
  sorts  ATOM ATOMIC-FUN PROC NAME PROC-APPL FORMALS
         TB-CONFIG DEF T-SCRIPT
  lexical syntax
    [A-Z][A-Za-z0-9\-]* → NAME
    delta               → ATOMIC-FUN
    tau                 → ATOMIC-FUN
  context-free syntax
    NAME                                → VNAME

    ATOMIC-FUN "(" TERM-LIST ")"        → ATOM
    ATOMIC-FUN                          → ATOM
    ATOM                                → PROC

    PROC " +" PROC                      → PROC        {left}
    PROC "." PROC                       → PROC        {right}
    "(" PROC ")"                        → PROC        {bracket}

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

    PROC-APPL                           → PROC

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

    process NAME FORMALS is PROC        → DEF
    toolbus "(" {PROC-APPL ","} + ")"   → TB-CONFIG
    DEF* TB-CONFIG                      → T-SCRIPT
  priorities
    PROC "."PROC → PROC > PROC " +"PROC → PROC
  variables
    Pnm [0-9']*        → NAME
    Name [0-9′]*        → NAME
    Vars [0-9′]*        → {VAR ","}*
    P [0-9′]*           → PROC
    Atom [0-9′]*        → ATOM
    AtomicFun [0-9′]*   → ATOMIC-FUN
    ProcAppls           → {PROC-APPL ","} +
    TB-Config           → TB-CONFIG
    T-Script [0-9′]*    → T-SCRIPT
    Script [0-9′]*      → T-SCRIPT
    Defs [0-9′]*        → DEF*
    Formals [0-9′]*     → FORMALS
    Chars               → CHAR*
exports
  context-free syntax
    "δ"                                  → ATOMIC-FUN
    fun(ATOM)                            → ATOMIC-FUN
    args(ATOM)                           → TERM
    γ(ATOMIC-FUN, ATOMIC-FUN)            → BOOL
    γ1(ATOMIC-FUN, ATOMIC-FUN)           → BOOL

    process-definition(NAME, T-SCRIPT)   → DEF

    parse(STRING)                        → T-SCRIPT
    proc2term(PROC)                      → TERM
    term2proc(TERM)                      → PROC
    atomic-fun2string(ATOMIC-FUN)        → STRING
    name2string(NAME)                    → STRING
equations
```

To gain maximal flexibility, we will never access atoms directly using their concrete syntactic form. Rather, we will obtain their function symbol and arguments through the following access functions fun and args. In this way, it will be possible to make extensions of the sort ATOM without affecting other modules in the specification.[1]

$$\text{fun}(\mathit{AtomicFun}(\mathit{OptTs})) = \mathit{AtomicFun} \qquad \textbf{[fun-std-1]}$$

$$\text{fun}(\mathit{AtomicFun}) = \mathit{AtomicFun} \qquad \textbf{[fun-std-2]}$$

$$\text{args}(\mathit{AtomicFun}(\mathit{OptTs})) = [\mathit{OptTs}] \qquad \textbf{[args-std-1]}$$

$$\text{args}(\mathit{AtomicFun}) = [] \qquad \textbf{[args-std-2]}$$

In ToolBus scripts one can use the constants `delta`, but in this specification it will be represented by the constant $\delta$.

$$\texttt{delta} = \delta \qquad \textbf{[delta-1]}$$

Define the communication function $\gamma$; it defines which atomic functions can communicate.

$$\gamma(\mathit{AtomicFun}_1, \mathit{AtomicFun}_2) = \qquad \textbf{[communication-1]}$$
$$\gamma_1(\mathit{AtomicFun}_1, \mathit{AtomicFun}_2) \vee \gamma_1(\mathit{AtomicFun}_2, \mathit{AtomicFun}_1)$$

Since $\gamma$ itself is commutative, we introduce an auxiliary (non-commutative) version of it ($\gamma_1$) that is applied twice in the above definition. The default definition for $\gamma_1$ is given below. Later on in the specification, new communication pairs will be introduced by extending the definition of $\gamma_1$.

$$\gamma_1(\mathit{AtomicFun}_1, \mathit{AtomicFun}_2) = \text{false} \quad \textbf{otherwise} \qquad \textbf{[cm-def]}$$

[1]See Section 6.6 for an example of such an extension.

Define a lookup function on ToolBus scripts to retrieve a process definition with a given name.

process-definition(*Pnm*, *Defs*$_1$ process *Pnm* *Formals* is *P* *Defs*$_2$ *TB-Config*) = **[process-definition-1]**
process *Pnm* *Formals* is *P*

In Section 7.8 the need will arize to have a two-way mapping between process expressions and terms. Below we define these mappings for the constructs defined so far. In the remainder of the specification we will systematically extend them for new constructs at the moment that new constructs are introduced in the specification.

| | | |
|---|---|---|
| proc2term(atomic-fun(*Chars*)(*OptTs*)) | = `atom`(string(""" *Chars* """), *OptTs*) | [p2t-std] |
| proc2term($\delta$) | = `atom("delta")` | [p2t-delta-1] |
| proc2term(`delta`) | = `atom("delta")` | [p2t-delta-2] |
| proc2term(`tau`) | = `atom("tau")` | [p2t-tau] |
| proc2term($P_1$ + $P_2$) | = `plus`(proc2term($P_1$), proc2term($P_2$)) | [p2t-plus] |
| proc2term($P_1$ . $P_2$) | = `dot`(proc2term($P_1$), proc2term($P_2$)) | [p2t-dot] |
| proc2term(*Pnm*(*OptTs*)) | = `call`(*Pnm*, *OptTs*) | [p2t-call] |

| | | |
|---|---|---|
| term2proc(`atom`(string(""" *Chars* """), *OptTs*)) | = atomic-fun(*Chars*)(*OptTs*) | [t2p-std] |
| term2proc(`atom("delta")`) | = `delta` | [t2p-delta] |
| term2proc(`atom("tau")`) | = `tau` | [t2p-tau] |
| term2proc(`plus`($T_1$, $T_2$)) | = term2proc($T_1$) + term2proc($T_2$) | [t2p-plus] |
| term2proc(`dot`($T_1$, $T_2$)) | = term2proc($T_1$) . term2proc($T_2$) | [t2p-dot] |
| term2proc(`call`(*Pnm*, *OptTs*)) | = *Pnm*(*OptTs*) | [t2p-call] |

| | | |
|---|---|---|
| atomic-fun2string(atomic-fun(*Chars*)) | = string(""" *Chars* """) | [af2s-1] |
| name2string(name(*Chars*)) | = string(""" *Chars* """) | [n2s-1] |

## 5.2 Prepare atoms, processes and terms

**Module** Prepare
**imports** Tscript$^{(5.1)}$ Environments$^{(4.4)}$
**exports**
  **context-free syntax**

```
    NAME "$" NAME                        → VNAME
    prep-term(TERM, NAME, ENV)           → TERM
    prep-term-list(TERM-LIST, NAME, ENV) → TERM-LIST
    prep-proc(PROC, NAME, ENV)           → PROC
```

**equations**
One global name space is shared between all processes in the ToolBus. To avoid name conflicts, each name is implicitly postfixed with the name of the process in which it occurs. Clashes between names declared in different process definitions are avoided by *resolving names* in process expressions in the following manner:

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

- Replace all variables that have an associated value in a given environment E by their value. This is used for formal/actual parameter binding both for process instantiation (e.g., the use of a process name inside a process expression), and for process creation (by means of create). In both cases, formals are replaced by their actual values throughout the process expression associated with the given process name. Note that *result occurrences* of formal/actual parameters (both marked with ?), are accessed via the environment to ensure that modifications made by a called process are propagated to the environment of the calling process.

For convenience in the specification, also uniformly replace constants in terms by function applications with an empty list of arguments (see, (prep-term-8), below)
*Resolve names in terms.*

$$\text{prep-term}(Bool, Pnm, Env) = Bool \qquad \textbf{[prep-bool]}$$

$$\text{prep-term}(Int, Pnm, Env) = Int \qquad \textbf{[prep-int]}$$

$$\text{prep-term}(String, Pnm, Env) = String \qquad \textbf{[prep-str]}$$

$$\frac{\begin{array}{c} Var = Name \ \$ \ Pnm : \text{declared-type}(Name \ \$ \ Pnm, Env), \\ \text{value}(Var, Env) = T, \ \ T \neq Var \end{array}}{\text{prep-term}(Name, Pnm, Env) = T} \qquad \textbf{[prep-name-1]}$$

$$\frac{\begin{array}{c} Var = Name \ \$ \ Pnm : \text{declared-type}(Name \ \$ \ Pnm, Env), \\ \text{value}(Var, Env) = Var \end{array}}{\text{prep-term}(Name, Pnm, Env) = Var} \qquad \textbf{[prep-name-2]}$$

$$\frac{\begin{array}{c} Var = Name \ \$ \ Pnm : Type, \\ \text{value}(Var, Env) = T, \ \ T \neq Var \end{array}}{\text{prep-term}(Name : Type, Pnm, Env) = T} \qquad \textbf{[prep-name-3]}$$

$$\frac{\begin{array}{c} Var = Name \ \$ \ Pnm : Type, \\ \text{value}(Var, Env) = Var \end{array}}{\text{prep-term}(Name : Type, Pnm, Env) = Var} \qquad \textbf{[prep-name-4]}$$

$$\text{prep-term}(Name \ ? \ , Pnm, Env) = Name \ \$ \ Pnm : \text{declared-type}(Name \ \$ \ Pnm, Env) \ ? \qquad \textbf{[prep-name-5]}$$

$$\text{prep-term}(Name : Type \ ? \ , Pnm, Env) = Name \ \$ \ Pnm : Type \ ? \qquad \textbf{[prep-name-6]}$$

$$\text{prep-term}(Id, Pnm, Env) = Id() \qquad \textbf{[prep-id-1]}$$

$$\text{prep-term}(Id(OptTs), Pnm, Env) = Id(\text{prep-term-list}(OptTs, Pnm, Env)) \qquad \textbf{[prep-id-2]}$$

$$\frac{\text{prep-term}(T, Pnm, Env) = T', \ \ \text{prep-term}([OptTs], Pnm, Env) = [OptTs']}{\text{prep-term}([T, OptTs], Pnm, Env) = [T', OptTs']} \qquad \textbf{[prep-list-1]}$$

$$\text{prep-term}([], Pnm, Env) = [] \qquad \textbf{[prep-list-2]}$$

*Resolve names in term lists.*

$$\frac{\text{prep-term}(T, Pnm, Env) = T', \ \ \text{prep-term-list}(OptTs, Pnm, Env) = OptTs'}{\text{prep-term-list}(T, OptTs, Pnm, Env) = T', OptTs'} \qquad \textbf{[prep-tl-1]}$$

$$\text{prep-term-list}(T, Pnm, Env) = \text{prep-term}(T, Pnm, Env) \qquad \textbf{[prep-tl-2]}$$

$$\text{prep-term-list}(, Pnm, Env) = \qquad \textbf{[prep-tl-3]}$$

*Resolve names in process expressions.*

```
prep-proc(AtomicFun(OptTs), Pnm, Env)  =  AtomicFun(prep-term-list(OptTs, Pnm, Env)) [prep-atom-1]
prep-proc(AtomicFun, Pnm, Env)         =  AtomicFun                                   [prep-atom-2]

prep-proc(P1 + P2, Pnm, Env) =                                                        [prep-plus]
 prep-proc(P1, Pnm, Env) + prep-proc(P2, Pnm, Env)
prep-proc(P1 . P2, Pnm, Env) =                                                        [prep-dot]
 prep-proc(P1, Pnm, Env) . prep-proc(P2, Pnm, Env)

prep-proc(Pnm(OptTs), Pnm', Env)  =  Pnm(prep-term-list(OptTs, Pnm', Env))            [prep-call]
```

## 5.3 The global state of the TOOLBUS

The global state of the TOOLBUS will be represented by the sort BUS-STATE. We will introduce it in two steps:

- Define a generic, extensible, datastructure for representing records (Section 5.3.1).
- Use this notion to define the specific bus state needed (Section 5.3.2).

### 5.3.1 Representing the TOOLBUS state

```
Module StateRepr
imports  Tscript(5.1) Environments(4.4)
exports
  sorts BUS-VAL BUS-ASG FIELD BUS-STATE PROC-ID
  context-free syntax
    proc-id(INT)                                   → PROC-ID
    PROC-ID                                        → TERM

    ID                                             → FIELD
    ID "(" PROC-ID ")"                             → FIELD
    FIELD ":=" BUS-VAL                             → BUS-ASG

    no-bus-val                                     → BUS-VAL
    bus-state({BUS-ASG ";"}*)                      → BUS-STATE
    BUS-STATE "." FIELD                            → BUS-VAL
    BUS-STATE "[" BUS-ASG "]"                      → BUS-STATE
    duplicate(BUS-STATE, PROC-ID, PROC-ID)         → BUS-STATE
    dup1(BUS-STATE, PROC-ID, PROC-ID, BUS-STATE)   → BUS-STATE
    del-proc-id(BUS-STATE, PROC-ID)                → BUS-STATE
  variables
    BS [0-9']*       → BUS-STATE
    BusVal [0-9']*   → BUS-VAL
    BusAsg [0-9']*   → BUS-ASG
    BusAsgs [0-9']*→ {BUS-ASG ";"}*
    Field [0-9']*    → FIELD
    BS [0-9']*       → BUS-STATE
    Pid [0-9']*      → PROC-ID
```

**equations**

Introduce extensible records of (FIELD,BUS-VAL) pairs to represent the state of the ToolBus (sort BUS-STATE). The actual sorts to be used have to be embedded in BUS-VAL. Access to this representation is provided by the operator ".". Values can be modified using the operator "[ ]". This framework can be defined as follows.

*Retrieve the value of a field.*

$$\text{bus-state}(\mathit{BusAsgs}_1;\ \mathit{Field} := \mathit{BusVal};\ \mathit{BusAsgs}_2)\ .\ \mathit{Field}\ =\ \mathit{BusVal} \qquad \text{[retrieve-bus-1]}$$

$$\text{bus-state}(\mathit{BusAsgs})\ .\ \mathit{Field}\ =\ \text{no-bus-val}\quad \textbf{otherwise} \qquad \text{[retrieve-bus-2]}$$

*Assign a new value to a field.*

$$\text{bus-state}(\mathit{BusAsgs}_1;\ \mathit{Field} := \mathit{BusVal};\ \mathit{BusAsgs}_2)[\mathit{Field} := \mathit{BusVal}'] = \qquad \text{[asg-bus-1]}$$
$$\text{bus-state}(\mathit{BusAsgs}_1;\ \mathit{Field} := \mathit{BusVal}';\ \mathit{BusAsgs}_2)$$

$$\text{bus-state}(\mathit{BusAsgs}_1)[\mathit{Field} := \mathit{BusVal}]\ =\ \text{bus-state}(\mathit{BusAsgs}_1;\ \mathit{Field} := \mathit{BusVal})\quad \textbf{otherwise} \qquad \text{[asg-bus-2]}$$

*Duplicate all bus assignments for a given process identifier.*

$$\text{duplicate}(\text{BS},\ \mathit{Pid}_1,\ \mathit{Pid}_2)\ =\ \text{dupl}(\text{BS},\ \mathit{Pid}_1,\ \mathit{Pid}_2,\ \text{bus-state}()) \qquad \text{[duplicate-1]}$$

$$\text{dupl}(\text{bus-state}(\mathit{Id}(\mathit{Pid}_1) := \mathit{BusVal};\ \mathit{BusAsgs}_1),\ \mathit{Pid}_1,\ \mathit{Pid}_2,\ \text{bus-state}(\mathit{BusAsgs}_2)) = \qquad \text{[dupl-1]}$$
$$\text{dupl}(\text{bus-state}(\mathit{BusAsgs}_1),\ \mathit{Pid}_1,\ \mathit{Pid}_2,\ \text{bus-state}(\mathit{BusAsgs}_2;\ \mathit{Id}(\mathit{Pid}_1) := \mathit{BusVal};\ \mathit{Id}(\mathit{Pid}_2) := \mathit{BusVal}))$$

$$\text{dupl}(\text{bus-state}(),\ \mathit{Pid}_1,\ \mathit{Pid}_2,\ \text{BS})\ =\ \text{BS} \qquad \text{[dup2-2]}$$

$$\text{dupl}(\text{bus-state}(\mathit{BusAsg};\ \mathit{BusAsgs}_1),\ \mathit{Pid}_1,\ \mathit{Pid}_2,\ \text{bus-state}(\mathit{BusAsgs}_2)) = \qquad \text{[dupl-3]}$$
$$\text{dupl}(\text{bus-state}(\mathit{BusAsgs}_1),\ \mathit{Pid}_1,\ \mathit{Pid}_2,\ \text{bus-state}(\mathit{BusAsgs}_2;\ \mathit{BusAsg}))\quad \textbf{otherwise}$$

*Remove all bus assignments for a given process identifier.*

$$\text{del-proc-id}(\text{bus-state}(\mathit{BusAsgs}_1;\ \mathit{Id}(\mathit{Pid}) := \mathit{BusVal};\ \mathit{BusAsgs}_2),\ \mathit{Pid}) = \qquad \text{[del-proc-id-1]}$$
$$\text{del-proc-id}(\text{bus-state}(\mathit{BusAsgs}_1;\ \mathit{BusAsgs}_2),\ \mathit{Pid})$$
$$\text{del-proc-id}(\text{BS},\ \mathit{Pid}) = \text{BS}\quad \textbf{otherwise} \qquad \text{[del-proc-id-2]}$$

This framework provides considerable flexibility that will be fully exploited in this specification:

- The constant no-bus-val will be used to catch uninitialized fields, giving an opportunity for properly initializing them.
- The default equations for retrieval and assignment can be overruled by using fields whose value is not explicitly stored in the bus state, but is rather recomputed on each retrieval and stored in other parts of the bus state on each assignment. In this way, the access to specific fields can be completely controlled by adding an equation for these cases.
- The *duplicate* function is used to define inheritance in a modular fashion.

### 5.3.2 Bus state

**Module** BusState
**imports** StateRepr$^{(5.3.1)}$

**exports**

**sorts** CONTEXT

**context-free syntax**

| | |
|---|---|
| PROC-ID | $\rightarrow$ BUS-VAL |
| TERM | $\rightarrow$ BUS-VAL |
| get-proc-name(BUS-STATE, PROC-ID) | $\rightarrow$ TERM |
| get-new-proc-id(BUS-STATE) | $\rightarrow$ PROC-ID |
| context(PROC-ID, ENV, T-SCRIPT, BUS-STATE) | $\rightarrow$ CONTEXT |
| get-proc-id(CONTEXT) | $\rightarrow$ PROC-ID |
| get-env(CONTEXT) | $\rightarrow$ ENV |
| get-script(CONTEXT) | $\rightarrow$ T-SCRIPT |
| get-bus-state(CONTEXT) | $\rightarrow$ BUS-STATE |
| CONTEXT "/" ENV | $\rightarrow$ CONTEXT |
| CONTEXT "/" T-SCRIPT | $\rightarrow$ CONTEXT |
| CONTEXT "/" BUS-STATE | $\rightarrow$ CONTEXT |
| is-enabled(ATOM, CONTEXT) | $\rightarrow$ BOOL |

**variables**

*C* [*0-9′*]*$\rightarrow$ CONTEXT

**equations**

The bus state contains at least the following global fields:

- new-proc-id: a global counter for generating unique process identifiers.
- `name(`*k*`)`: the name of the process definition used to create this process.

See Appendix D for a summary of all the global fields that will be used in this specification. As a convention, define for each field $F$ of sort $S$ that may occur in a BUS-STATE, a function with name get-$F$ with sort $S$ as result sort.

$$\frac{\text{BS . proc-name}(\mathit{Pid}) = \text{proc-name}(\mathit{String})}{\text{get-proc-name}(\text{BS}, \mathit{Pid}) = \text{proc-name}(\mathit{String})} \qquad \textbf{[get-proc-name-1]}$$

$$\text{BS . new-proc-id} = \mathit{Pid} \ \Rightarrow \text{get-new-proc-id}(\text{BS}) \ = \ \mathit{Pid} \qquad \textbf{[get-new-proc-id-1]}$$

$$\begin{aligned}
&\text{get-proc-id}(\text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS})) &&= \mathit{Pid} &&\textbf{[cont-get-proc-id]}\\
&\text{get-env}(\text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS})) &&= \mathit{Env} &&\textbf{[cont-get-env]}\\
&\text{get-script}(\text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS})) &&= \mathit{Script} &&\textbf{[cont-get-script]}\\
&\text{get-bus-state}(\text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS})) &&= \text{BS} &&\textbf{[cont-get-but-state]}
\end{aligned}$$

$$\begin{aligned}
&\text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS}) \ / \ \mathit{Env}' &&= \text{context}(\mathit{Pid}, \mathit{Env}', \mathit{Script}, \text{BS}) &&\textbf{[cont-set-env]}\\
&\text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS}) \ / \ \mathit{Script}' &&= \text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}', \text{BS}) &&\textbf{[cont-set-script]}\\
&\text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS}) \ / \ \text{BS}' &&= \text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \text{BS}') &&\textbf{[cont-set-bus-state]}
\end{aligned}$$

By default, an atom is always enabled. By extending the definition below, conditionals will be defined (see Section 6.6).

$$\text{is-enabled}(\mathit{Atom}, C) = \text{true} \qquad \textbf{otherwise} \qquad \textbf{[is-enabled-def]}$$

## 5.4 Normalizing process expressions—action prefix form

```
Module ActionPrefixForm
imports  Prepare(5.2) BusState(5.3.2) Match(4.5)
exports
  sorts AP-FORM
  context-free syntax
    ATOM                                                    → AP-FORM
    ATOM "." PROC                                           → AP-FORM
    "(" AP-FORM ")"                                         → AP-FORM  {bracket}
    "<" {AP-FORM " +"}* ">"                                 → AP-FORM

    expand(PROC, CONTEXT)                                   → AP-FORM
    sum(AP-FORM, AP-FORM)                                   → AP-FORM
    make-sum(AP-FORM)                                       → AP-FORM
    dot(AP-FORM, PROC)                                      → AP-FORM

    create-env({GEN-VAR ","}*, NAME, {TERM ","}*, ENV)      → ENV
    bind-list({GEN-VAR ","}*, NAME, {TERM ","}*, ENV, ENV)  → ENV
  variables
    AP [0-9']*    → AP-FORM
    APs [0-9']*   → {AP-FORM " +"} +
    OAPs [0-9']*→ {AP-FORM " +"}*
equations
```

Transform process expressions into the *action-prefix form* $< AP_1 + \ldots + AP_n >$ that is well-suited for interpretation. This will permit the use of list matching instead of associative matching over the binary +-operator (associative matching is not available in ASF+SDF). In the specification, we represent action prefix form by the sort AP-FORM. The function expand defines the actual conversion from PROC to AP-FORM (using Process Algebra axioms A2, A4, and A5).

```
expand(Atom, C)             = Atom                                      [exp-atom]
expand(Atom . P, C)         = Atom . P                                  [exp-dot-1]
expand((P1 . P2) . P3, C)   = expand(P1 . P2 . P3, C)                   [exp-dot-2]
expand(P1 . P2, C)          = dot(expand(P1, C), P2)        otherwise   [exp-dot-3]
expand(P1 + P2, C)          = sum(expand(P1, C), expand(P2, C))         [exp-plus]
```

Create a "flat" sum by eliminating δ's and by flattening nested +-operators. (using Process Algebra axioms A1, A6, and A7).

```
sum(δ, AP)                  = AP                                        [sum-1]
sum(AP, δ)                  = AP                                        [sum-2]
sum(< OAPs1 > , < OAPs2 >)  = < OAPs1 + OAPs2 >                         [sum-3]
sum(Atom, < OAPs >)         = < Atom + OAPs >                           [sum-4]
sum(Atom . P, < OAPs >)     = < Atom . P + OAPs >                       [sum-5]
sum(< OAPs > , Atom)        = < OAPs + Atom >                           [sum-6]
sum(< OAPs > , Atom . P)    = < OAPs + Atom . P >                       [sum-7]
sum(AP1, AP2)               = < AP1 + AP2 >          otherwise          [sum-8]
```

Conceptually, we add a final equation to the definition of sum to ensure the random expansion of sums. It may be (but need not be) applied at any moment during the calculation of sum and permutes its arguments.[2]

[2]In order to execute this specification, this equation has to be left out since it causes non-termination.

$\text{sum}(\text{AP}_1, \text{AP}_2) = \text{sum}(\text{AP}_2, \text{AP}_1)$

When necessary, the function make-sum turns an action prefix form into a sum of alternatives. It is used to ensure that the top-level process expression of each process is always a sum.

$\text{make-sum}(< \mathit{OAPs} >) = < \mathit{OAPs} >$ [make-sum-1]

$\text{make-sum}(\text{AP}) = < \text{AP} >$ **otherwise** [make-sum-2]

Construct the "dot" of two process expressions $P_1$ and $P_2$, taking into account the inward propagation of $P_2$ into sums occurring in $P_1$.

$\text{dot}(\delta, P) = \delta$ [dot-1]

$\text{dot}(\mathit{Atom}, P) = \mathit{Atom} \,.\, P$ [dot-2]

$\text{dot}(\mathit{Atom} \,.\, P_1, P_2) = \mathit{Atom} \,.\, P_1 \,.\, P_2$ [dot-3]

$\text{dot}(< \delta + \mathit{OAPs} > , P_2) = \text{dot}(< \mathit{OAPs} > , P_2)$ [dot-4]

$\text{dot}(< \mathit{Atom} + \mathit{OAPs} > , P_2) = \text{sum}(\mathit{Atom} \,.\, P_2, \text{dot}(< \mathit{OAPs} > , P_2))$ [dot-5]

$\text{dot}(< \mathit{Atom} \,.\, P_1 + \mathit{OAPs} > , P_2) = \text{sum}(\mathit{Atom} \,.\, P_1 \,.\, P_2, \text{dot}(< \mathit{OAPs} > , P_2))$ [dot-6]

$\text{dot}(< \; > , P) = \delta$ [dot-7]

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

process *Pnm Formals* is *P*

Whenever *Pnm* appears as the leftmost operand of a process expression it is changed (by the expand) into the disambiguated and expanded version of the process expression *P*. In this way, process names are expanded into their definition only when needed and endless recursion due to the leftmost innermost reduction strategy used by the ASF+SDF system can be avoided.

$$\frac{\begin{array}{c}\text{process-definition}(\mathit{Pnm}, \text{get-script}(C)) = \text{process } \mathit{Pnm}\ (\mathit{GenVars}) \text{ is } P,\\ \mathit{Env} = \text{create-env}(\mathit{GenVars}, \mathit{Pnm}, \mathit{OptTs}, \text{get-env}(C))\end{array}}{\text{expand}(\mathit{Pnm}(\mathit{OptTs}), C) = \text{expand}(\text{prep-proc}(P, \mathit{Pnm}, \mathit{Env}), C)}$$ [exp-call]

Define the function create-env that is used for the construction of a new environment during process creation.

$\text{create-env}(\mathit{GenVars}, \mathit{Pnm}, \mathit{OptTs}, \mathit{Env}) = \text{bind-list}(\mathit{GenVars}, \mathit{Pnm}, \mathit{OptTs}, [], \mathit{Env})$ [create-env-1]

It uses the auxiliary function bind-list. Note the two forms of binding for "by-value" parameters (represented by the use of an ordinary variable as formal) and "by-reference" parameters (represented by the use of a result variable as formal). See Section 4.4 for the treatment of both forms of binding.

$$\frac{\begin{array}{c}\text{require-type}(\mathit{Type}, T) = \text{true},\\ T' = \text{substitute}(T, \mathit{Env}),\\ [\mathit{Entries}'] = [\mathit{Name}\ \$\ \mathit{Pnm} : \mathit{Type} \mapsto T', \mathit{Entries}]\end{array}}{\begin{array}{l}\text{bind-list}(\mathit{Name} : \mathit{Type}, \mathit{GenVars}, \mathit{Pnm}, T, \mathit{OptTs}, [\mathit{Entries}], \mathit{Env}) =\\ \text{bind-list}(\mathit{GenVars}, \mathit{Pnm}, \mathit{OptTs}, [\mathit{Entries}'], \mathit{Env})\end{array}}$$ [bind-list-1]

$$\frac{[\mathit{Entries}'] = [\mathit{Name}\ \$\ \mathit{Pnm} : \mathit{Type} \mapsto \mathit{Vname} : \mathit{Type}\ ?, \mathit{Entries}]}{\begin{array}{l}\text{bind-list}(\mathit{Name} : \mathit{Type}\ ?, \mathit{GenVars}\ , \mathit{Pnm}, \mathit{Vname} : \mathit{Type}\ ?, \mathit{OptTs}\ , [\mathit{Entries}], \mathit{Env}) =\\ \text{bind-list}(\mathit{GenVars}, \mathit{Pnm}, \mathit{OptTs}, [\mathit{Entries}'], \mathit{Env})\end{array}}$$ [bind-list-2]

Observe, in the following equation the *two* occurrences of the empty term list (denoted by the empty string!).

$\text{bind-list}(, \mathit{Pnm}, , [\mathit{Entries}], \mathit{Env}) = [\mathit{Entries}]$ [bind-list-3]

$\text{bind-list}(\texttt{SVars}, \mathit{Pnm}, \mathit{OptTs}, [\mathit{Entries}], \mathit{Env}) = []$ **otherwise** [bind-list-4]

Observe that applying bind-list to lists of unequal length yields an empty environment.

## 5.5 Representation and interpretation of a ToolBus

```
Module ToolBus
imports  ActionPrefixForm(5.4) Match(4.5)
exports
  sorts TOOLS BUS E-PROCESSES PROC-REPR PROCESSES
        NEXT-INFO L-AP-FORM
  context-free syntax
    "λ" "_" ENV "(" AP-FORM ")"                   → L-AP-FORM
    "ρ" "_" PROC-ID "(" L-AP-FORM ")"             → PROC-REPR
    mk-proc-repr(PROC-ID, ENV, AP-FORM)           → PROC-REPR
    get-pid(PROC-REPR)                            → PROC-ID
    "□"                                           → ATOM
    "□"                                           → PROC-REPR
    "{" {PROC-REPR " ||"}* "}"                    → PROCESSES

    "E" "_" T-SCRIPT "(" PROCESSES ")"            → E-PROCESSES
    "λ" "_" BUS-STATE "(" E-PROCESSES ")"         → BUS

    "ν" "_" NEXT-INFO "(" AP-FORM ")"             → AP-FORM
    info(ATOM, CONTEXT, PROC-ID)                  → NEXT-INFO
    simple(ATOM, CONTEXT, PROC-ID)                → NEXT-INFO

    add-proc(PROC-APPL, PROC-ID, BUS)             → BUS
    add-procs({PROC-APPL ","} +, PROC-ID, BUS)    → BUS
    tools({PROC ","}*)                            → TOOLS
    add-tool(PROC, BUS)                           → BUS
    add-tools(TOOLS, BUS)                         → BUS
    create-bus(TOOLS, T-SCRIPT)                   → BUS

    simple-atomic-step(ATOM, CONTEXT)             → CONTEXT
    atomic-steps(BUS, BOOL)                       → BUS
    all-atomic-steps(BUS)                         → BUS
  variables
    Procs [0-9']*→ {PROC ","}*
    Tools [0-9']*→ TOOLS
    Chars [0-9']*→ CHAR*
    B [0-9']*     → BUS
    W             → BOOL
    PR [0-9']*    → PROC-REPR
    PRs [0-9']*   → {PROC-REPR " ||"}*
equations
```

**Structure of a** BUS. Recall from Section 5 that a ToolBus is completely characterized by the following process expression

$$\lambda_{BS}(E_{Script}(\{\rho_1(\lambda_{Env_1}(<AP_{11} + \ldots + AP_{1n_1}>))||\ldots||\rho_m(\lambda_{Env_m}(<AP_{m1} + \ldots + AP_{mn_m}>))\})).$$

The operands of the merge operator $||$ all have the form $\rho_k(\lambda_{Env}(<AP_1 + \ldots + AP_n>))$ and are the actual "processes" that are executing in parallel. They will be represented by the sort PROC-REPR (process representation) defined by the following functions. We will use the constant "□" (of sort ATOM)) as a "cursor" in an action prefix form.

$$\text{mk-proc-repr}(Pid, Env, \text{AP}) = \rho_{Pid}\,(\lambda_{Env}\,(\text{AP})) \qquad \textbf{[mk-proc-repr-1]}$$

$$\text{get-pid}(\rho_{Pid}\,(\lambda_{Env}\,(\text{AP}))) = Pid \qquad \textbf{[get-pid-1]}$$

The merge operator itself will be represented as an $n$-ary operator of the form $\{PR_1||...||PR_m\}$ (sort PROCESSES) and we will use the constant "$\Box$" (of sort PROC-REPR) as a "cursor" in such a list of processes.

A complete BUS is defined by enclosing a list of merged processes by a process creation operator containing the complete **T** script ($E_{Script}$) and a state operator containing the bus state ($\lambda_{BS}$).

The operator $\nu$ will be used to determine the "next" step to be taken in a process expression. It is parameterized with a function info with the following default meaning.

$$\text{info}(\mathit{Atom}, C, \mathit{Pid}) = \text{simple}(\mathit{Atom}, C, \mathit{Pid}) \quad \textbf{otherwise} \qquad \textbf{[info-def]}$$

By extending the default definition of the function $\nu$ below one can define, for instance, monitoring (see Section 7.7.

$$\nu_{\text{simple}(\mathit{Atom},\ C,\ \mathit{Pid}_2)}(\Box\ .\ P) = \text{make-sum}(\text{expand}(P, C)) \quad \textbf{otherwise} \qquad \textbf{[nu-def]}$$

**Construction of a** BUS**.** A list of process names—as appearing in each ToolBus configuration (TB-CONFIG), see Section 5.1—can be transformed into a complete bus as follows. First, add-proc extends the bus with a new process given a process name with optional actual parameters. This function will be used in two manners: for the construction of the initial bus and, during interpretation, for the construction of a new process resulting from a create atom (see Section 6.7).

$$\frac{\begin{array}{c}\text{process-definition}(\mathit{Pnm}, \mathit{Script}) = \text{process } \mathit{Pnm}\ (\mathit{GenVars}) \text{ is } P,\\ \mathit{Pid}_2 = \text{get-new-proc-id}(\text{BS}),\\ \mathit{Pid}_2 = \text{proc-id}(\mathit{Int}),\\ \mathit{Env} = \text{create-env}(\mathit{GenVars}, \mathit{Pnm}, \mathit{OptTs}, []),\\ \mathit{Pnm} = \text{name}(\mathit{Chars}),\ \mathit{String} = \text{string}(\text{"\""}\ \mathit{Chars}\ \text{"\""}),\\ \text{AP} = \text{expand}(\text{prep-proc}(P, \mathit{Pnm}, \mathit{Env})\ .\ \delta, \text{context}(\mathit{Pid}_2, \mathit{Env}, \mathit{Script}, \text{BS})),\\ \text{PR} = \text{mk-proc-repr}(\mathit{Pid}_2, \mathit{Env}, \text{make-sum}(\text{AP})),\\ \text{BS}' = \text{duplicate}(\text{BS}, \mathit{Pid}_1, \mathit{Pid}_2),\\ \text{BS}'' = \text{BS}[\texttt{new-proc-id} := \text{proc-id}(\mathit{Int} + 1)][\texttt{proc-name}(\mathit{Pid}_2) := \texttt{proc-name}(\mathit{String})]\end{array}}{\text{add-proc}(\mathit{Pnm}(\mathit{OptTs}), \mathit{Pid}_1, \lambda_{\text{BS}}(\text{E}_{\mathit{Script}}(\{\mathit{PRs}\}))) = \lambda_{\text{BS}''}(\text{E}_{\mathit{Script}}(\{\mathit{PRs} \mid\mid \text{PR}\}))} \qquad \textbf{[add-proc-1]}$$

A complete bus can then be constructed by repeated application of add-proc:

$$\frac{\text{add-proc}(\mathit{Pnm}(\mathit{OptTs}), \mathit{Pid}, B) = B'}{\text{add-procs}(\mathit{Pnm}(\mathit{OptTs}), \mathit{ProcAppls}, \mathit{Pid}, B) = \text{add-procs}(\mathit{ProcAppls}, \mathit{Pid}, B')} \qquad \textbf{[add-procs-1]}$$

$$\text{add-procs}(\mathit{Pnm}(\mathit{OptTs}), \mathit{Pid}, B) = \text{add-proc}(\mathit{Pnm}(\mathit{OptTs}), \mathit{Pid}, B) \qquad \textbf{[add-procs-2]}$$

Finally, define how a list of processes (representing tool behaviour) can be added to the bus state.

$$\frac{\begin{array}{c}\mathit{Pid} = \text{get-new-proc-id}(\text{BS}),\\ \mathit{Pid} = \text{proc-id}(\mathit{Int}),\\ \text{AP} = \text{expand}(\text{prep-proc}(P, \texttt{TOOL}, [])\ .\ \delta, \text{context}(\mathit{Pid}, [], \mathit{Script}, \text{BS})),\\ \text{PR} = \text{mk-proc-repr}(\mathit{Pid}, [], \text{make-sum}(\text{AP})),\\ \text{BS}' = \text{BS}[\texttt{new-proc-id} := \text{proc-id}(\mathit{Int} + 1)][\texttt{proc-name}(\mathit{Pid}) := \texttt{proc-name}(\texttt{"TOOL"})]\end{array}}{\text{add-tool}(P, \lambda_{\text{BS}}(\text{E}_{\mathit{Script}}(\{\mathit{PRs}\}))) = \lambda_{\text{BS}'}(\text{E}_{\mathit{Script}}(\{\mathit{PRs} \mid\mid \text{PR}\}))} \qquad \textbf{[add-tool-1]}$$

$$\frac{\text{add-tool}(P, B) = B'}{\text{add-tools}(\text{tools}(P, \mathit{Procs}), B) = \text{add-tools}(\text{tools}(\mathit{Procs}), B')} \qquad \textbf{[add-tools-1]}$$

$$\text{add-tools}(\text{tools}(), B) = B \qquad \textbf{[add-tools-3]}$$

Define the creation of a complete bus state.

$$\frac{\begin{array}{c} \mathit{Script} = \mathit{Defs}\ \mathsf{toolbus}(\mathit{ProcAppls}), \\ B = \lambda_{\mathsf{bus\text{-}state}(\mathtt{new\text{-}proc\text{-}id} := \mathsf{proc\text{-}id}(0))}\ (\mathsf{E}_{\mathit{Script}}\ (\{\square\})), \\ B' = \mathsf{add\text{-}procs}(\mathit{ProcAppls},\ \mathsf{proc\text{-}id}(-\ 1),\ B), \\ B'' = \mathsf{add\text{-}tools}(\mathsf{tools}(\mathit{Procs}),\ B') \end{array}}{\mathsf{create\text{-}bus}(\mathsf{tools}(\mathit{Procs}),\ \mathit{Script}) = B''} \qquad \textbf{[create-bus-1]}$$

**Interpretation of a** BUS. Define the basic interpretation function atomic-steps which will be used later on to define specific features. It loops through all processes in a bus and performs all possible atomic steps in each process by repeatedly applying simple-atomic-step as long as this is possible. The function simple-atomic-step will be defined separately for each specific feature (see Sections 6 and 7).
Perform a simple atomic step in the current process.

$$\frac{\begin{array}{c} \mathsf{PR} = \rho_{\mathit{Pid}}\ (\lambda_{\mathit{Env}}\ (<\ \mathit{OAPs}' + \mathit{Atom}\ .\ P + \mathit{OAPs}''\ >)), \\ C = \mathsf{context}(\mathit{Pid},\ \mathit{Env},\ \mathit{Script},\ \mathsf{BS}), \\ \mathsf{is\text{-}enabled}(\mathit{Atom},\ C) = \mathsf{true}, \\ [\mathit{OptTs}] = \mathsf{args}(\mathit{Atom}), \\ \mathsf{simple\text{-}atomic\text{-}step}(\mathsf{fun}(\mathit{Atom})(\mathit{OptTs}),\ C) = C', \\ C' = \mathsf{context}(\mathit{Pid},\ \mathit{Env}',\ \mathit{Script}',\ \mathsf{BS}'), \\ \mathsf{PR}' = \rho_{\mathit{Pid}}\ (\lambda_{\mathit{Env}'}\ (\nu_{\mathsf{info}(\mathit{Atom},\ C',\ \mathsf{proc\text{-}id}(-\ 1))}\ (\square\ .\ P))) \end{array}}{\begin{array}{l} \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1\ \|\ \square\ \|\ \mathsf{PR}\ \|\ \mathit{PRs}_2\})),\ W) = \\ \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}'}\ (\mathsf{E}_{\mathit{Script}'}\ (\{\mathit{PRs}_1\ \|\ \mathsf{PR}'\ \|\ \square\ \|\ \mathit{PRs}_2\})),\ \mathsf{true}) \end{array}} \qquad \textbf{[as-1-trans]}$$

Remove a process that has no choices left.

$$\frac{\mathsf{PR} = \rho_{\mathit{Pid}}\ (\lambda_{\mathit{Env}}\ (<\ \delta\ >)),\quad \mathsf{BS}' = \mathsf{del\text{-}proc\text{-}id}(\mathsf{BS},\ \mathit{Pid})}{\begin{array}{l} \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1\ \|\ \square\ \|\ \mathsf{PR}\ \|\ \mathit{PRs}_2\})),\ W) = \\ \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}'}\ (\mathsf{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1\ \|\ \square\ \|\ \mathit{PRs}_2\})),\ W) \end{array}} \qquad \textbf{[as-2]}$$

When all processes have been visited, continue, provided that some atomic step was executed in the last iteration. The list of processes is randomly permuted[3] before proceeding to the next iteration of atomic-steps.

$$\frac{\{\mathit{PRs}\} = \{\mathit{PRs}_1\ \|\ \mathit{PRs}_2\}}{\begin{array}{l} \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\mathit{PRs}\ \|\ \square\})),\ \mathsf{true}) = \\ \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\square\ \|\ \mathit{PRs}_2\ \|\ \mathit{PRs}_1\})),\ \mathsf{false}) \end{array}} \qquad \textbf{[as-3]}$$

The function atomic-steps terminates when all processes have been visited and no atomic step has been executed,

$$\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\mathit{PRs}\ \|\ \square\})),\ \mathsf{false})\ =\ \lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\square\ \|\ \mathit{PRs}\})) \qquad \textbf{[as-4]}$$

Continue with the next process, if no atomic step can be done in the current one.

$$\begin{array}{l} \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1\ \|\ \square\ \|\ \mathsf{PR}\ \|\ \mathit{PRs}_2\})),\ W) = \\ \mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1\ \|\ \mathsf{PR}\ \|\ \square\ \|\ \mathit{PRs}_2\})),\ W)\quad \textbf{otherwise} \end{array} \qquad \textbf{[as-5]}$$

[3]This random permutation is important at the level of the *formal specification* given. When *executing* this specification, probably an arbitrary, fixed, choice will be made. We insist, however, that an actual implementation makes a random choice here.

Perform all possible atomic steps.

all-atomic-steps($\lambda_{\mathrm{BS}}$ ($\mathrm{E}_{Script}$ ({□ || *PRs*}))) = **[all-at-steps-1]**

atomic-steps($\lambda_{\mathrm{BS}}$ ($\mathrm{E}_{Script}$ ({□ || *PRs*})), false)

In the remainder of this specification we will give equations for the function simple-atomic-step for the various atoms that can appear in **T** scripts. Here, we define only the case for a single $\tau$ step.

simple-atomic-step(tau(), *C*) = *C* **[tau]**

## 5.6 Interpreters for T scripts

In this section we present two typical interpreters that can be defined in our framework. The first one describes an untimed system, while the second one includes a notion of discrete time. Although the definition of specific features is postponed until Section 6 and Section 7, we give the following definitions here to illustrate the use of our framework.

### 5.6.1 A ToolBus interpreter

Module Interpreter
**imports** ToolBus$^{(5.5)}$ Iteration$^{(6.1)}$ FreeMerge$^{(6.2)}$ Messages$^{(6.4)}$ Notes$^{(6.8)}$ Expressions$^{(6.5)}$ Conditionals$^{(6.6)}$ Create$^{(6.7)}$ Reconfigure$^{(7.8)}$ Connect-Disconnect$^{(7.3)}$ Execute-Terminate$^{(7.4)}$ Eval-Do$^{(7.5)}$ Monitoring$^{(7.7)}$ ToolControlProcess$^{(7.2.2)}$
**exports**
**context-free syntax**
interpret(TOOLS, T-SCRIPT) $\rightarrow$ BUS
**equations**
The interpretation function interpret takes a description of the behaviour of tools executing in the outside world and a ToolBus configuration as input and produces the resulting BUS as result. To this end, the external tool processes as well as the ToolBus configuration are first converted into a new bus, which is then interpreted using all-steps. The function add-TCP-defs will be defined in Section 7.2.2 and adds a number of standard process definitions to the given **T** script.

interpret(*Tools*, *Script*) = **[interpret-1]**

all-atomic-steps(create-bus(*Tools*, add-TCP-defs(*Script*)))

Variations on this interpreter can made by including or excluding certain features from the list of imports of this module.

### 5.6.2 A ToolBus interpreter with time

The second interpreter, extends the previous one by also including a notion of "time" as defined in Section 6.9 and the notions of "delay" and "timeout" as defined in Section 6.10. In the previous interpreter, external actions completely dictate the behaviour of the system. In the following system, atomic actions may be executed in each time slice resulting in a somewhat different structure of the interpretation rules.

Module InterpreterWithTime
**imports** ToolBus$^{(5.5)}$ Iteration$^{(6.1)}$ FreeMerge$^{(6.2)}$ Messages$^{(6.4)}$ Notes$^{(6.8)}$ Expressions$^{(6.5)}$ Conditionals$^{(6.6)}$ Create$^{(6.7)}$ Reconfigure$^{(7.8)}$ Connect-Disconnect$^{(7.3)}$ Execute-Terminate$^{(7.4)}$ Eval-Do$^{(7.5)}$ Monitoring$^{(7.7)}$ ToolControlProcess$^{(7.2.2)}$ Delay-Timeout$^{(6.10)}$

**exports**
**context-free syntax**
interpret-dt(TOOLS, T-SCRIPT, INT) → BUS
**variables**
*Time* → INT
*MaxTime* → INT
**equations**
At time 1, the discrete time bus is created. The bus variables time and max-time are set to, respectively, 1 and the maximal time slice. (Both variables are introduced in Section 6.9.)

$$\frac{\text{create-bus}(\mathit{Tools}, \text{add-TCP-defs}(\mathit{Script})) = \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}\}))}{\begin{array}{l}\text{interpret-dt}(\mathit{Tools}, \mathit{Script}, \mathit{MaxTime}) = \\ \text{all-atomic-steps}(\lambda_{\text{BS}[\texttt{time} := 1][\texttt{max-time} := \mathit{MaxTime}]}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}\})))\end{array}} \quad \textbf{[interpret-dt-1]}$$

We add a new equation for atomic-steps that may increment the state variable `time` until the maximal time slice. Note that this new equation *may* always be applied until the maximal time slice. However, we define not when it *should* be applied. This is as close as we can get to modeling a system in which the execution time of atomic steps may vary and the time slice transitions are determined by an external clock. An actual implementation will only apply this rule on predetermined moments, e.g., every second.

$$\frac{\text{get-time}(\text{BS}) = \mathit{Time},\ \mathit{Time} < \text{get-max-time}(\text{BS})}{\begin{array}{l}\text{atomic-steps}(\lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}\})), \mathit{W}) = \\ \text{atomic-steps}(\lambda_{\text{BS}[\texttt{time} := \mathit{Time}\ +\ 1]}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}\})), \mathit{W})\end{array}} \quad \textbf{[time-trans]}$$

# Chapter 6

# Features of ToolBus processes

Given the interpretation framework defined in the previous sections, we are now in the position to define features that can be fitted in the framework. When defining a new feature, the following extensions can be made. Recall from Section 5.5 that the function atomic-steps is the fundamental interpretation function defined on a bus.

New *atoms* can be added by extending the sort ATOMIC-FUN (Section 5.1) or by introducing atoms with a completely new syntactic structure. In the latter case, the definitions of the functions fun, args (Section 5.1) and prep-proc (Section 5.2) have to be extended as well. The interpretation of atoms can be defined by extending the definition of the function simple-atomic-step (Section 5.5).

New *composite process expressions* can be added by extending the sort PROC (Section 5.1). The preparation of new process expressions has to be defined by extending the definition of prep-proc (Section 5.2). Conversion to action prefix form has to be defined by extending the definition of expand (Section 5.4).

The function simple-atomic-step applies to "simple" atoms, i.e., atoms whose interpretation only relies on information that is directly available to the process in which the atom appears.

When the interpretation of an atom involves information from other processes, extensions of atomic-steps is the definition method of choice.

The features that will be introduced are:

- *Iteration* (Section 6.1).
- *Free merge* (Section 6.2).
- *Let* (Section 6.3).
- *Messages* (Section 6.4).
- *Expressions* (Section 6.5).
- *Conditionals* (Section 6.6).
- *Dynamic process creation* (Section 6.7).
- *Notification of processes* (Section 6.8).
- *Discrete time* (Section 6.9).
- *Delay and timeout* (Section 6.10).

## 6.1 Iteration

Repeated execution of process expressions is achieved by the binary Kleene star $P_1 * P_2$: it executes zero or more *repetitions* of process $P_1$ followed by process $P_2$.

**Module** Iteration
**imports** ToolBus$^{(5.5)}$
**exports**
**context-free syntax**
PROC "$*$" PROC $\rightarrow$ PROC {**left**}
**priorities**
PROC "$*$"PROC $\rightarrow$ PROC $>$ PROC "."PROC $\rightarrow$ PROC

**equations**
Extend the preparation of processes (Section 5.2) for the iteration operator.

$$\text{prep-proc}(P_1 * P_2,\ Pnm,\ Env) = \text{prep-proc}(P_1,\ Pnm,\ Env) * \text{prep-proc}(P_2,\ Pnm,\ Env) \quad \text{[prep-iter]}$$

Extend the expansion of process expressions (Section 5.4) for the iteration operator. Expansion of the process expression $P_1$ $*$ $P_2$ yields $P_1'$ ; $P_1$ $*$ $P_2$ + $P_2'$, where $P_1'$ and $P_2'$ are the expanded version of $P_1$, respectively, $P_2$.

$$\text{expand}(P_1 * P_2,\ C) = \text{sum}(\text{expand}(P_1\ .\ P_1 * P_2,\ C),\ \text{expand}(P_2,\ C)) \quad \text{[exp-iter]}$$

Extend the two-way mapping between process expressions and terms for the iteration operator.

$$\text{proc2term}(P_1 * P_2) = \texttt{star}(\text{proc2term}(P_1),\ \text{proc2term}(P_2)) \quad \text{[p2t-iter]}$$
$$\text{term2proc}(\texttt{star}(T_1,\ T_2)) = \text{term2proc}(T_1) * \text{term2proc}(T_2) \quad \text{[t2p-iter]}$$

## 6.2 Free merge

The parallel composition of two process expressions $P_1$ and $P_2$ inside one ToolBus process is achieved by the free merge operator $P_1$ || $P_2$. Note that no communication is possible between $P_1$ and $P_2$ since this is only permitted between process expressions appearing in *different* ToolBus processes.

**Module** FreeMerge
**imports** ToolBus$^{(5.5)}$
**exports**
**context-free syntax**
PROC " ||" PROC $\rightarrow$ PROC {**right**}
PROC "$\lfloor\!\lfloor$ " PROC $\rightarrow$ PROC {**right**}
lmerge(AP-FORM, PROC) $\rightarrow$ AP-FORM
**priorities**
PROC " ||"PROC $\rightarrow$ PROC $<$ PROC " +"PROC $\rightarrow$ PROC, PROC "$\lfloor\!\lfloor$ "PROC $\rightarrow$ PROC $<$ PROC " +"PROC $\rightarrow$ PROC
**equations**
Extend the preparation of processes (Section 5.2) for the free merge operator.

$$\text{prep-proc}(P_1 \parallel P_2,\ Pnm,\ Env) = \text{prep-proc}(P_1,\ Pnm,\ Env) \parallel \text{prep-proc}(P_2,\ Pnm,\ Env) \quad \text{[prep-mrg]}$$

Extend the expansion of process expressions (Section 5.4) for the free merge operator.

$$\text{expand}(P_1 \parallel P_2,\ C) = \text{sum}(\text{expand}(P_1 \lfloor\!\lfloor\ P_2,\ C),\ \text{expand}(P_2 \lfloor\!\lfloor\ P_1,\ C)) \quad \text{[exp-mrg-1]}$$
$$\text{expand}((P_1 \parallel P_2) \parallel P_3,\ C) = \text{expand}(P_1 \parallel P_2 \parallel P_3,\ C) \quad \text{[exp-mrg-2]}$$
$$\text{expand}((P_1 \lfloor\!\lfloor\ P_2) \lfloor\!\lfloor\ P_3,\ C) = \text{expand}(P_1 \lfloor\!\lfloor\ P_2 \parallel P_3,\ C) \quad \text{[exp-mrg-3]}$$
$$\text{expand}(P_1 \lfloor\!\lfloor\ P_2,\ C) = \text{lmerge}(\text{expand}(P_1,\ C),\ P_2) \quad \text{[exp-mrg-4]}$$

$$\text{lmerge}(\delta, P) = \delta \qquad \textbf{[lmerge-1]}$$
$$\text{lmerge}(Atom, P) = Atom\ .\ P \qquad \textbf{[lmerge-2]}$$
$$\text{lmerge}(Atom\ .\ P_1, P_2) = Atom\ .\ P_1 \parallel P_2 \qquad \textbf{[lmerge-3]}$$
$$\text{lmerge}(<\text{AP} + OAPs> , P) = \text{sum}(\text{lmerge}(\text{AP}, P), \text{lmerge}(< OAPs > , P)) \qquad \textbf{[lmerge-4]}$$
$$\text{lmerge}(< \ > , P) = \delta \qquad \textbf{[lmerge-5]}$$

Extend the two-way mapping between process expressions and terms for the merge operator.

$$\text{proc2term}(P_1 \parallel P_2) = \texttt{merge}(\text{proc2term}(P_1), \text{proc2term}(P_2)) \qquad \textbf{[p2t-mrg]}$$
$$\text{term2proc}(\texttt{merge}(T_1, T_2)) = \text{term2proc}(T_1) \parallel \text{term2proc}(T_2) \qquad \textbf{[t2p-mrg]}$$

## 6.3 Let

New variables (and their required type) can be introduced in a process expression $P$ by `let` $Var_1$ `:` $Type_1$`, ... in` $P$ `endlet`.

**Module** Let
**imports** ToolBus$^{(5.5)}$
**exports**
  **sorts** VAR-LIST
  **lexical syntax**
    delete-vars → ATOMIC-FUN
  **context-free syntax**

| | |
|---|---|
| {VAR ","}* | → VAR-LIST |
| let {VAR ","}* in PROC endlet | → PROC |
| extend(VAR-LIST, ENV) | → ENV |
| prep-vars(VAR-LIST, NAME, ENV) | → VAR-LIST |
| vars2term(VAR-LIST) | → TERM |
| term2vars(TERM) | → VAR-LIST |

**equations**
Extend the preparation of processes (Section 5.2) for the let operator.

$$\frac{Var' = \text{prep-term}(Var, Pnm, Env),\quad Vars' = \text{prep-vars}(Vars, Pnm, Env)}{\text{prep-vars}(Var, Vars, Pnm, Env) = Var', Vars'} \qquad \textbf{[prep-vars-1]}$$

$$\text{prep-vars}(, Pnm, Env) = \qquad \textbf{[prep-vars-2]}$$

$$\frac{\text{prep-vars}(Vars, Pnm, Env) = Vars'}{\begin{array}{l}\text{prep-proc}(\text{let } Vars \text{ in } P \text{ endlet}, Pnm, Env) = \\ \text{let } Vars' \text{ in prep-proc}(P, Pnm, \text{extend}(Vars', Env)) \text{ endlet}\end{array}} \qquad \textbf{[prep-let]}$$

$$\frac{\text{has-no-vars}(Type) = \text{true},\quad \text{extend}(Vars, [Entries]) = [Entries']}{\text{extend}(Vname : Type, Vars, [Entries]) = [Vname : Type \mapsto Vname : Type, Entries']} \qquad \textbf{[extend-1]}$$

$$\text{extend}(, Env) = Env \qquad \textbf{[extend-2]}$$

Extend the expansion of process expressions (Section 5.4) for the let operator.

$$\text{expand}(\text{let } \mathit{Vars} \text{ in } P \text{ endlet},\ C) \ = \ \text{expand}(P \,.\, \texttt{delete-vars}(\text{vars2term}(\mathit{Vars})),\ C \,/\, \text{extend}(\mathit{Vars},\ \text{get-env}(C))) \quad \textbf{[exp-let]}$$

Delete the variables previously introduced by the let operator.

$$\text{simple-atomic-step}(\texttt{delete-vars}(T),\ C) = C \,/\, \text{delete}(T,\ \text{get-env}(C)) \qquad \textbf{[endlet-1]}$$

Extend the two-way mapping between process expressions and terms for the let operator.

$$\frac{\text{vars2term}(\mathit{Vars}) = [\mathit{OptTs}]}{\text{vars2term}(\mathit{Var},\ \mathit{Vars}) = [\mathit{Var},\ \mathit{OptTs}]} \qquad \textbf{[vars2term-1]}$$

$$\text{vars2term}() \ = \ [] \qquad \textbf{[vars2term-2]}$$

$$\frac{\text{term2vars}([\mathit{OptTs}]) = \mathit{Vars}}{\text{term2vars}([\mathit{Var},\ \mathit{OptTs}]) = \mathit{Var},\ \mathit{Vars}} \qquad \textbf{[term2vars-1]}$$

$$\text{term2vars}([]) \ = \qquad \textbf{[term2vars-2]}$$

$$\text{proc2term}(\text{let } \mathit{Vars} \text{ in } P \text{ endlet}) \ = \ \texttt{letin}(\text{vars2term}(\mathit{Vars}),\ \text{proc2term}(P)) \qquad \textbf{[p2t-let]}$$

$$\frac{\mathit{Vars} = \text{term2vars}(T_1)}{\text{term2proc}(\texttt{letin}(T_1,\ T_2)) = \text{let } \mathit{Vars} \text{ in term2proc}(T_2) \text{ endlet}} \qquad \textbf{[t2p-let]}$$

## 6.4 Messages

The atoms `snd-msg` and `rec-msg` are intended for sending and receiving messages between two processes using synchronous communication. A `snd-msg` can communicate with exactly one `rec-msg` that matches the `snd-msg`'s argument list. Both atoms will assign values to result variables (marked with ?) appearing in their argument lists; these can be used later on in the process expression in which these atoms occur.

Observe that the following specification defines binary, synchronous communication in its full generality, i.e., not only for the atoms snd-msg and rec-msg but for *all* pairs of atoms that can communicate.

**Module** Messages
**imports** ToolBus$^{(5.5)}$
**exports**
  **lexical syntax**
    snd-msg $\rightarrow$ ATOMIC-FUN
    rec-msg $\rightarrow$ ATOMIC-FUN
**equations**
Define relevant communications.

$$\gamma_1(\texttt{snd-msg},\ \texttt{rec-msg}) \ = \ \text{true} \qquad \textbf{[cm-msg]}$$

Define binary, synchronous, communication between processes.

$$\frac{\begin{array}{c}
\mathsf{PR}_1 = \rho_{Pid_1}\ (\lambda_{Env_1}\ (< OAPs_1 + Atom_1\ .\ P_1 + OAPs_1{}' >)), \\
\mathsf{PR}_2 = \rho_{Pid_2}\ (\lambda_{Env_2}\ (< OAPs_2 + Atom_2\ .\ P_2 + OAPs_2{}' >)), \\
\gamma(\mathsf{fun}(Atom_1), \mathsf{fun}(Atom_2)) = \mathsf{true}, \\
C_1 = \mathsf{context}(Pid_1, Env_1, Script, \mathsf{BS}),\ \ C_2 = \mathsf{context}(Pid_2, Env_2, Script, \mathsf{BS}), \\
\mathsf{is\text{-}enabled}(Atom_1, C_1) \wedge \mathsf{is\text{-}enabled}(Atom_2, C_2) = \mathsf{true}, \\
[Ts_1] = \mathsf{args}(Atom_1),\ \ [Ts_2] = \mathsf{args}(Atom_2), \\
([Entries_1], [Entries_2]) = \mathsf{match}([Ts_1], Env_1, [Ts_2], Env_2), \\
Env_1{}' = \mathsf{update}([Entries_1], Env_1),\ \ Env_2{}' = \mathsf{update}([Entries_2], Env_2), \\
\mathsf{PR}_1{}' = \rho_{Pid_1}\ (\lambda_{Env_1{}'}\ (\nu_{\mathsf{info}(Atom_1,\ C_1\ /\ Env_1{}',\ Pid_2)}\ (\Box\ .\ P_1))), \\
\mathsf{PR}_2{}' = \rho_{Pid_2}\ (\lambda_{Env_2{}'}\ (\nu_{\mathsf{info}(Atom_2,\ C_2\ /\ Env_2{}',\ Pid_1)}\ (\Box\ .\ P_2)))
\end{array}}{\begin{array}{l}
\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{Script}\ (\{PRs \mid\mid \Box \mid\mid \mathsf{PR}_1 \mid\mid PRs' \mid\mid \mathsf{PR}_2 \mid\mid PRs''\})), W) = \\
\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\ (\mathsf{E}_{Script}\ (\{PRs \mid\mid \mathsf{PR}_1{}' \mid\mid \mathsf{PR}_2{}' \mid\mid \Box \mid\mid PRs' \mid\mid PRs''\})), \mathsf{true})
\end{array}} \quad \textbf{[msg-trans]}$$

An explanation of the thirteen conditions of this equation is as follows:

1–3 The list of processes contains processes $\mathsf{PR}_1$ and $\mathsf{PR}_1$ whose action prefix form has an alternative that begins with communicating atoms $Atom_1$ and $Atom_2$. Recall that the communication function $\gamma$ was defined in Section 5.1.

4–6 Determine the context of each atom and check that each atom is enabled in its context.

7–9 If the arguments of these atoms match, we get a pair of variable bindings.

10–11 These variable bindings are used to update the environments of $\mathsf{PR}_1$ and $\mathsf{PR}_2$. Observe that a two-way flow of information between sender and receiver is possible due to the possible occurrence of the ? operator in the arguments of both atoms.

12–13 The next step is determined in both process expressions.

In the right-hand side of the conclusion of this equation, the parallel merge of processes is updated. Also note, that the second argument of atomic-steps is set to true to indicate that work could be done in some process.

## 6.5 Expressions

The atomic process $V\ := Term$ assigns the result of evaluating $Term$ to variable $V$. Variables occurring in $Term$ are replaced by their current value. Function symbols occurring in $Term$ are interpreted as already explained in Section 2.3.2.

```
Module Expressions
imports ToolBus(5.5)
exports
  lexical syntax
    asg → ATOMIC-FUN
  context-free syntax
    VAR ":=" TERM                 → ATOM
    interpret(TERM, CONTEXT)      → TERM
    is-asg-compatible(VAR, TERM) → BOOL
equations
```

Extend the preparation of processes (Section 5.2) to the assignment operator:

$$\frac{Var = \text{prep-term}(Name,\ Pnm,\ Env)}{\text{prep-proc}(Name := T,\ Pnm,\ Env) = Var := \text{prep-term}(T,\ Pnm,\ Env)} \qquad \textbf{[prep-asg]}$$

Extend the definition of the functions fun and args (Section 5.1).

$$\text{fun}(Var := T) = \text{asg} \qquad \textbf{[fun-asg]}$$

$$\text{args}(Var := T) = [Var,\ T] \qquad \textbf{[args-asg]}$$

Define an interpretation function for expressions. In many of the following equations (e.g., interpret-1) seemingly unnecessary conditions appear. These conditions are, however, necessary to convert the type of the result of interpret (always of sort TERM) to a more specific type that is included in TERM (e.g., sort BOOL).

*Interpretation of elementary values.*

$$\text{interpret}(Bool,\ C) = Bool \qquad \textbf{[interp-bool]}$$

$$\text{interpret}(Int,\ C) = Int \qquad \textbf{[interp-int]}$$

$$\text{interpret}(String,\ C) = String \qquad \textbf{[interp-str]}$$

$$\text{interpret}(Var,\ C) = \text{value}(Var,\ \text{get-env}(C)) \qquad \textbf{[interp-var]}$$

$$\text{interpret}(Id,\ C) = Id \qquad \textbf{[interp-id]}$$

$$\frac{\text{interpret}([OptTs],\ C) = [OptTs']}{\text{interpret}([T,\ OptTs],\ C) = [\text{interpret}(T,\ C),\ OptTs']} \qquad \textbf{[interp-list-1]}$$

$$\text{interpret}([],\ C) = [] \qquad \textbf{[interp-list-2]}$$

We leave the remaining, elementary, cases undefined (i.e., placeholder and application with one or more arguments). Function application with one or more arguments are therefore only permitted if they are defined by one of the following equations.

*Operations on Booleans.*

$$\frac{\text{interpret}(T,\ C) = Bool}{\text{interpret}(\texttt{not}(T),\ C) = \neg\ Bool} \qquad \textbf{[interp-not]}$$

$$\frac{\text{interpret}(T_1,\ C) = Bool_1,\ \text{interpret}(T_2,\ C) = Bool_2}{\text{interpret}(\text{and}(T_1,\ T_2),\ C) = Bool_1 \wedge Bool_2} \qquad \textbf{[interp-and]}$$

$$\frac{\text{interpret}(T_1,\ C) = Bool_1,\ \text{interpret}(T_2,\ C) = Bool_2}{\text{interpret}(\texttt{or}(T_1,\ T_2),\ C) = Bool_1 \vee Bool_2} \qquad \textbf{[interp-or]}$$

*Equality and inequality.*

$$\text{interpret}(\text{equal}(T_1,\ T_2),\ C) = \text{ms-eq}(\text{interpret}(T_1,\ C);\ \text{interpret}(T_2,\ C)) \qquad \textbf{[interp-equal]}$$

$$\text{interpret}(\texttt{not-equal}(T_1,\ T_2),\ C) = \neg\ \text{ms-eq}(\text{interpret}(T_1,\ C);\ \text{interpret}(T_2,\ C)) \qquad \textbf{[interp-not-equal]}$$

*Operations on integers.*

$$\frac{\text{interpret}(T_1,\ C) = Int_1,\ \text{interpret}(T_2,\ C) = Int_2}{\text{interpret}(\text{add}(T_1,\ T_2),\ C) = Int_1 + Int_2} \qquad \textbf{[interp-add]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = \mathit{Int}_1,\ \mathsf{interpret}(T_2,\ C) = \mathit{Int}_2}{\mathsf{interpret}(\mathtt{sub}(T_1,\ T_2),\ C) = \mathit{Int}_1 - \mathit{Int}_2} \quad \textbf{[interp-sub]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = \mathit{Int}_1,\ \mathsf{interpret}(T_2,\ C) = \mathit{Int}_2}{\mathsf{interpret}(\mathtt{mul}(T_1,\ T_2),\ C) = \mathit{Int}_1 * \mathit{Int}_2} \quad \textbf{[interp-mul]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = \mathit{Int}_1,\ \mathsf{interpret}(T_2,\ C) = \mathit{Int}_2}{\mathsf{interpret}(\mathtt{less}(T_1,\ T_2),\ C) = \mathit{Int}_1 < \mathit{Int}_2} \quad \textbf{[interp-less]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = \mathit{Int}_1,\ \mathsf{interpret}(T_2,\ C) = \mathit{Int}_2}{\mathsf{interpret}(\mathtt{less\text{-}equal}(T_1,\ T_2),\ C) = \mathit{Int}_1 \Leftarrow \mathit{Int}_2} \quad \textbf{[interp-less-equal]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = \mathit{Int}_1,\ \mathsf{interpret}(T_2,\ C) = \mathit{Int}_2}{\mathsf{interpret}(\mathtt{greater}(T_1,\ T_2),\ C) = \mathit{Int}_1 > \mathit{Int}_2} \quad \textbf{[interp-greater]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = \mathit{Int}_1,\ \mathsf{interpret}(T_2,\ C) = \mathit{Int}_2}{\mathsf{interpret}(\mathtt{greater\text{-}equal}(T_1,\ T_2),\ C) = \mathit{Int}_1 \geq \mathit{Int}_2} \quad \textbf{[interp-greater-equal]}$$

*Operations on term lists.*

$$\frac{\mathsf{interpret}(T,\ C) = [T_1,\ \mathit{OptTs}]}{\mathsf{interpret}(\mathtt{first}(T),\ C) = T_1} \quad \textbf{[interp-first]}$$

$$\frac{\mathsf{interpret}(T,\ C) = [T_1,\ \mathit{OptTs}]}{\mathsf{interpret}(\mathtt{next}(T),\ C) = [\mathit{OptTs}]} \quad \textbf{[interp-next]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = T_1{}',\ \mathsf{interpret}(T_2,\ C) = T_2{}'}{\mathsf{interpret}(\mathtt{join}(T_1,\ T_2),\ C) = [\mathsf{append}(\mathsf{mk\text{-}term\text{-}list}(T_1{}');\ \mathsf{mk\text{-}term\text{-}list}(T_2{}'))]} \quad \textbf{[interp-join]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = T_1{}',\ \mathsf{interpret}(T_2,\ C) = T_2{}'}{\mathsf{interpret}(\mathtt{member}(T_1,\ T_2),\ C) = \mathsf{is\text{-}elem}(T_1{}';\ \mathsf{mk\text{-}term\text{-}list}(T_2{}'))} \quad \textbf{[interp-member]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = T_1{}',\ \mathsf{interpret}(T_2,\ C) = T_2{}'}{\mathsf{interpret}(\mathtt{subset}(T_1,\ T_2),\ C) = \mathsf{ms\text{-}subset}(\mathsf{mk\text{-}term\text{-}list}(T_1{}');\ \mathsf{mk\text{-}term\text{-}list}(T_2{}'))} \quad \textbf{[interp-subset]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = T_1{}',\ \mathsf{interpret}(T_2,\ C) = T_2{}'}{\mathsf{interpret}(\mathtt{diff}(T_1,\ T_2),\ C) = [\mathsf{ms\text{-}diff}(\mathsf{mk\text{-}term\text{-}list}(T_1{}');\ \mathsf{mk\text{-}term\text{-}list}(T_2{}'))]} \quad \textbf{[interp-diff]}$$

$$\frac{\mathsf{interpret}(T_1,\ C) = T_1{}',\ \mathsf{interpret}(T_2,\ C) = T_2{}'}{\mathsf{interpret}(\mathtt{inter}(T_1,\ T_2),\ C) = [\mathsf{ms\text{-}inter}(\mathsf{mk\text{-}term\text{-}list}(T_1{}');\ \mathsf{mk\text{-}term\text{-}list}(T_2{}');\ )]} \quad \textbf{[interp-inter]}$$

$$\frac{\mathsf{interpret}(T,\ C) = T'}{\mathsf{interpret}(\mathtt{size}(T),\ C) = \mathsf{ms\text{-}size}(\mathsf{mk\text{-}term\text{-}list}(T'))} \quad \textbf{[interp-size]}$$

*Miscellaneous functions.*

$\mathsf{interpret}(\mathtt{process\text{-}id},\ C) = \mathsf{get\text{-}proc\text{-}id}(C)$ **[interp-process-id]**

$$\frac{\mathsf{get\text{-}proc\text{-}name}(\mathsf{get\text{-}bus\text{-}state}(C),\ \mathsf{get\text{-}proc\text{-}id}(C)) = \mathtt{proc\text{-}name}(String)}{\mathsf{interpret}(\mathtt{process\text{-}name},\ C) = String}$$ **[interp-process-name]**

*Quoting of terms.*

$\mathsf{interpret}(\mathtt{quote}(T),\ C) = \mathsf{substitute}(T,\ \mathsf{get\text{-}env}(C))$ **[interp-quote]**

*The signature of all predefined functions in expressions.*

```
interpret(functions, C) = [function(not(< bool >), < bool >),        [interp-functions]
                           function(and(< bool >, < bool >), < bool >),
                           function(or(< bool >, < bool >), < bool >),
                           function(equal(< term >, < term >), < bool >),
                           function(not-equal(< term >, < term >), < bool >),
                           function(add(< int >, < int >), < int >),
                           function(add(< int >, < int >), < int >),
                           function(sub(< int >, < int >), < int >),
                           function(mul(< int >, < int >), < int >),
                           function(less(< int >, < int >), < bool >),
                           function(less-equal(< int >, < int >), < bool >),
                           function(greater(< int >, < int >), < bool >),
                           function(greater-equal(< int >, < int >), < bool >),
                           function(first(< list >), < term >),
                           function(next(< list >), < list >),
                           function(join(< list >, < list >), < list >),
                           function(member(< term >, < list >), < bool >),
                           function(subset(< list >, < list >), < bool >),
                           function(diff(< list >, < list >), < list >),
                           function(inter(< list >, < list >), < list >),
                           function(size(< list >), < int >),
                           function(process-id, < int >),
                           function(process-name, < str >),
                           function(quote(< term >), < term >),
                           function(functions, < list >)]
```

Define the atomic step resulting from interpretation of an assignment.

$$\frac{\begin{array}{c} T' = \mathsf{interpret}(T,\ C), \\ \mathsf{is\text{-}asg\text{-}compatible}(Var,\ T') = \mathsf{true}, \\ Env' = \mathsf{assign}(Var,\ T',\ \mathsf{get\text{-}env}(C)) \end{array}}{\mathsf{simple\text{-}atomic\text{-}step}(\mathsf{asg}(Var,\ T),\ C) = C\ /\ Env'}$$ **[asg]**

Define the notion *assignment compatible.* Assignment to a variable is allowed if it is either of type term or it has the same type as the interpreted right hand side of the assignment.

$\mathsf{is\text{-}asg\text{-}compatible}(Name\ \$\ Pnm : \mathtt{term},\ T) = \mathsf{true}$ **[is-asg-compat-1]**

$\mathsf{is\text{-}asg\text{-}compatible}(Name\ \$\ Pnm : Type,\ T) = \mathsf{require\text{-}type}(Type,\ T)$ **otherwise** **[is-asg-compat-2]**

Extend the two-way mapping between process expressions and terms for the assignment operator.

$\mathsf{proc2term}(Vname := T) = \mathsf{asg}(Vname,\ T)$ **[p2t-asg]**

$\mathsf{term2proc}(\mathsf{asg}(Vname,\ T)) = Vname := T$ **[t2p-asg]**

## 6.6 Conditionals

Conditionals may have one of the two forms:

- `if` *Term* `then` $P_1$ `else` $P_2$ `fi`: *Term* should evaluate to a Boolean value. If it evaluates to `true`, $P_1$ is executed, otherwise $P_2$ is executed.
- `if` *Term* `then` $P$ `fi`: *Term* should evaluate to a Boolean value and if it evaluates to `true`, $P_1$ is executed. Otherwise, this construct reduces to `delta`.

**Module** Conditionals
**imports** Tscript(5.1) Expressions(6.5) StateRepr(5.3.1) Booleans(4.1)
**exports**
**context-free syntax**

```
if TERM then PROC else PROC fi → PROC
if TERM then PROC fi           → PROC

TERM ":→" ATOM                 → ATOM

add-cond(TERM, AP-FORM)        → AP-FORM
```

**equations**

Conditionals are defined by extending atoms with a Boolean test (denoted by the conditional operator :→) and by propagating tests appearing in conditionals to the level of atoms where they will ultimately be evaluated. Observe that the evaluation of the test and, when it yields true, execution of the following atom should be an indivisible, atomic, action in order to avoid possible intervening changes of the global state.

$\text{fun}(T :\rightarrow Atom) = \text{fun}(Atom)$ **[fun-cond]**

$\text{args}(T :\rightarrow Atom) = \text{args}(Atom)$ **[args-cond]**

Extend the preparation of processes (Section 5.2) for conditionals.

$\text{prep-proc}(\text{if } T \text{ then } P_1 \text{ else } P_2 \text{ fi}, Pnm, Env) =$ **[prep-ifte]**
$\text{if prep-term}(T, Pnm, Env) \text{ then prep-proc}(P_1, Pnm, Env) \text{ else prep-proc}(P_2, Pnm, Env) \text{ fi}$

$\text{prep-proc}(\text{if } T \text{ then } P \text{ fi}, Pnm, Env) =$ **[prep-ift]**
$\text{if prep-term}(T, Pnm, Env) \text{ then prep-proc}(P, Pnm, Env) \text{ fi}$

Extend the expansion of process expressions (Section 5.4) for conditionals.

$\text{expand}(\text{if } T \text{ then } P_1 \text{ else } P_2 \text{ fi}, C) =$ **[exp-ifte]**
$\text{sum}(\text{expand}(\text{if } T \text{ then } P_1 \text{ fi}, C), \text{expand}(\text{if not}(T) \text{ then } P_2 \text{ fi}, C))$

$\text{expand}(\text{if } T \text{ then } P_1 \text{ fi} . P_2, C) = \text{dot}(\text{add-cond}(T, \text{expand}(P_1, C)), P_2)$ **[exp-ift]**

$\text{add-cond}(T, \delta) = \delta$ **[add-cond-1]**

$\text{add-cond}(T, Atom) = T :\rightarrow Atom$ **[add-cond-2]**

$\text{add-cond}(T, Atom . P) = T :\rightarrow Atom . P$ **[add-cond-3]**

$\text{add-cond}(T, < \text{AP} + OAPs >) = \text{sum}(\text{add-cond}(T, \text{AP}), \text{add-cond}(T, < OAPs >))$ **[add-cond-4]**

$\text{add-cond}(T, < >) = \delta$ **[add-cond-5]**

Flatten nested conditions.

$T_1 :\rightarrow T_2 :\rightarrow Atom = \text{and}(T_1, T_2) :\rightarrow Atom$ **[flat-cond]**

Extend the is-enabled predicate for atoms with a non-empty condition.

$$\frac{\text{interpret}(T,\ C) = \mathit{Bool}}{\text{is-enabled}(T :\rightarrow \mathit{Atom},\ C) = \mathit{Bool}} \qquad \textbf{[is-enabled-cond]}$$

Extend the two-way mapping between process expressions and terms for conditionals.

$$\frac{\text{proc2term}(\mathit{Atom}) = \texttt{atom}(\mathit{OptTs})}{\text{proc2term}(T :\rightarrow \mathit{Atom}) = \texttt{atom}(\mathit{OptTs},\ \texttt{cond}(T))} \qquad \textbf{[p2t-cond]}$$

$$\frac{\mathit{Atom} = \text{term2proc}(\texttt{atom}(\mathit{OptTs}_1,\ \mathit{OptTs}_2))}{\text{term2proc}(\texttt{atom}(\mathit{OptTs}_1,\ \texttt{cond}(T),\ \mathit{OptTs}_2)) = T :\rightarrow \mathit{Atom}} \qquad \textbf{[t2p-cond]}$$

## 6.7 Dynamic process creation

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

**Module** Create
**imports** ToolBus$^{(5.5)}$
**exports**
**context-free syntax**

```
  create "(" NAME "(" TERM-LIST ")" "," TERM ")" → ATOM
  create                                         → ATOMIC-FUN

  pname(ATOM)                                    → NAME
```

**equations**
Extend the preparation of processes (Section 5.2) for process creation.

$$\text{prep-proc}(\text{create}(\mathit{Pnm}(\mathit{OptTs}),\ T),\ \mathit{Pnm}',\ \mathit{Env}) = \qquad \textbf{[prep-create]}$$
$$\text{create}(\mathit{Pnm}(\text{prep-term-list}(\mathit{OptTs},\ \mathit{Pnm}',\ \mathit{Env})),\ \text{prep-term}(T,\ \mathit{Pnm}',\ \mathit{Env}))$$

Extend the definition of the functions fun and args (Section 5.1).

$$\text{fun}(\text{create}(\mathit{Pnm}(\mathit{OptTs}),\ T)) \ = \ \text{create} \qquad \textbf{[fun-create]}$$
$$\text{args}(\text{create}(\mathit{Pnm}(\mathit{OptTs}),\ T)) \ = \ [[\mathit{OptTs}],\ T] \qquad \textbf{[args-create]}$$

Define an auxiliary function to extract the process name from a create atom.

$$\text{pname}(\text{create}(\mathit{Pnm}(\mathit{OptTs}),\ T)) \ = \ \mathit{Pnm} \qquad \textbf{[pname-1]}$$

Define the atomic steps for process creation.

$$\frac{\begin{array}{c}
\text{PR} = \rho_{\mathit{Pid}}\ (\lambda_{\mathit{Env}}\ (<\ \mathit{OAPs}' + \mathit{Atom}\ .\ P + \mathit{OAPs}''\ >)),\\
C = \text{context}(\mathit{Pid},\ \mathit{Env},\ \mathit{Script},\ \text{BS}),\\
\text{is-enabled}(\mathit{Atom},\ C) = \text{true},\\
\text{fun}(\mathit{Atom}) = \text{create},\ \ [[\mathit{OptTs}],\ \mathit{Var}\,?] = \text{substitute}(\text{args}(\mathit{Atom}),\ \mathit{Env}),\\
\mathit{Pnm} = \text{pname}(\mathit{Atom}),\\
C' = C\ /\ \text{assign}(\mathit{Var},\ \text{get-new-proc-id}(\text{BS}),\ \text{get-env}(C)),\\
\text{PR}' = \rho_{\mathit{Pid}}\ (\lambda_{\mathit{Env}}\ (\nu_{\text{info}(\mathit{Atom},\ C',\ \text{proc-id}(-\ 1))}\ (\Box\ .\ P)))
\end{array}}{\begin{array}{l}
\text{atomic-steps}(\lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1\ ||\ \Box\ ||\ \text{PR}\ ||\ \mathit{PRs}_2\})),\ W) =\\
\text{atomic-steps}(\text{add-proc}(\mathit{Pnm}(\mathit{OptTs}),\ \mathit{Pid},\ \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1\ ||\ \text{PR}'\ ||\ \Box\ ||\ \mathit{PRs}_2\}))),\ \text{true})
\end{array}} \qquad \textbf{[create-trans]}$$

Extend the two-way mapping between process expressions and terms for the create operator.

```
proc2term(create(Pnm(OptTs), T)) =                                   [p2t-create]
 atom("create", name2string(Pnm), [OptTs], T)
term2proc(atom("create", string("\"" Chars "\""), [OptTs], T)) =     [t2p-create]
 create(name(Chars)(OptTs), T)
```

## 6.8 Notes

Notes provide an asynchronous, reliable, broadcasting mechanism. They are defined by the following atoms:

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

```
Module Notes
imports ToolBus(5.5)
exports
  sorts SUBSCRIPTIONS NOTES
  lexical syntax
    snd-note    → ATOMIC-FUN
    rec-note    → ATOMIC-FUN
    no-note     → ATOMIC-FUN
    subscribe   → ATOMIC-FUN
    unsubscribe → ATOMIC-FUN
  context-free syntax
    subs-list(TERM-LIST)                       → SUBSCRIPTIONS
    note-list(TERM-LIST)                       → NOTES

    NOTES                                      → BUS-VAL
    SUBSCRIPTIONS                              → BUS-VAL

    get-subs(BUS-STATE, PROC-ID)               → SUBSCRIPTIONS
    get-notes(BUS-STATE, PROC-ID)              → NOTES
    matching-subscription(TERM, SUBSCRIPTIONS) → BOOL
    matching-note(TERM, NOTES)                 → BOOL
    del-notes(TERM, NOTES)                     → NOTES
    distr-note(TERM, PROCESSES, BUS-STATE)     → BUS-STATE
exports
  variables
    S [0-9']*   → SUBSCRIPTIONS
    N [0-9']*   → NOTES
    Note        → TERM
```

*Notes* $[0\text{-}9']{*} \rightarrow$ {TERM ","}*
**equations**
Extend PROC-REPR to support *notes.* This is achieved by adding two new components to the process representation:

- subs: a list of subscriptions (representing all forms of notes to which the process has subscribed).
- notes: a list of notes (representing received but unread notes).

Define access functions to support subscriptions and notes.

$$\text{BS . notes}(Pid) = \text{no-bus-val} \Rightarrow \text{get-notes}(\text{BS}, Pid) = \text{note-list}() \quad \textbf{[get-notes-1]}$$
$$\text{BS . notes}(Pid) = N \Rightarrow \text{get-notes}(\text{BS}, Pid) = N \quad \textbf{[get-notes-2]}$$

$$\text{BS . subs}(Pid) = \text{no-bus-val} \Rightarrow \text{get-subs}(\text{BS}, Pid) = \text{subs-list}() \quad \textbf{[get-subs-1]}$$
$$\text{BS . subs}(Pid) = S \Rightarrow \text{get-subs}(\text{BS}, Pid) = S \quad \textbf{[get-subs-2]}$$

Define three utility functions for deleting notes from a list of notes, for finding a matching subscription in a list of subscriptions, and for distributing a note to all subscribed TOOLBUS processes.

*Delete notes from a list of notes.*

$$\text{del-notes}(T, \text{note-list}()) = \text{note-list}() \quad \textbf{[del-notes-1]}$$

$$\frac{\text{cmatchp}(T, Note) = \text{true}}{\text{del-notes}(T, \text{note-list}(Note, Notes)) = \text{del-notes}(T, \text{note-list}(Notes))} \quad \textbf{[del-notes-2]}$$

$$\frac{\text{cmatchp}(T, Note) = \text{false},\ \text{del-notes}(T, \text{note-list}(Notes)) = \text{note-list}(Notes')}{\text{del-notes}(T, \text{note-list}(Note, Notes)) = \text{note-list}(Note, Notes)} \quad \textbf{[del-notes-3]}$$

*Find a matching subscription.*

$$\frac{S = \text{subs-list}(OptTs_1, T', OptTs_2),\ \text{cmatchp}(T', T) = \text{true}}{\text{matching-subscription}(T, S) = \text{true}} \quad \textbf{[matching-subs-1]}$$

$$\text{matching-subscription}(T, S) = \text{false} \quad \textbf{otherwise} \quad \textbf{[matching-subs-2]}$$

*Find a matching note.*

$$\frac{N = \text{notes}(OptTs_1, T', OptTs_2),\ \text{cmatchp}(T', T) = \text{true}}{\text{matching-note}(T, N) = \text{true}} \quad \textbf{[matching-note-1]}$$

$$\text{matching-note}(T, N) = \text{false} \quad \textbf{otherwise} \quad \textbf{[matching-note-2]}$$

*Distribute a note to all subscribed* TOOLBUS *processes.*

$$\frac{\begin{array}{c} Pid = \text{get-pid}(\text{PR}), \\ \text{matching-subscription}(Note, \text{get-subs}(\text{BS}, Pid)) = \text{true}, \\ \text{note-list}(Notes) = \text{get-notes}(\text{BS}, Pid), \\ \text{BS}' = \text{BS}[\text{notes}(Pid) := \text{note-list}(Notes, Note)] \end{array}}{\text{distr-note}(Note, \{\text{PR} \mid\mid PRs\}, \text{BS}) = \text{distr-note}(Note, \{PRs\}, \text{BS}')} \quad \textbf{[distr-nts-1]}$$

$$\frac{\begin{array}{c} Pid = \text{get-pid}(\text{PR}), \\ \text{matching-subscription}(Note, \text{get-subs}(\text{BS}, Pid)) = \text{false} \end{array}}{\text{distr-note}(Note, \{\text{PR} \mid\mid PRs\}, \text{BS}) = \text{distr-note}(Note, \{PRs\}, \text{BS})} \quad \textbf{[distr-nts-2]}$$

$$\text{distr-note}(\mathit{Note}, \{\}, \mathsf{BS}) \;=\; \mathsf{BS} \qquad \textbf{[distr-nts-3]}$$

Define the atomic steps required for the atoms snd-note, rec-note, no-note, subscribe and unsubscribe.

$$\frac{\begin{array}{c}
\mathsf{PR} = \rho_{\mathit{Pid}}\,(\lambda_{\mathit{Env}}\,(< \mathit{OAPs}' + \mathit{Atom}\,.\,P + \mathit{OAPs}'' >)),\\
C = \text{context}(\mathit{Pid}, \mathit{Env}, \mathit{Script}, \mathsf{BS}),\\
\text{is-enabled}(\mathit{Atom}, C) = \text{true},\\
\text{fun}(\mathit{Atom}) = \texttt{snd-note},\ \ \text{args}(\mathit{Atom}) = [T],\\
T' = \text{substitute}(T, \mathit{Env}),\\
\mathsf{BS}' = \text{distr-note}(T', \{\mathit{PRs}_1 \mid\mid \mathit{PRs}_2\}, \mathsf{BS}),\\
\mathsf{PR}' = \rho_{\mathit{Pid}}\,(\lambda_{\mathit{Env}}\,(\nu_{\text{info}(\mathit{Atom},\, C\,/\,\mathsf{BS}',\, \text{proc-id}(-\,1))}\,(\Box\,.\,P)))
\end{array}}{\begin{array}{c}
\text{atomic-steps}(\lambda_{\mathsf{BS}}\,(\mathsf{E}_{\mathit{Script}}\,(\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \mathsf{PR} \mid\mid \mathit{PRs}_2\})), W) =\\
\text{atomic-steps}(\lambda_{\mathsf{BS}'}\,(\mathsf{E}_{\mathit{Script}}\,(\{\mathit{PRs}_1 \mid\mid \mathsf{PR}' \mid\mid \Box \mid\mid \mathit{PRs}_2\})), \text{true})
\end{array}} \qquad \textbf{[snd-note]}$$

$$\frac{\begin{array}{c}
\mathsf{BS} = \text{get-bus-state}(C),\\
\mathit{Env} = \text{get-env}(C),\\
\mathit{Pid} = \text{get-proc-id}(C),\\
T' = \text{substitute}(T, \mathit{Env}),\\
\text{get-notes}(\mathsf{BS}, \mathit{Pid}) = \text{note-list}(\mathit{Notes}, T_1, \mathit{Notes}'),\\
([\mathit{Entries}], [\mathit{Entries}_1]) = \text{match}(T', \mathit{Env}, T_1, []),\\
\mathit{Env}' = \text{update}([\mathit{Entries}], \mathit{Env}),\\
\mathsf{BS}' = \mathsf{BS}[\texttt{notes}(\mathit{Pid}) := \text{note-list}(\mathit{Notes}, \mathit{Notes}')]
\end{array}}{\text{simple-atomic-step}(\texttt{rec-note}(T), C) = C\,/\,\mathit{Env}'\,/\,\mathsf{BS}'} \qquad \textbf{[rec-note]}$$

$$\frac{\begin{array}{c}
T' = \text{substitute}(T, \text{get-env}(C)),\\
\text{matching-note}(T', \text{get-notes}(\text{get-bus-state}(C), \text{get-proc-id}(C))) = \text{false}
\end{array}}{\text{simple-atomic-step}(\texttt{no-note}(T), C) = C} \qquad \textbf{[no-note]}$$

Observe that the subscription list is a multi-set of note names.

$$\frac{\begin{array}{c}
\mathsf{BS} = \text{get-bus-state}(C),\\
\mathit{Pid} = \text{get-proc-id}(C),\\
\text{get-subs}(\mathsf{BS}, \mathit{Pid}) = \text{subs-list}(\mathit{OptTs}),\\
T' = \text{substitute}(T, \text{get-env}(C)),\\
\text{has-no-vars}(T') = \text{true},\\
S = \text{subs-list}(\mathit{OptTs}, T'),\\
\mathsf{BS}' = \mathsf{BS}[\texttt{subs}(\mathit{Pid}) := S]
\end{array}}{\text{simple-atomic-step}(\texttt{subscribe}(T), C) = C\,/\,\mathsf{BS}'} \qquad \textbf{[subscribe]}$$

$$\frac{\begin{array}{c}
\mathsf{BS} = \text{get-bus-state}(C),\\
\mathit{Pid} = \text{get-proc-id}(C),\\
T' = \text{substitute}(T, \text{get-env}(C)),\\
\text{has-no-vars}(T') = \text{true},\\
\text{get-subs}(\mathsf{BS}, \mathit{Pid}) = \text{subs-list}(\mathit{OptTs}_1, T', \mathit{OptTs}_2),\\
S = \text{subs-list}(\mathit{OptTs}_1, \mathit{OptTs}_2),\\
N = \text{del-notes}(T', \text{get-notes}(\mathsf{BS}, \mathit{Pid})),\\
\mathsf{BS}' = \mathsf{BS}[\texttt{subs}(\mathit{Pid}) := S][\texttt{notes}(\mathit{Pid}) := N]
\end{array}}{\text{simple-atomic-step}(\texttt{unsubscribe}(T), C) = C\,/\,\mathsf{BS}'} \qquad \textbf{[unsubscribe]}$$

## 6.9 Discrete time

**Module** DiscreteTime
**imports** BusState$^{(5.3.2)}$ Expressions$^{(6.5)}$
**exports**
  **context-free syntax**
```
    get-time(BUS-STATE)      → INT
    get-max-time(BUS-STATE)  → INT
```
**equations**
Extend the bus state with a time field.

$$\text{BS} . \texttt{time} = \text{no-bus-val} \Rightarrow \text{get-time(BS)} = 0 \quad \text{[get-time-1]}$$
$$\text{BS} . \texttt{time} = \mathit{Int} \Rightarrow \text{get-time(BS)} = \mathit{Int} \quad \text{[get-time-2]}$$

Extend the bus state with a max time field.

$$\text{BS} . \texttt{max-time} = \text{no-bus-val} \Rightarrow \text{get-max-time(BS)} = 0 \quad \text{[get-max-time-1]}$$
$$\text{BS} . \texttt{max-time} = \mathit{Int} \Rightarrow \text{get-max-time(BS)} = \mathit{Int} \quad \text{[get-max-time-2]}$$

Extend expressions with a new function for obtaining the current (absolute) time.

$$\text{interpret}(\texttt{current-time}, C) = \text{get-time}(\text{get-bus-state}(C)) \quad \text{[interp-current-time]}$$

## 6.10 Delay and timeout

The following attributes can be attached to atomic processes in order to define their behaviour in time:

- `delay`: relative execution delay.
- `abs-delay`: absolute execution delay.
- `timeout`: relative timeout for execution.
- `abs-timeout`: absolute timeout for execution.

We only permit the following combinations of these attributes:

- relative time: `delay`, `delay/timeout`, `timeout`.
- absolute time: `abs-delay`, `abs-delay/abs-timeout`, `abs-timeout`.

Other combinations, e.g., mixtures of relative and absolute time are forbidden. This is a static constraint on the **T** script. For reasons of simplicity, however, we do not enforce this constraint in the following specification.

**Module** Delay-Timeout
**imports** ToolBus$^{(5.5)}$ DiscreteTime$^{(6.9)}$ Conditionals$^{(6.6)}$
**exports**
  **sorts** TIMER-FUN TIMER
  **lexical syntax**
```
    delay       → TIMER-FUN
    abs-delay   → TIMER-FUN
    timeout     → TIMER-FUN
    abs-timeout → TIMER-FUN
```

**context-free syntax**

```
  TIMER-FUN "(" TERM ")" → TIMER
  ATOM TIMER             → ATOM
```

**variables**

```
  Time     → INT
  Start    → INT
  End      → INT
  TimerFun → TIMER-FUN
```

**equations**

Extend the preparation of processes (Section 5.2) to atoms with delays and timeouts.

$$\frac{\mathit{Atom}' = \text{prep-proc}(\mathit{Atom}, \mathit{Pnm}, \mathit{Env}),\ \ T' = \text{prep-term}(T, \mathit{Pnm}, \mathit{Env})}{\text{prep-proc}(\mathit{Atom}\ \mathit{TimerFun}(T), \mathit{Pnm}, \mathit{Env}) = \mathit{Atom}'\ \mathit{TimerFun}(T')} \qquad \textbf{[prep-timer]}$$

Extend expansion for atoms with delays and timeouts. At the moment of expansion, a condition is constructed representing the desired delay or timeout. It contains the subexpression `current-time` that will yield the current time at the moment that the condition is evaluated. In the generated condition, relative time is always converted into absolute time.

$$\frac{\begin{array}{c}\mathit{Start} = \text{interpret}(T, C),\ \ \mathit{Atom}'\ .\ P = \text{expand}(\mathit{Atom}\ .\ P, C),\\ \mathit{Time} = \text{get-time}(\text{get-bus-state}(C))\end{array}}{\begin{array}{l}\text{expand}(\mathit{Atom}\ \texttt{delay}(T)\ .\ P, C) =\\ \texttt{greater-equal(current-time},\ \mathit{Start}\ +\ \mathit{Time})\ :\!\rightarrow \mathit{Atom}'\ .\ P\end{array}} \qquad \textbf{[exp-delay]}$$

$$\frac{\mathit{Start} = \text{interpret}(T, C),\ \ \mathit{Atom}'\ .\ P = \text{expand}(\mathit{Atom}\ .\ P, C)}{\begin{array}{l}\text{expand}(\mathit{Atom}\ \texttt{abs-delay}(T)\ .\ P, C) =\\ \texttt{greater-equal(current-time},\ \mathit{Start})\ :\!\rightarrow \mathit{Atom}'\ .\ P\end{array}} \qquad \textbf{[exp-abs-delay]}$$

$$\frac{\begin{array}{c}\mathit{End} = \text{interpret}(T, C),\ \ \mathit{Atom}'\ .\ P = \text{expand}(\mathit{Atom}\ .\ P, C),\\ \mathit{Time} = \text{get-time}(\text{get-bus-state}(C))\end{array}}{\begin{array}{l}\text{expand}(\mathit{Atom}\ \texttt{timeout}(T)\ .\ P, C) =\\ \texttt{less-equal(current-time},\ \mathit{End}\ +\ \mathit{Time})\ :\!\rightarrow \mathit{Atom}'\ .\ P\end{array}} \qquad \textbf{[exp-timeout]}$$

$$\frac{\mathit{End} = \text{interpret}(T, C),\ \ \mathit{Atom}'\ .\ P = \text{expand}(\mathit{Atom}\ .\ P, C)}{\begin{array}{l}\text{expand}(\mathit{Atom}\ \texttt{abs-timeout}(T)\ .\ P, C) =\\ \texttt{less-equal(current-time},\ \mathit{End})\ :\!\rightarrow \mathit{Atom}'\ .\ P\end{array}} \qquad \textbf{[exp-abs-timeout]}$$

Extend the two-way mapping between process expressions and terms for timer functions.

$$\begin{array}{l}\text{proc2term}(\text{atomic-fun}(\mathit{Chars}_1)(\mathit{OptTs})\ \text{timer-fun}(\mathit{Chars}_2)(T)) =\\ \texttt{atom}(\text{id}(\mathit{Chars}_1)(\mathit{OptTs}),\ \text{id}(\mathit{Chars}_2)(T))\end{array} \qquad \textbf{[p2t-timer]}$$

$$\begin{array}{l}\text{term2proc}(\texttt{atom}(\text{id}(\mathit{Chars}_1)(\mathit{OptTs}),\ \text{id}(\mathit{Chars}_2)(T))) =\\ \text{atomic-fun}(\mathit{Chars}_1)(\mathit{OptTs})\ \text{timer-fun}(\mathit{Chars}_2)(T)\end{array} \qquad \textbf{[t2p-timer]}$$

# Chapter 7

# Features of ToolBus tools

## 7.1 Tool definitions

We extend **T** scripts (Section 5.1) with tool definitions. A tool has a *name*, formal parameters, and is characterized by a number of features: a list of (identifier, string) pairs. Before a tool is executed, occurrences of formal parameter names in the strings defining features are replaced by their actual value.

Our general approach here is that a tool definition should contain all information needed to execute an instance of the tool, but we do not specify how a tool instance comes into existence. The interpretation of the names of features is therefore not fixed here, but in the examples we will assume the following feature names:

- *command*: the command needed to start the execution of a tool;
- *host*: the computer on which the tool will be executing.

In a similar manner as for process definitions (see Section 5.1), we introduce a function definition to map a tool name onto its definition.

```
Module ToolDefs
imports ToolBus$^{(5.5)}$
exports
  sorts FEATURE-ASG FEATURES TOOL-ID
  context-free syntax
    ID ":=" STRING                    → FEATURE-ASG
    "{" {FEATURE-ASG ";"}* "}"        → FEATURES
    tool ID FORMALS is FEATURES       → DEF
    tool-definition(ID, T-SCRIPT)     → DEF

    ID "(" INT ")"                    → TOOL-ID
    TOOL-ID                           → TERM
    TOOL-ID                           → BUS-VAL
    get-new-tool-id(ID, BUS-STATE)    → TOOL-ID
  variables
    Features [0-9']*→ {FEATURE-ASG ";"}*
exports
  context-free syntax
    string-repr(TERM)                           → STRING
    replace(STRING, ENV)                        → STRING
    features2term(FEATURES, ENV)                → TERM
    add-TCP-defs(T-SCRIPT)                      → T-SCRIPT
    get-controlled-tool(BUS-STATE, PROC-ID)     → TOOL-ID
```

get-controlled-ext-tool(BUS-STATE, PROC-ID) → INT
get-controlling-process(BUS-STATE, TOOL-ID) → PROC-ID
**exports**
**variables**
*Tid* [*0-9'*]∗→ TOOL-ID
**hiddens**
**variables**
*Chars* [*0-9'*]∗→ CHAR∗
**equations**

Add a field new-tool-id to the bus state to keep track of tool instances.

$$\text{BS . new-tool-id} = \text{no-bus-val} \Rightarrow \text{get-new-tool-id}(\mathit{Id}, \text{BS}) = \mathit{Id}(0) \quad \textbf{[get-new-tool-id-1]}$$

$$\text{BS . new-tool-id} = \text{tool-id}(\mathit{Int}) \Rightarrow \text{get-new-tool-id}(\mathit{Id}, \text{BS}) = \mathit{Id}(\mathit{Int}) \quad \textbf{[get-new-tool-id-2]}$$

Define an auxiliary function string-repr that maps a term onto a string representation (if possible). Observe that Asf+Sdf gives access to the text of lexical items by means of a standard convention: the sort name in question (e.g., STRING) (written in all lower case letters, e.g., string) acts as a conversion function from lists of characters (the built-in sort CHAR) to e.g. STRING. In the following equations, the functions string, nat-con, and svar play this role.

$$\text{string-repr}(\text{string}(\text{"\""}\ \mathit{Chars}\ \text{"\""})) = \text{string}(\text{"\""}\ \mathit{Chars}\ \text{"\""}) \quad \textbf{[str-repr-1]}$$

$$\text{string-repr}(\text{nat-con}(\mathit{Chars})) = \text{string}(\text{"\""}\ \mathit{Chars}\ \text{"\""}) \quad \textbf{[str-repr-2]}$$

$$\text{string-repr}(T) = T \quad \textbf{otherwise} \quad \textbf{[str-repr-3]}$$

Define a replacement function on strings: given a string and an environment (representing the replacements to be performed) a new string is constructed in which all variables have been replaced by their string representation.

$$\text{replace}(\mathit{String}, []) = \mathit{String} \quad \textbf{[replace-1]}$$

$$\frac{\begin{array}{c}\mathit{Vname} = \text{name}(\mathit{Chars}_2),\\ \text{string-repr}(T) = \text{string}(\text{"\""}\ \mathit{Chars}_4\ \text{"\""})\end{array}}{\begin{array}{l}\text{replace}(\text{string}(\text{"\""}\ \mathit{Chars}_1\ \mathit{Chars}_2\ \mathit{Chars}_3\ \text{"\""}), [\mathit{Vname} : \mathit{Type} \mapsto T, \mathit{Entries}]) =\\ \text{replace}(\text{string}(\text{"\""}\ \mathit{Chars}_1\ \mathit{Chars}_4\ \mathit{Chars}_3\ \text{"\""}), [\mathit{Vname} : \mathit{Type} \mapsto T, \mathit{Entries}])\end{array}} \quad \textbf{[replace-2]}$$

$$\text{replace}(\mathit{String}, [\mathit{Var} \mapsto T, \mathit{Entries}]) = \text{replace}(\mathit{String}, [\mathit{Entries}]) \quad \textbf{otherwise} \quad \textbf{[replace-3]}$$

Given a list of features and an environment, transform the features into a list of terms representing the features after proper replacement of variables.

$$\text{features2term}(\{\}, \mathit{Env}) = [] \quad \textbf{[f2t-1]}$$

$$\frac{\begin{array}{c}\text{replace}(\mathit{String}, \mathit{Env}) = \mathit{String}',\\ \text{features2term}(\{\mathit{Features}\}, \mathit{Env}) = [\mathit{OptTs}]\end{array}}{\text{features2term}(\{\mathit{Id} := \mathit{String};\ \mathit{Features}\}, \mathit{Env}) = [\mathit{Id}(\mathit{String}'), \mathit{OptTs}]} \quad \textbf{[f2t-2]}$$

Define a function that extracts the tool definition from a ToolBus script corresponding to given tool name.

$$\begin{array}{l}\text{tool-definition}(\mathit{Id}, \mathit{Defs}_1\ \text{tool}\ \mathit{Id}\ \mathit{Formals}\ \text{is}\ \{\mathit{Features}\}\ \mathit{Defs}_2\ \mathit{TB\text{-}Config}) =\\ \quad \text{tool}\ \mathit{Id}\ \mathit{Formals}\ \text{is}\ \{\mathit{Features}\}\end{array} \quad \textbf{[tool-definition-1]}$$

Get the tool identification of the tool controlled by a (tool control) process.

$$\text{BS . control-tool}(\mathit{Pid}) = [\mathit{Tid}, \mathit{Int}] \Rightarrow \text{get-controlled-tool}(\text{BS}, \mathit{Pid}) = \mathit{Tid} \quad \textbf{[get-ct-1]}$$

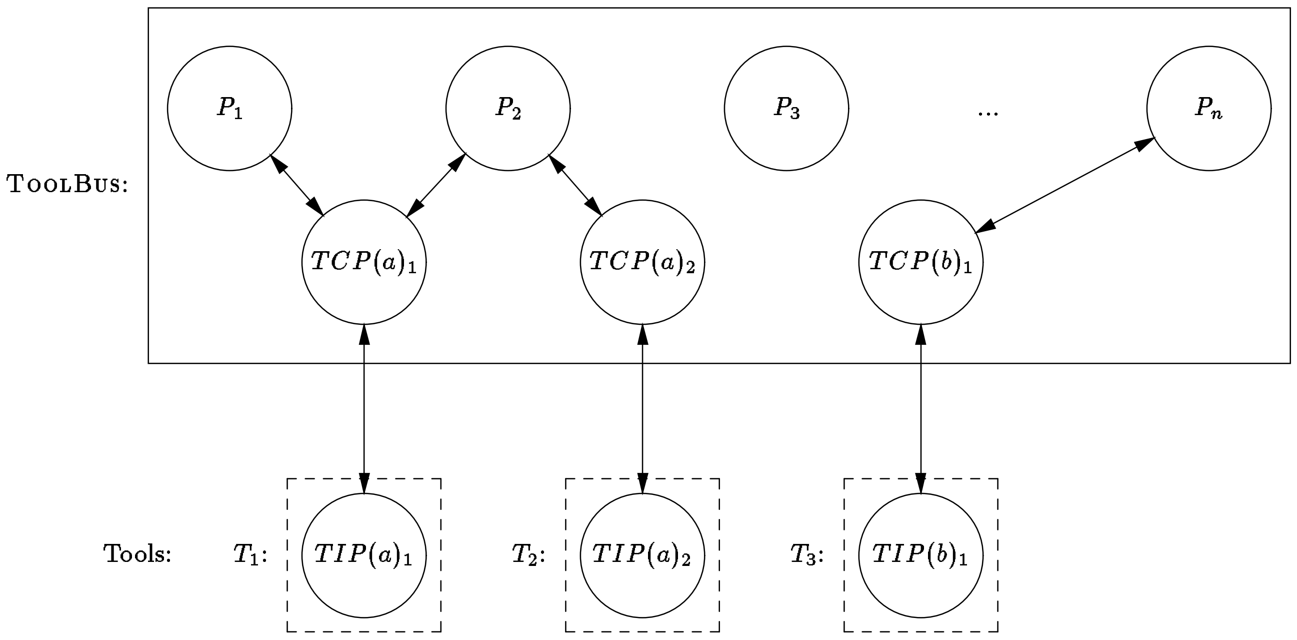

Figure 7.1: Organization of the ToolBus including control over tools

$$\text{get-controlled-tool}(\mathsf{BS}, \mathit{Pid}) \;=\; \mathtt{tool\text{-}id}(-\,1) \quad \textbf{otherwise} \qquad \textbf{[get-ct-2]}$$

Get the external tool identification of the tool controlled by a (tool control) process.

$$\mathsf{BS}\;.\;\mathtt{control\text{-}tool}(\mathit{Pid}) = [\mathit{Tid},\ \mathit{Int}] \;\Rightarrow \text{get-controlled-ext-tool}(\mathsf{BS}, \mathit{Pid}) \;=\; \mathit{Int} \qquad \textbf{[get-cet-1]}$$

$$\text{get-controlled-ext-tool}(\mathsf{BS}, \mathit{Pid}) \;=\; -\,1 \quad \textbf{otherwise} \qquad \textbf{[get-cet-2]}$$

Get the process identification of the tool control process controlling a given tool.

$$\frac{\mathsf{BS} = \text{bus-state}(\mathit{BusAsgs}_1;\ \mathtt{control\text{-}tool}(\mathit{Pid}) := [\mathit{Tid},\ \mathit{Tid}'];\ \mathit{BusAsgs}_2)}{\text{get-controlling-process}(\mathsf{BS},\ \mathit{Tid}) = \mathit{Pid}} \qquad \textbf{[get-cp-1]}$$

## 7.2 Controlling tools

In order to make explicit how tools are being controlled by the ToolBus, we will now refine the global architecture of the ToolBus as already sketched in Figure 2.1.

A refined view of the ToolBus architecture is shown in Figure 7.1. It is more detailed in the following respects:

- For each tool instance we introduce one *tool control process* (TCP): a process inside the ToolBus that controls the interactions between all processes in the ToolBus with this instance of the tool.
- For each tool instance we introduce (outside the ToolBus): a single *tool instance process* (TIP): a process that controls the behaviour of this particular tool instance.

Observe that TCPs are ToolBus processes (and thus run *inside* the ToolBus), while TIPs run "in the outside world", i.e., they are introduced for the purpose of modeling the behaviour of tools in the

| ToolBus Proc. | Tool Control Process | | Tool | § |
|---|---|---|---|---|
| rec-connect | snd-connect-by-TCP | rec-connect-before-TCP | snd-connect | 7.3 |
| new-tool-id† | | | | 7.3 |
| | control-tool-by-TCP | | | 7.3 |
| execute | | | | 7.4 |
| snd-execute-to-tool† | | | rec-execute | 7.4 |
| snd-eval | rec-eval-by-TCP | snd-eval-by-TCP | rec-eval | 7.5 |
| rec-value | snd-value-by-TCP | rec-value-by-TCP | snd-value | 7.5 |
| snd-cancel | rec-cancel-by-TCP | snd-cancel-by-TCP | rec-cancel | 7.5 |
| snd-do | rec-do-by-TCP | snd-do-by-TCP | rec-do | 7.5 |
| rec-event | snd-event-by-TCP | rec-event-by-TCP | snd-event | 7.6 |
| snd-ack-event | rec-ack-event-by-TCP | snd-ack-event-by-TCP | rec-ack-event | 7.6 |
| rec-disconnect | snd-disconnect-by-TCP | rec-disconnect-by-TCP | snd-disconnect | 7.3 |
| snd-terminate | rec-terminate-by-TCP | snd-terminate-by-TCP | rec-terminate | 7.4 |
| attach-monitor | | rec-attach-monitor-by-TCP | snd-attach-monitor | 7.7 |
| detach-debugger | | rec-detach-debugger-by-TCP | snd-detach-monitor | 7.7 |
| snd-monitor† | | | | 7.7 |
| rec-monitor† | | | | 7.7 |
| continuation† | | | | 7.7 |
| shutdown | | | | 7.8 |
| reconfigure | | rec-reconfigure-by-TCP | snd-reconfigure | 7.8 |
| restart† | | | | 7.8 |
| | | | pick | 7.2.1 |

Figure 7.2: Overview of tool-related primitives

**Notes:**

(1) Primitives labeled with † are auxiliary notions used in the definition, but they are not directly available when defining ToolBus processes.

(2) All primitives in the two columns labeled Tool Control Process are intended for defining the Tool Control Process, but they are not directly available when defining other ToolBus processes.

outside world, but they are not part of the ToolBus proper. Each TIP can be understood as the potential of the outside world to execute one instance of a certain tool. Also note that the phrase "tool instance" is used to indicate that several tool instance processes executing in the outside world may correspond to a *single* tool definition in the **T** script.

In Figure 7.1 we see:

- Ordinary ToolBus processes $P_1, ..., P_n$.
- Two tool instance processes for tool "a": $TIP(a)_1$ and $TIP(a)_2$.
- One tool instance process for tool "b": $TIP(b)_1$.
- Three tool control protocols: $TCP(a)_1$ (controls $TIP(a)_1$), $TCP(a)_2$ (controls $TIP(a)_2$), $TCP(b)_1$ (controls $TIP(b)_1$).

The behaviour of the tools represented in Figure 2.1 by boxes labeled $T_1$, $T_2$, $T_3$, is now actually described by the processes $TIP(a)_1$, $TIP(a)_2$, and $TIP(b)_1$, respectively.

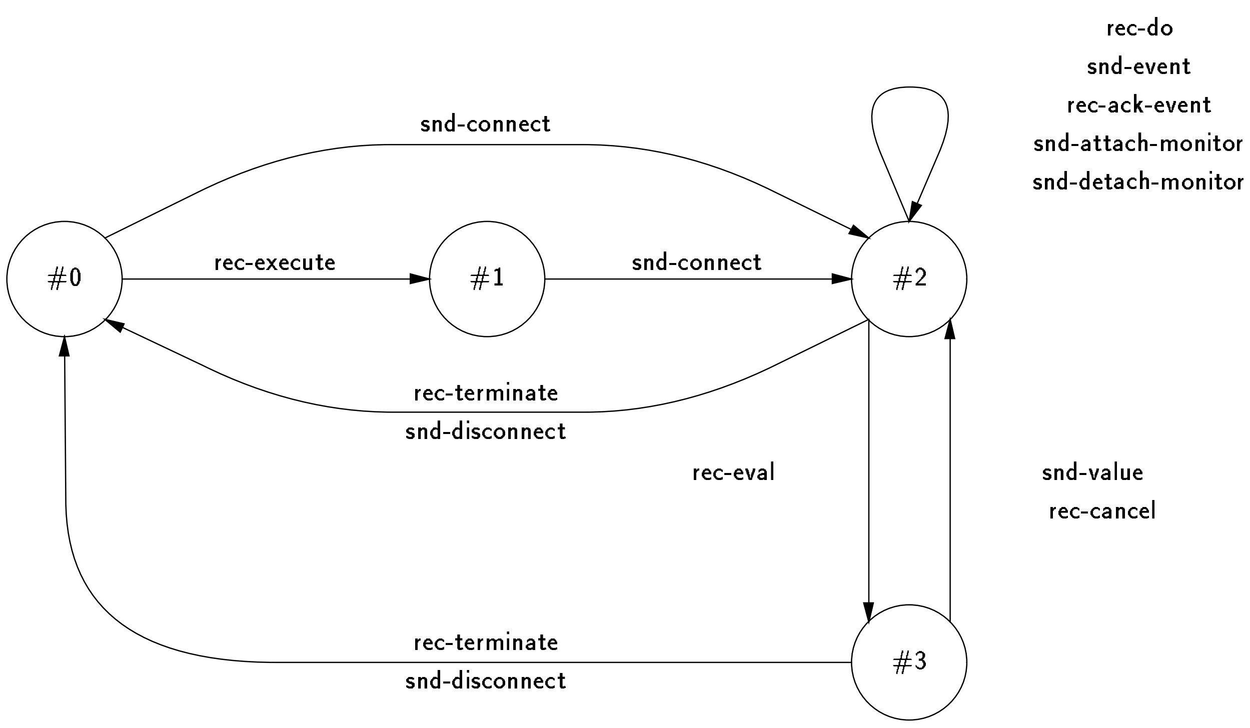

Figure 7.3: The tool instance process (TIP)

The tool instance process is now first described in Section 7.2.1. The tool control process is described in Section 7.2.2. For reference purposes, we give here already a preview of all tool-related primitives in Figure 7.2.

### 7.2.1 The tool instance process

The control over tools is best understood by first considering the behaviour of a tool as shown in Figure 7.3. We give here now a simplified textual description of tool behaviour. A tool can be in one of four states:

- State #0: the tool is either not executing at all, or it is executing but not connected to the ToolBus. In this state an execution request from the ToolBus (rec-execute) can be received causing a transition to state #1, or a connection request can be sent to the ToolBus (snd-connect) causing a transition to state #2. In both cases, the tool receives a unique tool identification from the ToolBus that remains valid until the execution or connection of this tool instance ends.

- State #1: the tool has already accepted an execution request before entering this state; it will leave this state when the preparations for executing/connecting the tool have been completed by sending a connect request (snd-connect) and going to state #2.

- State #2: the tool is executing and connected to the ToolBus. The atom rec-eval causes a transition to state #3. The atoms rec-do, snd-event and rec-ack-event cause a self-transition. The atoms rec-terminate and snd-disconnect cause a transition back to state #0.

  Occurrences of events of the form snd-event(*ToolId, Term*) and their acknowledgement rec-ack-event(*ToolId, Term*) are controlled by maintaining a *list of pending, unacknowledged, events U* and enforcing the following rules:

  – a snd-event transition is only allowed if *Term* does not occur in *U*;

  - when a snd-event transition is made, *Term* is added to *U*;
  - a rec-ack-event transition for event *Term* is only possible if *Term* occurs in *U*; it is removed from *U* when the transition is made.

- State #3: the atom snd-value causes a transition to state #2, while the atoms rec-terminate and snd-disconnect cause a transition back to state #0.

This behaviour of a *tool* can be specified as follows.

## Module ToolInstanceProcess

**imports** Execute-Terminate$^{(7.4)}$ Connect-Disconnect$^{(7.3)}$ Eval-Do$^{(7.5)}$ Events$^{(7.6)}$ Iteration$^{(6.1)}$ Conditionals$^{(6.6)}$ Notes$^{(6.8)}$ Create$^{(6.7)}$ Monitoring$^{(7.7)}$ Reconfigure$^{(7.8)}$ Let$^{(6.3)}$

**exports**

**context-free syntax**

```
pick(TERM-LIST)        → PROC
def-TIP                → DEF
def-TIP-EVAL-DO        → DEF
def-TIP-EVENTS         → DEF
def-TIP-MONITOR        → DEF
def-TIP-RECONFIGURE    → DEF
```

**equations**

A *tool instance process* TIP describes the behaviour of one instance of one tool. We do not know, of course, anything about the internals of each tool instance. It is impossible to describe which values the tool computes, which events it generates, and how long it takes to perform these computations. The only thing we do know, and that we should make precise, is the external behaviour of each tool instance as seen from the TOOLBUS.

We define a skeleton process definition for tool instances and introduce the process pick to model the aspects we cannot describe: it assigns an arbitrary term to the result variables appearing in its argument list. We do not give a further definition for pick. Each actual tool instance process should satisfy the following constraints:

- It is a specialization of the process TIP defined below.
- All occurrences of pick are replaced by specific steps to compute the desired values.
- It may implement a subset of TIP.

The definition of process TIP is as follows. It has two parameters: the name of the tool and the (unique) external identification of this tool instance.

```
def-TIP =                                                          [def-TIP-1]
 process TIP (ToolName : str, ExtToolId : int)
 is let TidInTB : term, PendingEvents : list, Term : term, Feat : term
   in (rec-execute(ToolName, Feat ?, TidInTB)
       . snd-connect(ToolName, ExtToolId, TidInTB)
        + snd-connect(ToolName, ExtToolId))
      . PendingEvents := []
       . (TIP-EVAL-DO(ExtToolId)
          + TIP-EVENTS(ExtToolId, PendingEvents ?)
          + TIP-MONITOR(ExtToolId)
          + TIP-RECONFIGURE(ExtToolId))
         * (snd-disconnect(ExtToolId)
            + rec-terminate(ExtToolId, Term ?)) endlet
```

The handling of eval/do requests by a tool instance consists of two alternatives:

- Receive a do request (from the Tool Control Process): perform the desired operation.
- Receive an eval request (from the Tool Control Process): perform the desired operation and send the resulting value back to the Tool Control Process. This may be interrupted by a cancel or terminate request (from the Tool Control Process).

```
def-TIP-EVAL-DO =                                              [def-TIP-EVAL-DO-1]
 process TIP-EVAL-DO (ExtToolId : int)
 is let Term : term, AnyVal : term
   in rec-do(ExtToolId, Term ?)
      + rec-eval(ExtToolId, Term ?)
        . (pick(AnyVal ?)
          . snd-value(ExtToolId, AnyVal)
           + rec-cancel(ExtToolId)
           + rec-terminate(ExtToolId)
             . delta) endlet
```

The handling of events produced by a tool instance consists of two steps:

- Generate an event and send it to the Tool Control Process.
- Receive an ack-event (from the Tool Control Process) that indicates that the handling of a previous event has been completed.

In both cases, the second argument of snd-event/rec-ack-event identifies the event in question.

```
def-TIP-EVENTS =                                               [def-TIP-EVENTS-1]
 process TIP-EVENTS (ExtToolId : int, PendingEvents : list ?)
 is let Term : term, AnyTerm : Term, AnyTerms : list
   in pick(AnyTerm ?, AnyTerms ?)
     . if not(is-element(AnyTerm, PendingEvents))
      then snd-event(ExtToolId, AnyTerm, AnyTerms)
           . PendingEvents := join(PendingEvents, [AnyTerm])
      else tau fi
      + rec-ack-event(ExtToolId, Term ?)
        . if is-element(Term, PendingEvents)
         then PendingEvents := diff(PendingEvents, [Term]) fi endlet
```

A tool instance may first attach itself as monitor to a selection of processes in the ToolBus and, later on, it may detach itself as monitor from a (possibly different) selection of processes.

```
def-TIP-MONITOR =                                              [def-TIP-MONITOR-1]
 process TIP-MONITOR (ExtToolId : int)
 is let AnySel : list, AnyKind : term
   in pick(AnySel ?, AnyKind ?) . snd-attach-monitor(ExtToolId, AnyKind, AnySel)
      + pick(AnySel ?) . snd-detach-monitor(ExtToolId, AnySel) endlet
```

A tool instance may generate a reconfigure request, that will effectively load a new **T** script into the current ToolBus.

```
def-TIP-RECONFIGURE =                                          [def-TIP-RECONFIGURE-1]
 process TIP-RECONFIGURE (ExtToolId : int)
 is let AnyScript : str, AnySel : list
   in pick(AnyScript ?, AnySel ?) . snd-reconfigure(ExtToolId, AnyScript, AnySel) endlet
```

### 7.2.2 The tool control process

Viewed from the TOOLBUS, a *tool control process* will enforce the behaviour of a tool. It is in many respects the complement of the tool instance process given above.

```
Module ToolControlProcess
imports  Execute-Terminate(7.4) Connect-Disconnect(7.3) Eval-Do(7.5) Events(7.6)
         Messages(6.4) Iteration(6.1) Conditionals(6.6) Notes(6.8) Monitoring(7.7)
         Reconfigure(7.8) Let(6.3)
exports
  context-free syntax
    def-TCP         → DEF
    def-EVENTS      → DEF
    def-EVAL-DO     → DEF
    def-LOGGER      → DEF
    def-VIEWER      → DEF
    def-CONTROLLER  → DEF
    def-MONITOR     → DEF
    def-TERM        → DEF
    def-END-TOOL    → DEF
equations
```

Each instance of the tool control process TCP controls the behaviour of one tool instance. It has one parameter: the identity of the tool instance (both in the external world and inside the TOOLBUS). The overall behaviour of a tool control process is an infinite loop in which the following alternatives are handled:

- Receive events/acknowledgements (see process `EVENTS`, below).
- Handle eval/do communication (see process `EVAL-DO`, below).
- Handle monitoring communication (see process `MONITOR`, below).

The loop ends when the execution of the tool instance terminates (see process `END-TOOL`, below)

```
def-TCP =                                                          [def-TCP-1]
 process TCP (ToolId : term, ExtToolId : int)
 is let PendingEvents : list, PendingViews : list, PendingControls : list
   in control-tool-by-TCP(ToolId, ExtToolId)
      . PendingEvents := []
       . PendingViews := []
        . PendingControls := []
         . (EVENTS(ToolId, ExtToolId, PendingEvents ?)
            + EVAL-DO(ToolId, ExtToolId)
            + MONITOR(ToolId, ExtToolId, PendingViews ?, PendingControls ?))
           * END-TOOL(ToolId, ExtToolId) endlet
```

Handling events consists of the following two alternatives:

- Receive an event from the tool instance. The event is allowed if its second argument (a term) does not yet occur in `PendingEvents`. Add that term to `PendingEvents` and forward the event to a TOOLBUS process.
- Receive an acknowledgement of a previous event from a TOOLBUS process. This is only allowed if the second argument of the acknowledgement (a term) does occur in `PendingEvents`. If so, delete it from `PendingEvents` and forward the acknowledgement to the tool instance.

```
def-EVENTS =                                                        [def-EVENTS-1]
 process EVENTS (ToolId : term, ExtToolId : int, PendingEvents : list ?)
 is let Term : term, Terms : list
   in rec-event-by-TCP(ExtToolId, Term ?, Terms ?)
      . if not(is-element(Term, PendingEvents))
       then PendingEvents := join(PendingEvents, [Term])
            . (snd-event-by-TCP(ToolId, Term, Terms)  + TERM(ToolId)) fi
     + rec-ack-event-by-TCP(ToolId, Term ?)
       . if is-element(Term, PendingEvents)
         then PendingEvents := diff(PendingEvents, [Term])
              . snd-ack-event-by-TCP(ExtToolId, Term) fi endlet
```

Handling eval/do requests consists of the following two alternatives:

- Receive a do request (from a ToolBus process): forward it to the tool.
- Receive an eval request (from a ToolBus process): forward it to the tool. It may be followed by a value produced by the tool or a cancel request from a process.

```
def-EVAL-DO =                                                       [def-EVAL-DO-1]
 process EVAL-DO (ToolId : term, ExtToolId : int)
 is let Term : term, Val : term
   in rec-do-by-TCP(ToolId, Term ?) . snd-do-by-TCP(ExtToolId, Term)
     + rec-eval-by-TCP(ToolId, Term ?)
       . snd-eval-by-TCP(ExtToolId, Term)
        . (rec-value-by-TCP(ExtToolId, Val ?)
          . (snd-value-by-TCP(ToolId, Val)  + TERM(ToolId, ExtToolId))
           + rec-cancel-by-TCP(ToolId) . snd-cancel-by-TCP(ExtToolId)) endlet
```

Handling of monitoring communication consists of the following alternatives:

- Communication related to loggers.
- Communication related to viewers.
- Communication related to controllers.
- Process termination messages to monitors.
- Attach or detach monitors.
- Reconfiguration requests.

```
def-MONITOR =                                                       [def-MONITOR-1]
 process MONITOR (ToolId : term, ExtToolId : int, PendingViews : list ?, PendingControls : list ?)
 is let ProcId : int, Sel : list, TScript : str, Kind : term
   in LOGGER(ToolId, ExtToolId)
     + VIEWER(ToolId, ExToolId, PendingViews ?)
     + CONTROLLER(ToolId, ExToolId, PendingControls ?)
     + rec-monitor(ToolId, terminates(ProcId ?))
       . snd-do(ExtToolId, terminates(ProcId))
     + rec-attach-monitor-by-TCP(ExtToolId, Kind ?, Sel ?)
       . (attach-monitor(ToolId, Kind, Sel)  + TERM(ToolId, ExtToolId))
     + rec-detach-monitor-by-TCP(ExtToolId, Sel ?)
       . (detach-monitor(Sel)  + TERM(ToolId, ExtToolId))
     + rec-reconfigure-by-TCP(ExtToolId, TScript ?, Sel ?)
       . (reconfigure(TScript, Sel)  + TERM(ToolId, ExtToolId)) endlet
```

The following three processes LOGGER, VIEWER, and CONTROLLER describe the communication with the three kinds of monitors.

```
def-LOGGER =                                                                    [def-LOGGER-1]
 process LOGGER (Toolid : term, ExtToolId : int)
 is let PID1 : term, AF : term, Args : list, EV : list, NL : list, PID2 : term, Proc : term
   in rec-monitor(ToolId, logpoint(PID1 ?, AF ?, Args ?, EV ?, NL ?, PID2 ?, Proc ?))
      . snd-do-by-TCP(ExtToolId, logpoint(PID1, AF, Args, EV, NL, PID2, Proc)) endlet
```

```
def-VIEWER =                                                                    [def-VIEWER-1]
 process VIEWER (Toolid : term, ExtToolId : int, PendingViews : list ?)
 is let PID1 : int, AF : term, Args : list, EV : list, NL : list, PID2 : int, Proc : term
   in rec-monitor(ToolId, viewpoint(PID1 ?, AF ?, Args ?, EV ?, NL ?, PID2 ?, Proc ?))
      . if not(is-element(PID1, PendingViews))
        then PendingViews := join(PendingViews, [PID1])
             . snd-do-by-TCP(ExtToolId, viewpoint(PID1, AF, Args, EV, NL, PID2, Proc)) fi
     + rec-event-by-TCP(ExtToolId, continue(PID1 ?))
       . if is-element(PID1, PendingViews)
         then PendingViews := diff(PendingViews, [PID1])
              . snd-monitor(ToolId, continue(PID1))
                . snd-ack-event-by-TCP(ExtToolId, continue(PID1)) fi endlet
```

```
def-CONTROLLER =                                                                [def-CONTROLLER-1]
 process CONTROLLER (Toolid : term, ExtToolId : int, PendingControls : list ?)
 is let PID1 : int, AF : term, Args : list, EV : list, NL : list, PID2 : int,
       Proc : term
   in rec-monitor(ToolId, controlpoint(PID1 ?, AF ?, Args ?, EV ?, NL ?, PID2 ?, Proc ?))
      . if not(is-element(PID1, PendingControls))
        then PendingControls := join(PendingControls, [PID1])
             . snd-do-by-TCP(ExtToolId, controlpoint(PID1, AF, Args, EV, NL, PID2, Proc)) fi
     + rec-event-by-TCP(ExtToolId, continue(PID1 ?, Proc ?, EV ?))
       . if is-element(PID1, PendingControls)
         then PendingControls := diff(PendingControls, [PID1])
              . snd-monitor(ToolId, continue(PID1, Proc, EV))
                . snd-ack-event-by-TCP(ExtToolId, continue(PID1, Proc, EV)) fi endlet
```

The execution of a tool instance ends when:

- A disconnection request is generated by the tool instance.
- A termination message is sent by a TOOLBUS process.

```
def-END-TOOL =                                                                  [def-END-TOOL-1]
 process END-TOOL (ToolId : term, ExtToolId : int)
 is let Term : term
   in (rec-disconnect-by-TCP(ExtToolId) . snd-disconnect-by-TCP(ToolId)
       + rec-terminate-by-TCP(ToolId, Term ?) . snd-terminate-by-TCP(ExtToolId, Term)) . delta endlet
```

The explicit termination of this tool instance by a process appears as explicit alternative in several of the above scripts. The termination request is forwarded to the tool and the Tool Control Process as a whole becomes inactive.

```
def-TERM =                                                                      [def-TERM-1]
 process TERM (ToolId : term, ExtToolId : int)
 is rec-terminate-by-TCP(ToolId) . snd-terminate-by-TCP(ExtToolId) . delta
```

Extend a ToolBus script with all definitions needed by the tool control process.

add-TCP-defs(*Defs* toolbus(*ProcAppls*)) = **[add-TCP-defs-1]**
*Defs* def-TCP def-EVENTS def-EVAL-DO def-LOGGER def-VIEWER def-CONTROLLER def-MONITOR def-TERM def-END-TOOL toolbus(*ProcAppls*)

## 7.3 Connection and disconnection of tools

The following primitives handle the connection and disconnection of tools:

- `rec-connect` assigns a new tool identifier to its second argument (a result variable) that is needed in all subsequent operations addressing this tool instance.
- `rec-disconnect` receives a disconnection request from a tool instance.

| ToolBus Process | Tool Control Process | | Tool |
|---|---|---|---|
| rec-connect | snd-connect-by-TCP | rec-connect-before-TCP | snd-connect |
| new-tool-id† | | | |
| | control-tool-by-TCP | | |
| rec-disconnect | snd-disconnect-by-TCP | rec-disconnect-by-TCP | snd-disconnect |

**Module** Connect-Disconnect
**imports** ToolBus$^{(5.5)}$ ToolDefs$^{(7.1)}$ Expressions$^{(6.5)}$ Create$^{(6.7)}$
**exports**
**lexical syntax**

| | |
|---|---|
| snd-connect | → ATOMIC-FUN |
| snd-disconnect | → ATOMIC-FUN |
| rec-connect-before-TCP | → ATOMIC-FUN |
| snd-connect-by-TCP | → ATOMIC-FUN |
| rec-disconnect-by-TCP | → ATOMIC-FUN |
| snd-disconnect-by-TCP | → ATOMIC-FUN |
| rec-disconnect | → ATOMIC-FUN |
| new-tool-id | → ATOMIC-FUN |
| control-tool-by-TCP | → ATOMIC-FUN |

**context-free syntax**
rec-connect(TERM-LIST) → PROC
**equations**
Define relevant communications.

$\gamma_1$(`snd-connect`, `rec-connect-before-TCP`) = true **[cm-connect]**

$\gamma_1$(`snd-disconnect`, `rec-disconnect-by-TCP`) = true **[cm-disconnect]**

$\gamma_1$(`snd-disconnect-by-TCP`, `rec-disconnect`) = true **[cm-disconnect-TCP]**

Extend the preparation of processes (Section 5.2) for rec-connect.

prep-proc(rec-connect(*Ts*), *Pnm*, *Env*) = rec-connect(prep-term-list(*Ts*, *Pnm*, *Env*)) **[prep-rec-connect]**

Extend the expansion of process expressions (Section 5.4) for rec-connect. There are two cases to be distinguished here:

- The initiative for the connection is taken by the tool. In that case, a new tool identifier has to be generated for this new tool instance.

- The connection is the result of a previous execute action. In that case, the tool identifier has already been generated at the moment that the execute action was performed.

In the first case, the rec-connect atom is expanded into the sequence rec-connect-before-TCP, assignment of a new tool identification to variable Var in the current process, and creation of a new tool control process for the tool just connected. The new tool control process is instantiated with both the internal (i.e., inside the ToolBus) and the external identification of this tool instance.

$$\frac{\begin{array}{c} T = \mathit{Var}\,?, \\ \text{outermost-type-of}(\mathit{Var}) = \mathit{Id}, \\ \text{tool-definition}(\mathit{Id}, \text{get-script}(C)) = \text{tool } \mathit{Id}\ \mathit{Formals} \text{ is } \{\mathit{Features}\}, \\ \text{require-type}(\mathit{Id}, \mathit{Var}) = \text{true} \end{array}}{\begin{array}{l} \text{expand}(\text{rec-connect}(T), C) = \\ \texttt{rec-connect-before-TCP}(\mathit{Id}(), \texttt{ExtToolId} : \texttt{int}\ ?) \\ .\ \texttt{new-tool-id}(\mathit{Id}, \mathit{Var}\,?)\ .\ \text{create}(\texttt{TCP}(\mathit{Var}, \texttt{ExtToolId} : \texttt{int}), \texttt{Cid} : \texttt{int}\ ?) \end{array}} \quad \textbf{[exp-rec-connect-1]}$$

In the second case, the rec-connect atom is expanded into the sequence rec-connect-before-TCP, and the creation of a new tool control process.

$$\frac{\mathit{Id}(\mathit{Int}) = \text{substitute}(T, \text{get-env}(C))}{\begin{array}{l} \text{expand}(\text{rec-connect}(T), C) = \\ \texttt{rec-connect-before-TCP}(\mathit{Id}(), \texttt{ExtToolId} : \texttt{int}\ ?, \mathit{Id}(\mathit{Int})) \\ .\ \text{create}(\texttt{TCP}(\mathit{Id}(\mathit{Int}), \texttt{ExtToolId} : \texttt{int}), \texttt{Cid} : \texttt{int}\ ?) \end{array}} \quad \textbf{[exp-rec-connect-2]}$$

The auxiliary atom new-tool-id computes a new tool identification:

$$\frac{\begin{array}{c} \text{BS} = \text{get-bus-state}(C), \\ \text{get-new-tool-id}(\mathit{Id}, \text{BS}) = \mathit{Id}(\mathit{Int}), \\ \mathit{Env}' = \text{assign}(\mathit{Var}, \mathit{Id}(\mathit{Int}), \text{get-env}(C)), \\ \text{BS}' = \text{BS}[\texttt{new-tool-id} := \texttt{tool-id}(\mathit{Int} + 1)] \end{array}}{\text{simple-atomic-step}(\texttt{new-tool-id}(\mathit{Id}, \mathit{Var}\,?), C) = C \,/\, \mathit{Env}' \,/\, \text{BS}'} \quad \textbf{[new-tool-id]}$$

The auxiliary atom control-tool is used in the definition of TCP (see Section 7.2.2) to establish a "control-tool" relation between each tool control process and the tool it controls. This information is used in the definitions of shutdown and reconfigure (see Section 7.8).

$$\frac{\begin{array}{c} \text{BS} = \text{get-bus-state}(C),\ \ \mathit{Pid} = \text{get-proc-id}(C),\ \ \mathit{Env} = \text{get-env}(C), \\ \text{BS}' = \text{BS}[\texttt{control-tool}(\mathit{Pid}) := \text{substitute}([T_1, T_2], \mathit{Env})] \end{array}}{\text{simple-atomic-step}(\texttt{control-tool-by-TCP}(T_1, T_2), C) = C \,/\, \text{BS}'} \quad \textbf{[control-tool]}$$

## 7.4 Execution and termination of tools

Execution and termination of tools is achieved by the following primitives:

- `execute`: start the execution of a tool.
- `snd-terminate`: terminate the execution of a tool.

| ToolBus Process | Tool Control Process | | Tool |
|---|---|---|---|
| execute | | | |
| snd-execute-to-tool† | | | rec-execute |
| snd-terminate | rec-terminate-by-TCP | snd-terminate-by-TCP | rec-terminate |

**Module** Execute-Terminate
**imports** ToolBus[(5.5)] ToolDefs[(7.1)] Connect-Disconnect[(7.3)]
**exports**
**lexical syntax**

|  |  |
|---|---|
| snd-terminate | $\rightarrow$ ATOMIC-FUN |
| snd-execute-to-tool | $\rightarrow$ ATOMIC-FUN |
| rec-terminate-by-TCP | $\rightarrow$ ATOMIC-FUN |
| snd-terminate-by-TCP | $\rightarrow$ ATOMIC-FUN |
| rec-execute | $\rightarrow$ ATOMIC-FUN |
| rec-terminate | $\rightarrow$ ATOMIC-FUN |

**context-free syntax**
execute(TERM-LIST) $\rightarrow$ PROC
**equations**
Define relevant communications.

$$\gamma_1(\texttt{snd-execute-to-tool}, \texttt{rec-execute}) = \text{true} \quad \text{[cm-execute]}$$

$$\gamma_1(\texttt{snd-terminate}, \texttt{rec-terminate-by-TCP}) = \text{true} \quad \text{[cm-terminate]}$$

$$\gamma_1(\texttt{snd-terminate-by-TCP}, \texttt{rec-terminate}) = \text{true} \quad \text{[cm-terminate-TCP]}$$

$$\text{proc2term}(\text{execute}(\mathit{Ts})) = \text{exec}(\mathit{Ts}) \quad \text{[p2t-execute]}$$

$$\text{term2proc}(\text{exec}(\mathit{Ts})) = \text{execute}(\mathit{Ts}) \quad \text{[t2p-execute]}$$

Extend the preparation of processes (Section 5.2) for execute.

$$\text{prep-proc}(\text{execute}(\mathit{Ts}), \mathit{Pnm}, \mathit{Env}) = \text{execute}(\text{prep-term-list}(\mathit{Ts}, \mathit{Pnm}, \mathit{Env})) \quad \text{[prep-execute]}$$

Extend the expansion of process expressions (Section 5.4) for execute. Tool execution is treated by replacing the execute atom by new-tool-id, snd-execute-to-tool followed by rec-connect. The features defining the tool are first retrieved from its tool definition and formal parameters appearing in the features are replaced by their actual value. Next, the features are converted into a list of terms which contains all information needed to execute the tool. When the tool is successfully executing, it will send a request to connect containing the tool name and the newly generated tool identifier. This will uniquely identify the connection request with the execution command just executed. Note that concurrently performed execute command can be distinguished in this manner.

$$\frac{\begin{array}{c}\text{outermost-type-of}(\mathit{Var}) = \mathit{Id},\\ \text{tool-definition}(\mathit{Id}, \text{get-script}(C)) = \text{tool } \mathit{Id}\ (\mathit{GenVars}) \text{ is } \{\mathit{Features}\},\\ \mathit{Env} = \text{get-env}(C),\\ [\mathit{OptTs}'] = \text{substitute}([\mathit{OptTs}], \mathit{Env}),\\ T = \text{features2term}(\{\mathit{Features}\}, \text{create-env}(\mathit{GenVars}, \texttt{TOOL}, [\mathit{OptTs}'], \mathit{Env}))\end{array}}{\begin{array}{l}\text{expand}(\text{execute}(\mathit{Id}(\mathit{OptTs}), \mathit{Var}\,?), C) =\\ \texttt{new-tool-id}(\mathit{Id}, \mathit{Var}\,?)\ .\ \texttt{snd-execute-to-tool}(\mathit{Id}(), T, \mathit{Var})\ .\ \text{rec-connect}(\mathit{Var})\end{array}} \quad \text{[exp-execute]}$$

## 7.5 Evaluation and do requests to tools

The following primitives send a request to a tool:

- `snd-eval`: request a tool to evaluate a term. The first argument serves as the identification of the tool (as produced by rec-connect or execute), while the second one is the term to be evaluated.
- `rec-value`: receive from a tool the result of a previous evaluation request.
- `snd-cancel`: cancel a previous `snd-eval`.

- `snd-do`: request a tool to evaluate a term and ignore the resulting value.

| TOOLBUS Process | Tool Control Process | | Tool |
|---|---|---|---|
| snd-eval | rec-eval-by-TCP | snd-eval-by-TCP | rec-eval |
| rec-value | snd-value-by-TCP | rec-value-by-TCP | snd-value |
| snd-cancel | rec-cancel-by-TCP | snd-cancel-by-TCP | rec-cancel |
| snd-do | rec-do-by-TCP | snd-do-by-TCP | rec-do |

**Module** Eval-Do
**imports** ToolBus$^{(5.5)}$
**exports**
**lexical syntax**

| | |
|---|---|
| snd-eval | → ATOMIC-FUN |
| rec-value | → ATOMIC-FUN |
| snd-do | → ATOMIC-FUN |
| rec-eval-by-TCP | → ATOMIC-FUN |
| snd-value-by-TCP | → ATOMIC-FUN |
| rec-do | → ATOMIC-FUN |
| rec-eval | → ATOMIC-FUN |
| snd-eval-by-TCP | → ATOMIC-FUN |
| rec-value-by-TCP | → ATOMIC-FUN |
| snd-value | → ATOMIC-FUN |
| rec-do-by-TCP | → ATOMIC-FUN |
| snd-do-by-TCP | → ATOMIC-FUN |
| snd-cancel | → ATOMIC-FUN |
| rec-cancel-by-TCP | → ATOMIC-FUN |
| snd-cancel-by-TCP | → ATOMIC-FUN |
| rec-cancel | → ATOMIC-FUN |

**equations**
Define relevant communications.

$$\gamma_1(\texttt{snd-eval}, \texttt{rec-eval-by-TCP}) = \mathsf{true} \quad \text{[cm-eval]}$$
$$\gamma_1(\texttt{snd-eval-by-TCP}, \texttt{rec-eval}) = \mathsf{true} \quad \text{[cm-eval-TCP]}$$
$$\gamma_1(\texttt{snd-value}, \texttt{rec-value-by-TCP}) = \mathsf{true} \quad \text{[cm-value]}$$
$$\gamma_1(\texttt{snd-value-by-TCP}, \texttt{rec-value}) = \mathsf{true} \quad \text{[cm-value-TCP]}$$
$$\gamma_1(\texttt{snd-do}, \texttt{rec-do-by-TCP}) = \mathsf{true} \quad \text{[cm-do]}$$
$$\gamma_1(\texttt{snd-do-by-TCP}, \texttt{rec-do}) = \mathsf{true} \quad \text{[cm-do-TCP]}$$
$$\gamma_1(\texttt{snd-cancel}, \texttt{rec-cancel-by-TCP}) = \mathsf{true} \quad \text{[cm-cancel]}$$
$$\gamma_1(\texttt{snd-cancel-by-TCP}, \texttt{rec-cancel}) = \mathsf{true} \quad \text{[cm-TCP]}$$

## 7.6 Events produced by tools

Tools can also take the initiative by sending an "event" to the TOOLBUS which can be handled by a TOOLBUS process using the following primitives:

- `rec-event`: receive an event from a tool. The first argument of `rec-event` is a tool identification. The second argument serves as an identification of the source of the event. The remaining, optional, arguments give the details of the event in question.

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

| TOOLBUS Process | Tool Control Process | | Tool |
|---|---|---|---|
| rec-event<br>snd-ack-event | snd-event-by-TCP<br>rec-ack-event-by-TCP | rec-event-by-TCP<br>snd-ack-event-by-TCP | snd-event<br>rec-ack-event |

**Module** Events
**imports** BusState$^{(5.3.2)}$
**exports**
**lexical syntax**

```
snd-event            → ATOMIC-FUN
rec-event-by-TCP     → ATOMIC-FUN
snd-event-by-TCP     → ATOMIC-FUN
rec-event            → ATOMIC-FUN
snd-ack-event        → ATOMIC-FUN
rec-ack-event-by-TCP → ATOMIC-FUN
snd-ack-event-by-TCP → ATOMIC-FUN
rec-ack-event        → ATOMIC-FUN
```

**equations**
Define relevant communications.

$\gamma_1$(`snd-event`, `rec-event-by-TCP`) = true **[cm-event]**

$\gamma_1$(`snd-event-by-TCP`, `rec-event`) = true **[cm-event-TCP]**

$\gamma_1$(`snd-ack-event`, `rec-ack-event-by-TCP`) = true **[cm-ack-event]**

$\gamma_1$(`snd-ack-event-by-TCP`, `rec-ack-event`) = true **[cm-ack-event-TCP]**

## 7.7 Monitoring of TOOLBUS processes by tools

Monitoring of **T** scripts is possible with the following primitives:

- `attach-monitor`: attach a monitoring tool to a process. Note that *any* tool can act as monitor for any TOOLBUS process. As a result, information will be sent to the tool allowing the detailed monitoring of the execution of the process.
- `detach-monitor`: detach a monitoring tool from a process.

| TOOLBUS Process | Tool Control Process | Tool |
|---|---|---|
| attach-monitor<br>detach-monitor<br>snd-monitor†<br>rec-monitor†<br>continuation† | rec-attach-monitor-by-TCP<br>rec-detach-monitor-by-TCP | snd-attach-monitor<br>snd-detach-monitor |

**Module** Monitoring
**imports** ToolBus$^{(5.5)}$ Messages$^{(6.4)}$ Notes$^{(6.8)}$ ToolDefs$^{(7.1)}$
**exports**
**sorts** MONITOR

**lexical syntax**

```
  attach-monitor              → ATOMIC-FUN
  detach-monitor              → ATOMIC-FUN
  rec-attach-monitor-by-TCP   → ATOMIC-FUN
  rec-detach-monitor-by-TCP   → ATOMIC-FUN
  snd-attach-monitor          → ATOMIC-FUN
  snd-detach-monitor          → ATOMIC-FUN
  snd-monitor                 → ATOMIC-FUN
  rec-monitor                 → ATOMIC-FUN
  continuation                → ATOMIC-FUN
```

**context-free syntax**

```
  ID "(" TOOL-ID ")"                                                → MONITOR
  none                                                              → MONITOR
  MONITOR                                                           → BUS-VAL

  get-monitor(BUS-STATE, PROC-ID)                                   → MONITOR
  has-monitor-attached(BUS-STATE, PROC-ID)                          → BOOL
  is-monitor-atom(ATOMIC-FUN)                                       → BOOL
  selects(TERM, PROC-ID, BUS-STATE)                                 → BOOL
  select-and-set-monitor(TERM, PROCESSES, MONITOR, BUS-STATE)       → BUS-STATE

  mk-logpoint(PROC-ID, ATOM, ENV, BUS-STATE, PROC-ID)               → TERM
  mk-viewpoint(PROC-ID, ATOM, ENV, BUS-STATE, PROC-ID)              → TERM
  mk-controlpoint(PROC-ID, ATOM, ENV, BUS-STATE, PROC-ID, PROC)     → TERM
  monitor-info(MONITOR, ATOM, CONTEXT, PROC-ID)                     → NEXT-INFO
```

**exports**

**variables**

```
  Mid [0-9']*   → TOOL-ID
  MonitorKind   → ID
  Monitor       → MONITOR
```

**hiddens**

**variables**

```
  Break [0-9']* → TERM
  Chars [0-9']* → CHAR*
  Sel           → TERM
```

**equations**

Define relevant communications.

$\gamma_1$(`snd-attach-monitor`, `rec-attach-monitor-by-TCP`) = true [cm-attach-monitor]

$\gamma_1$(`snd-detach-monitor`, `rec-detach-monitor-by-TCP`) = true [cm-detach-monitor]

$\gamma_1$(`snd-monitor`, `rec-monitor`) = true [cm-monitor]

Add a field `monitor` to each process representation. Its value is a term of the form $MonitorKind(Tid)$, where $MonitorKind$ if logger, viewer, or controller, and $Tid$ is the tool identification of the monitoring tool attached to the process. The value of the `monitor` field is none if no monitoring tool is attached). Also define the predicate has-monitor-attached to test whether a process is being monitored or not.

BS . `monitor`(*Pid*) = no-bus-val ⇒ get-monitor(BS, *Pid*) = none [get-monitor-1]

BS . `monitor`(*Pid*) = *MonitorKind*(*Tid*) ⇒ get-monitor(BS, *Pid*) = *MonitorKind*(*Tid*) [get-monitor-2]

BS . `monitor`(*Pid*) = none ⇒ get-monitor(BS, *Pid*) = none [get-monitor-3]

get-monitor(BS, *Pid*) = none ⇒ has-monitor-attached(BS, *Pid*) = false [hma-1]

has-monitor-attached(BS, *Pid*) = true **otherwise** [hma-2]

Special care is needed to exclude from monitoring the exchange of monitoring information between monitored process and the tool control process of its attached monitoring tool. The atoms snd-monitor and rec-monitor are equivalent to snd-msg and rec-msg except that no information will be sent to any monitor when they are executed. The atom continuation is executed by a process being monitored by a controller.

$$\text{is-monitor-atom}(\texttt{snd-monitor}) = \text{true} \quad \textbf{[is-monitor-atom-1]}$$

$$\text{is-monitor-atom}(\texttt{rec-monitor}) = \text{true} \quad \textbf{[is-monitor-atom-2]}$$

$$\text{is-monitor-atom}(\texttt{continuation}) = \text{true} \quad \textbf{[is-monitor-atom-3]}$$

$$\text{is-monitor-atom}(\mathit{AtomicFun}) = \text{false} \quad \textbf{otherwise} \quad \textbf{[is-monitor-atom-4]}$$

Attaching and detaching monitors to/from processes requires a selection mechanism for processes. We introduce a selects predicate that given a "selection criterion" (a term-list) and a process representation yields true when either the name or the process identifier of the process occur in the given term-list.

$$\frac{\text{get-proc-name}(\text{BS}, \mathit{Pid}) = \texttt{proc-name}(\mathit{String})}{\text{selects}([\texttt{proc-name}(\mathit{String}), \mathit{OptTs}], \mathit{Pid}, \text{BS}) = \text{true}} \quad \textbf{[sel-1]}$$

$$\text{selects}([\mathit{Pid}, \mathit{OptTs}], \mathit{Pid}, \text{BS}) = \text{true} \quad \textbf{[sel-2]}$$

$$\text{selects}([], \mathit{Pid}, \text{BS}) = \text{false} \quad \textbf{[sel-3]}$$

$$\text{selects}([T, \mathit{OptTs}], \mathit{Pid}, \text{BS}) = \text{selects}([\mathit{OptTs}], \mathit{Pid}, \text{BS}) \quad \textbf{otherwise} \quad \textbf{[sel-4]}$$

Set the monitor of selected processes. Observe that we explicitly forbid attachment of a monitor to a process that has already a monitor attached, respectively, removing a monitor from a process that has no monitor attached. We also forbid attaching a monitor to either a tool or a tool control process.

$$\frac{\begin{array}{c}\mathit{Pid} = \text{get-pid}(\text{PR}),\ \text{selects}(\mathit{Sel}, \mathit{Pid}, \text{BS}) = \text{true},\\ \mathit{Monitor} \neq \text{none},\\ \text{has-monitor-attached}(\text{BS}, \mathit{Pid}) = \text{false},\\ \text{get-proc-name}(\text{BS}, \mathit{Pid}) \neq \texttt{proc-name("TCP")},\ \text{get-proc-name}(\text{BS}, \mathit{Pid}) \neq \texttt{"TOOL"},\\ \text{BS}' = \text{BS}[\texttt{monitor}(\mathit{Pid}) := \mathit{Monitor}]\end{array}}{\begin{array}{c}\text{select-and-set-monitor}(\mathit{Sel}, \{\text{PR} \mid\mid \mathit{PRs}\}, \mathit{Monitor}, \text{BS}) =\\ \text{select-and-set-monitor}(\mathit{Sel}, \{\mathit{PRs}\}, \mathit{Monitor}, \text{BS}')\end{array}} \quad \textbf{[sel-and-set-1]}$$

$$\frac{\begin{array}{c}\mathit{Pid} = \text{get-pid}(\text{PR}),\ \text{selects}(\mathit{Sel}, \mathit{Pid}, \text{BS}) = \text{true},\\ \text{has-monitor-attached}(\text{BS}, \mathit{Pid}) = \text{true},\\ \text{get-proc-name}(\text{BS}, \mathit{Pid}) \neq \texttt{proc-name("TCP")},\ \text{get-proc-name}(\text{BS}, \mathit{Pid}) \neq \texttt{"TOOL"},\\ \text{BS}' = \text{BS}[\texttt{monitor}(\mathit{Pid}) := \text{none}]\end{array}}{\begin{array}{c}\text{select-and-set-monitor}(\mathit{Sel}, \{\text{PR} \mid\mid \mathit{PRs}\}, \text{none}, \text{BS}) =\\ \text{select-and-set-monitor}(\mathit{Sel}, \{\mathit{PRs}\}, \text{none}, \text{BS}')\end{array}} \quad \textbf{[sel-and-set-2]}$$

$$\text{select-and-set-monitor}(\mathit{Sel}, \{\}, \mathit{Monitor}, \text{BS}) = \text{BS} \quad \textbf{[sel-and-set-3]}$$

$$\begin{array}{l}\text{select-and-set-monitor}(\mathit{Sel}, \{\text{PR} \mid\mid \mathit{PRs}\}, \mathit{Monitor}, \text{BS}) =\\ \text{select-and-set-monitor}(\mathit{Sel}, \{\mathit{PRs}\}, \mathit{Monitor}, \text{BS}) \quad \textbf{otherwise}\end{array} \quad \textbf{[sel-and-set-4]}$$

The interface with a monitoring tool strictly follows the standard data formats for tools. As a consequence, some notions (process expressions, environments) have to be converted to terms. We use the conversion functions env2term and proc2term for this purpose. The functions mk-logpoint, mk-viewpoint and mk-controlpoint construct a term containing all necessary information to completely characterize the execution of one atom in a process for each kind of monitoring. Note that the format of each monitoring point is identical, but that note all components are used, i.e., the last component containing a process expression is only used for control points.

$$\frac{\text{get-notes}(\mathrm{BS}, \mathit{Pid}_1) = \text{note-list}(\mathit{Notes})}{\begin{array}{l}\text{mk-logpoint}(\mathit{Pid}_1, \mathit{Atom}, \mathit{Env}, \mathrm{BS}, \mathit{Pid}_2) = \\ \texttt{logpoint}(\mathit{Pid}_1, \text{atomic-fun2string}(\text{fun}(\mathit{Atom})), \text{args}(\mathit{Atom}), \\ \qquad \text{env2term}(\mathit{Env}), [\mathit{Notes}], \mathit{Pid}_2, \texttt{"delta"})\end{array}} \qquad \textbf{[mk-logpoint-1]}$$

$$\frac{\text{get-notes}(\mathrm{BS}, \mathit{Pid}_1) = \text{note-list}(\mathit{Notes})}{\begin{array}{l}\text{mk-viewpoint}(\mathit{Pid}_1, \mathit{Atom}, \mathit{Env}, \mathrm{BS}, \mathit{Pid}_2) = \\ \texttt{viewpoint}(\mathit{Pid}_1, \text{atomic-fun2string}(\text{fun}(\mathit{Atom})), \text{args}(\mathit{Atom}), \\ \qquad \text{env2term}(\mathit{Env}), [\mathit{Notes}], \mathit{Pid}_2, \texttt{"delta"})\end{array}} \qquad \textbf{[mk-viewpoint-1]}$$

$$\frac{\text{get-notes}(\mathrm{BS}, \mathit{Pid}_1) = \text{note-list}(\mathit{Notes})}{\begin{array}{l}\text{mk-controlpoint}(\mathit{Pid}_1, \mathit{Atom}, \mathit{Env}, \mathrm{BS}, \mathit{Pid}_2, P) = \\ \texttt{controlpoint}(\mathit{Pid}_1, \text{atomic-fun2string}(\text{fun}(\mathit{Atom})), \text{args}(\mathit{Atom}), \\ \qquad \text{env2term}(\mathit{Env}), [\mathit{Notes}], \mathit{Pid}_2, \text{proc2term}(P))\end{array}} \qquad \textbf{[mk-controlpoint-1]}$$

After these preparations, define two functions for executing a step in a process. When a process is being monitored, the following equation transforms the normal "next" information into additional monitoring information.

$$\frac{\begin{array}{c}\mathrm{BS} = \text{get-bus-state}(C), \\ \mathit{Pid}_1 = \text{get-proc-id}(C), \\ \text{has-monitor-attached}(\mathrm{BS}, \mathit{Pid}_1) = \text{true}, \\ \text{is-monitor-atom}(\text{fun}(\mathit{Atom})) = \text{false}, \\ \text{get-monitor}(\mathrm{BS}, \mathit{Pid}_1) = \mathit{Monitor}\end{array}}{\text{info}(\mathit{Atom}, C, \mathit{Pid}_2) = \text{monitor-info}(\mathit{Monitor}, \mathit{Atom}, C, \mathit{Pid}_2)} \qquad \textbf{[info-mon]}$$

The "next" operation in the case of monitoring is now defined by inserting appropriate monitoring atoms to inform the monitor about the atom just executed. Recall, that the default case for this function was already defined in Section 5.3.2.

$$\frac{\mathit{Pid}_1 = \text{get-proc-id}(C)}{\begin{array}{l}{}^{\nu}\text{monitor-info}(\texttt{logger}(\mathit{Tid}), \mathit{Atom}, C, \mathit{Pid}_2)\,(\Box \,.\, P) = \\ < \texttt{snd-monitor}(\mathit{Tid}, \text{mk-logpoint}(\mathit{Pid}_1, \mathit{Atom}, \text{get-env}(C), \text{get-bus-state}(C), \mathit{Pid}_2)) \,.\, P >\end{array}} \qquad \textbf{[nu-log]}$$

$$\frac{\mathit{Pid}_1 = \text{get-proc-id}(C)}{\begin{array}{l}{}^{\nu}\text{monitor-info}(\texttt{viewer}(\mathit{Tid}), \mathit{Atom}, C, \mathit{Pid}_2)\,(\Box \,.\, P) = \\ < \texttt{snd-monitor}(\mathit{Tid}, \text{mk-viewpoint}(\mathit{Pid}_1, \mathit{Atom}, \text{get-env}(C), \text{get-bus-state}(C), \mathit{Pid}_2)) \\ \quad .\, \texttt{rec-monitor}(\mathit{Tid}, \texttt{continue}(\mathit{Pid}_1)) \,.\, P >\end{array}} \qquad \textbf{[nu-view]}$$

$$\frac{\mathit{Pid}_1 = \text{get-proc-id}(C)}{\begin{array}{l}{}^{\nu}\text{monitor-info}(\texttt{controller}(\mathit{Tid}), \mathit{Atom}, C, \mathit{Pid}_2)\,(\Box \,.\, P) = \\ < \texttt{snd-monitor}(\mathit{Tid}, \text{mk-controlpoint}(\mathit{Pid}_1, \mathit{Atom}, \text{get-env}(C), \text{get-bus-state}(C), \mathit{Pid}_2, P)) \\ \quad .\, \texttt{rec-monitor}(\mathit{Tid}, \texttt{continue}(\mathit{Pid}_1, \texttt{NewP ?}, \texttt{NewEnv ?})) \,.\, \texttt{continuation}(\texttt{NewP}, \texttt{NewEnv}) >\end{array}} \qquad \textbf{[nu-ctl]}$$

Define the atomic steps for attach-monitor and detach-monitor

$$\frac{\begin{array}{c}
\mathrm{PR} = \rho_{Pid}\,(\lambda_{Env}\,(< OAPs' + Atom\,.\,P + OAPs'' >)),\\
C = \mathrm{context}(Pid,\ Env,\ Script,\ \mathrm{BS}),\\
\mathrm{is\text{-}enabled}(Atom,\ C) = \mathrm{true},\\
\mathrm{fun}(Atom) = \texttt{attach-monitor},\ \mathrm{args}(Atom) = [T_1,\ T_2,\ T_3],\\
[Tid,\ MonitorKind(),\ T_3{}'] = \mathrm{substitute}([T_1,\ T_2,\ T_3],\ Env),\\
\mathrm{has\text{-}monitor\text{-}attached}(\mathrm{BS},\ Pid) = \mathrm{false},\\
\mathrm{BS}' = \mathrm{select\text{-}and\text{-}set\text{-}monitor}(T_3{}',\ \{PRs_1 \mid\mid PRs_2\},\ MonitorKind(Tid),\ \mathrm{BS}),\\
\mathrm{PR}' = \rho_{Pid}\,(\lambda_{Env}\,(\nu_{\mathrm{info}(Atom,\ C\ /\ \mathrm{BS}',\ \mathrm{proc\text{-}id}(-\ 1))}\,(\square\,.\,P)))
\end{array}}{\begin{array}{c}
\mathrm{atomic\text{-}steps}(\lambda_{\mathrm{BS}}\,(\mathrm{E}_{Script}\,(\{PRs_1 \mid\mid \square \mid\mid \mathrm{PR} \mid\mid PRs_2\})),\ W) =\\
\mathrm{atomic\text{-}steps}(\lambda_{\mathrm{BS}'}\,(\mathrm{E}_{Script}\,(\{PRs_1 \mid\mid \mathrm{PR}' \mid\mid \square \mid\mid PRs_2\})),\ \mathrm{true})
\end{array}}\ \textbf{[attach-monitor-trans]}$$

$$\frac{\begin{array}{c}
\mathrm{PR} = \rho_{Pid}\,(\lambda_{Env}\,(< OAPs' + Atom\,.\,P + OAPs'' >)),\\
C = \mathrm{context}(Pid,\ Env,\ Script,\ \mathrm{BS}),\\
\mathrm{is\text{-}enabled}(Atom,\ C) = \mathrm{true},\\
\mathrm{fun}(Atom) = \texttt{detach-monitor},\ \mathrm{args}(Atom) = [T],\ [T'] = \mathrm{substitute}([T],\ Env),\\
\mathrm{BS}' = \mathrm{select\text{-}and\text{-}set\text{-}monitor}(T',\ \{PRs_1 \mid\mid PRs_2\},\ \mathrm{none},\ \mathrm{BS}),\\
\mathrm{PR}' = \rho_{Pid}\,(\lambda_{Env}\,(\nu_{\mathrm{info}(Atom,\ C\ /\ \mathrm{BS}',\ \mathrm{proc\text{-}id}(-\ 1))}\,(\square\,.\,P)))
\end{array}}{\begin{array}{c}
\mathrm{atomic\text{-}steps}(\lambda_{\mathrm{BS}}\,(\mathrm{E}_{Script}\,(\{PRs_1 \mid\mid \square \mid\mid \mathrm{PR} \mid\mid PRs_2\})),\ W) =\\
\mathrm{atomic\text{-}steps}(\lambda_{\mathrm{BS}'}\,(\mathrm{E}_{Script}\,(\{PRs_1 \mid\mid \mathrm{PR}' \mid\mid \square \mid\mid PRs_2\})),\ \mathrm{true})
\end{array}}\ \textbf{[detach-monitor-trans]}$$

$$\frac{\begin{array}{c}
\mathrm{PR} = \rho_{Pid}\,(\lambda_{Env}\,(< OAPs' + Atom\,.\,P + OAPs'' >)),\\
C = \mathrm{context}(Pid,\ Env,\ Script,\ \mathrm{BS}),\\
\mathrm{is\text{-}enabled}(Atom,\ C) = \mathrm{true},\\
\mathrm{fun}(Atom) = \texttt{continuation},\\
\mathrm{args}(Atom) = [T_1,\ T_2],\ [T_1{}',\ T_2{}'] = \mathrm{substitute}([T_1,\ T_2],\ Env),\\
P = \mathrm{term2proc}(T_1{}'),\\
Env' = \mathrm{term2env}(T_2{}'),\\
\mathrm{PR}' = \rho_{Pid}\,(\lambda_{Env'}\,(\nu_{\mathrm{info}(Atom,\ C,\ \mathrm{proc\text{-}id}(-\ 1))}\,(\square\,.\,P)))
\end{array}}{\begin{array}{c}
\mathrm{atomic\text{-}steps}(\lambda_{\mathrm{BS}}\,(\mathrm{E}_{Script}\,(\{PRs_1 \mid\mid \square \mid\mid \mathrm{PR} \mid\mid PRs_2\})),\ W) =\\
\mathrm{atomic\text{-}steps}(\lambda_{\mathrm{BS}}\,(\mathrm{E}_{Script}\,(\{PRs_1 \mid\mid \mathrm{PR}' \mid\mid \square \mid\mid PRs_2\})),\ \mathrm{true})
\end{array}}\ \textbf{[continuation-trans]}$$

## 7.8 Reconfiguring the ToolBus

A complete shutdown or a dynamic reconfiguration of a running ToolBus is achieved by the following primitives:

- `shutdown`: terminate *all* currently executing tools as well as all ToolBus processes.
- `reconfigure`: reconfigure the ToolBus by reading a new script, selectively deleting process and tools, and restarting the bus using the new script. Observe that, in combination with the monitoring primitives described earlier, the ToolBus itself can be used to fully support the interactive development of **T** scripts.

| TOOLBUS Process | Tool Control Process | Tool |
|---|---|---|
| shutdown<br>reconfigure<br>restart† | rec-reconfigure-by-TCP | snd-reconfigure |

## Module Reconfigure

**imports** ToolBus$^{(5.5)}$ Expressions$^{(6.5)}$ Execute-Terminate$^{(7.4)}$ Monitoring$^{(7.7)}$ Eval-Do$^{(7.5)}$ ToolDefs$^{(7.1)}$

**exports**

**sorts** PB-PAIR

**lexical syntax**

```
shutdown                 → ATOMIC-FUN
reconfigure              → ATOMIC-FUN
restart                  → ATOMIC-FUN
rec-reconfigure-by-TCP   → ATOMIC-FUN
snd-reconfigure          → ATOMIC-FUN
```

**context-free syntax**

```
detach-deleted-monitor(TERM, PROC-ID, BUS-STATE) → BUS-STATE
"<" PROC "," BUS ">"                             → PB-PAIR
select-or-terminate(TERM, TERM, BUS)             → PB-PAIR
```

**equations**

Define relevant communications.

$$\gamma_1(\texttt{snd-reconfigure}, \texttt{rec-reconfigure-by-TCP}) = \text{true} \qquad \text{[cm-reconfigure]}$$

Remove a monitor from a process if the tool control process for that monitor is not selected.

$$\frac{\begin{array}{c}\text{has-monitor-attached}(\text{BS}, \mathit{Pid}_1) = \text{true}, \\ \text{get-monitor}(\text{BS}, \mathit{Pid}_1) = \mathit{MonitorKind}(\mathit{Tid}), \\ \text{get-controlling-process}(\text{BS}, \mathit{Tid}) = \mathit{Pid}_2, \\ \text{selects}(T, \mathit{Pid}_2, \text{BS}) = \text{false}\end{array}}{\text{detach-deleted-monitor}(T, \mathit{Pid}_1, \text{BS}) = \text{BS}[\texttt{monitor}(\mathit{Pid}_1) := \text{none}]} \qquad \text{[ddm-1]}$$

$$\text{detach-deleted-monitor}(T, \mathit{Pid}_1, \text{BS}) = \text{BS} \quad \textbf{otherwise} \qquad \text{[ddm-2]}$$

Both in case of a complete shutdown of the TOOLBUS and in case of a reconfiguration, it is necessary to gracefully terminate all (or a selection of) processes. The function select-or-terminate does the following:

- constructs a list of remaining processes.
- constructs a processes expression containing (a) snd-terminate-to-tool atoms to terminate all currently active tools (i.e., tool instance processes) that are not selected; (b) snd-monitor atoms to inform a monitor about the termination of a monitored process.

For a selected process, only take care that a monitoring tool that is not selected is detached from the process:

$$\frac{\begin{array}{c}\mathit{Pid} = \text{get-pid}(\text{PR}), \\ \text{selects}(T_1, \mathit{Pid}, \text{BS}) = \text{true}, \\ \text{detach-deleted-monitor}(T_1, \mathit{Pid}, \text{BS}) = \text{BS}'\end{array}}{\begin{array}{c}\text{select-or-terminate}(T_1, T_2, \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \text{PR} \mid\mid \mathit{PRs}_2\}))) = \\ \text{select-or-terminate}(T_1, T_2, \lambda_{\text{BS}'}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \text{PR} \mid\mid \Box \mid\mid \mathit{PRs}_2\})))\end{array}} \qquad \text{[sot-1]}$$

For a process that is not selected but has an attached monitor whose tool control process is selected, inform the monitor about the termination of this process:

$$\frac{\begin{array}{c} \mathit{Pid}_1 = \text{get-pid}(\text{PR}), \\ \text{selects}(T_1,\ \mathit{Pid}_1,\ \text{BS}) = \text{false}, \\ \text{has-monitor-attached}(\text{BS},\ \mathit{Pid}_1) = \text{true}, \\ \mathit{Mid} = \text{get-monitor}(\text{BS},\ \mathit{Pid}_1), \\ \text{get-controlling-process}(\text{BS},\ \mathit{Mid}) = \mathit{Pid}_2, \\ \text{selects}(T_1,\ \mathit{Pid}_2,\ \text{BS}) = \text{true}, \\ \text{BS}' = \text{del-proc-id}(\text{BS},\ \mathit{Pid}_1), \\ \text{select-or-terminate}(T_1,\ T_2, \\ \lambda_{\text{BS}'}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \mathit{PRs}_2\}))) = <P,\ \lambda_{\text{BS}''}\ (\text{E}_{\mathit{Script}'}\ (\{\mathit{PRs}_3\}))> \end{array}}{\begin{array}{c} \text{select-or-terminate}(T_1,\ T_2,\ \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \text{PR} \mid\mid \mathit{PRs}_2\}))) = \\ <\texttt{snd-monitor}(\mathit{Mid},\ \texttt{terminates}(\mathit{Pid}_1,\ T_2))\ .\ P,\ \lambda_{\text{BS}''}\ (\text{E}_{\mathit{Script}'}\ (\{\mathit{PRs}_3\}))> \end{array}} \quad [\text{sot-2}]$$

For a tool control process that is not selected, delete the process and construct an appropriate snd-terminate-by-TCP atom to terminate the tool it controlled.

$$\frac{\begin{array}{c} \mathit{Pid} = \text{get-pid}(\text{PR}), \\ \text{selects}(T_1,\ \mathit{Pid},\ \text{BS}) = \text{false}, \\ \text{get-proc-name}(\text{BS},\ \mathit{Pid}) = \texttt{proc-name}(\texttt{"TCP"}), \\ \text{BS}' = \text{del-proc-id}(\text{BS},\ \mathit{Pid}), \\ \text{select-or-terminate}(T_1,\ T_2,\ \lambda_{\text{BS}'}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \mathit{PRs}_2\}))) = <P,\ \lambda_{\text{BS}''}\ (\text{E}_{\mathit{Script}'}\ (\{\mathit{PRs}_3\}))> \end{array}}{\begin{array}{c} \text{select-or-terminate}(T_1,\ T_2,\ \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \text{PR} \mid\mid \mathit{PRs}_2\}))) = \\ <\texttt{snd-terminate-by-TCP}(\text{get-controlled-ext-tool}(\text{BS},\ \mathit{Pid}),\ T_2)\ .\ P,\ \lambda_{\text{BS}''}\ (\text{E}_{\mathit{Script}'}\ (\{\mathit{PRs}_3\}))> \end{array}} \quad [\text{sot-3}]$$

The remaining cases:

$$\text{select-or-terminate}(T_1,\ T_2,\ \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs} \mid\mid \Box\}))) = <\texttt{tau},\ \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}\}))> \quad [\text{sot-4}]$$

$$\frac{\text{BS}' = \text{del-proc-id}(\text{BS},\ \text{get-pid}(\text{PR}))}{\begin{array}{c} \text{select-or-terminate}(T_1,\ T_2,\ \lambda_{\text{BS}}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \text{PR} \mid\mid \mathit{PRs}_2\}))) = \\ \text{select-or-terminate}(T_1,\ T_2,\ \lambda_{\text{BS}'}\ (\text{E}_{\mathit{Script}}\ (\{\mathit{PRs}_1 \mid\mid \Box \mid\mid \mathit{PRs}_2\}))) \end{array}} \quad \textbf{otherwise} \quad [\text{sot-5}]$$

A shutdown of the TOOLBUS proceeds in three steps:

- Apply select-or-terminate to all processes. As a result, all tool instance processes and the current process are selected.
- Execute all atomic steps generated by the previous step in the current process. This amounts to sending a sequence of snd-terminate-to-tool messages to the tool instance processes.
- Stop the current process (and hence all activities in the TOOLBUS).

Observe that any monitoring of the process containing the shutdown atom is turned off before any further atomic step is performed.

$$\frac{\begin{array}{c}
\mathsf{PR} = \rho_{Pid}\,(\lambda_{Env}\,(<\, OAPs' + Atom\;.\;P + OAPs''\,>)),\\
\mathsf{fun}(Atom) = \mathtt{shutdown},\\
C = \mathsf{context}(Pid,\ Env,\ Script,\ \mathsf{BS}),\\
\mathsf{is\text{-}enabled}(Atom,\ C) = \mathsf{true},\\
\mathsf{args}(Atom) = [T],\\
T' = \mathsf{substitute}(T,\ Env),\\
B = \lambda_{\mathsf{BS}}\,(\mathsf{E}_{Script}\,(\{\square \mid\mid PRs_1 \mid\mid PRs_2\})),\\
\mathsf{select\text{-}or\text{-}terminate}([\mathtt{proc\text{-}name}(\texttt{"TOOL"})],\ T',\ B) = <\, P',\ B'\,>,\\
B' = \lambda_{\mathsf{BS}'}\,(\mathsf{E}_{Script'}\,(\{PRs_3\})),\\
\mathsf{BS}'' = \mathsf{BS}'[\mathtt{monitor}(Pid) := \mathsf{none}],\\
\mathsf{PR}' = \rho_{Pid}\,(\lambda_{Env}\,(\nu_{\mathsf{info}(Atom,\ C\ /\ \mathsf{BS}'',\ \mathsf{proc\text{-}id}(-\ 1))}\,(\square\;.\;P'\;.\;\delta)))
\end{array}}{\begin{array}{c}
\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\,(\mathsf{E}_{Script}\,(\{PRs_1 \mid\mid \square \mid\mid \mathsf{PR} \mid\mid PRs_2\})),\ W) =\\
\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}''}\,(\mathsf{E}_{Script'}\,(\{\square \mid\mid \mathsf{PR}' \mid\mid PRs_3\})),\ \mathsf{true})
\end{array}}\quad\textbf{[shutdown-trans]}$$

A reconfiguration of the TOOLBUS is based on a new **T** script and a selection of processes that should survive the reconfiguration. It consists of the following steps:

- Apply select-or-terminate to all processes. As a result, all tool instance processes, the current process, and an optional list of other processes (as indicated by the argument of reconfigure) are selected.

- Execute all atomic steps generated by the previous step in the current process. This amounts to sending a sequence of snd-terminate-to-tool messages to the tool instance processes.

- Restart the TOOLBUS with a new script; this is achieved by inserting a restart atom in the current process.

Observe that the process containing the reconfigure atom always survives the reconfiguration.

$$\frac{\begin{array}{c}
\mathsf{PR} = \rho_{Pid}\,(\lambda_{Env}\,(<\, OAPs' + Atom\;.\;P + OAPs''\,>)),\\
C = \mathsf{context}(Pid,\ Env,\ Script,\ \mathsf{BS}),\\
\mathsf{is\text{-}enabled}(Atom,\ C) = \mathsf{true},\\
\mathsf{fun}(Atom) = \mathtt{reconfigure},\ \ \mathsf{args}(Atom) = [T_1,\ T_2,\ T_3],\\
[T_1',\ T_2',\ T_3'] = \mathsf{substitute}([T_1,\ T_2,\ T_3],\ Env),\\
[OptTs] = T_2',\\
B = \lambda_{\mathsf{BS}}\,(\mathsf{E}_{Script}\,(\{\square \mid\mid PRs_1 \mid\mid PRs_2\})),\\
\mathsf{select\text{-}or\text{-}terminate}([\mathtt{proc\text{-}name}(\texttt{"TOOL"}),\ OptTs],\ T_3',\ B) = <\, P',\ B'\,>,\\
B' = \lambda_{\mathsf{BS}'}\,(\mathsf{E}_{Script'}\,(\{PRs_3\})),\\
\mathsf{BS}'' = \mathsf{detach\text{-}deleted\text{-}monitor}([OptTs],\ Pid,\ \mathsf{BS}'),\\
C' = C\ /\ Script'\ /\ \mathsf{BS}'',\\
\mathsf{PR}' = \rho_{Pid}\,(\lambda_{Env}\,(\nu_{\mathsf{info}(Atom,\ C',\ \mathsf{proc\text{-}id}(-\ 1))}\,(\square\;.\;P'\;.\;\mathtt{restart}(T_1')\;.\;P)))
\end{array}}{\begin{array}{c}
\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\,(\mathsf{E}_{Script}\,(\{PRs_1 \mid\mid \square \mid\mid \mathsf{PR} \mid\mid PRs_2\})),\ W) =\\
\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}''}\,(\mathsf{E}_{Script'}\,(\{\square \mid\mid \mathsf{PR}' \mid\mid PRs_3\})),\ \mathsf{true})
\end{array}}\quad\textbf{[reconfigure-trans]}$$

The restart atom is defined by a sequence of steps that resembles the sequence used for initializing a complete interpreter (see Section 5.6.1). Observe that we do *not* further specify the function parse here.

We assume that it is defined using standard parsing techniques.

$$\frac{\begin{array}{c}
\mathsf{PR} = \rho_{Pid}\,(\lambda_{Env}\,(< OAPs' + Atom\,.\,P + OAPs'' >)),\\
C = \mathsf{context}(Pid,\ Env,\ Script,\ \mathsf{BS}),\\
\mathsf{is\text{-}enabled}(Atom,\ C) = \mathsf{true},\\
\mathsf{fun}(Atom) = \mathtt{restart},\ \ \mathsf{args}(Atom) = [T],\\
String = \mathsf{interpret}(T,\ C),\\
\mathsf{parse}(String) = Defs\ \mathsf{toolbus}(ProcAppls),\\
Script' = \mathsf{add\text{-}TCP\text{-}defs}(Defs\ \mathsf{toolbus}(ProcAppls)),\\
\mathsf{PR}' = \rho_{Pid}\,(\lambda_{Env}\,(\nu_{\mathsf{info}(Atom,\ C\ /\ Script',\ \mathsf{proc\text{-}id}(-\ 1))}\,(\Box\,.\,P)))
\end{array}}{\begin{array}{l}
\mathsf{atomic\text{-}steps}(\lambda_{\mathsf{BS}}\,(\mathsf{E}_{Script}\,(\{PRs_1 \mid\mid \Box \mid\mid \mathsf{PR} \mid\mid PRs_2\})),\ W) =\\
\mathsf{atomic\text{-}steps}(\mathsf{add\text{-}procs}(ProcAppls,\ Pid,\\
\lambda_{\mathsf{BS}}\,(\mathsf{E}_{Script'}\,(\{PRs_1 \mid\mid \mathsf{PR}' \mid\mid \Box \mid\mid PRs_2\}))),\ \mathsf{true})
\end{array}}\quad \textbf{[restart-trans]}$$

Reconfiguring a TOOLBUS is a very powerful operation and one may have, legitimate, concerns about the overall consistency of the resulting, reconfigured, TOOLBUS. The approach taken here gives no guarantees, since the reconfigure atom only keeps an indicated list of old processes and creates new processes in the reconfigured TOOLBUS. All other consistency considerations have to be programmed explicitly in the **T** script itself. In particular, all attempts to communicate with no longer existing tools or processes should be avoided by properly informing old processes that still exist in the reconfigured bus.

# Chapter 8

# Discussion

In this paper we have:

- introduced the Discrete Time ToolBus architecture for interconnecting software components;
- created a framework for the definition of and experimentation with ToolBus features;
- given a formal, executable, specification of a Discrete Time ToolBus interpreter.

We consider as its main contributions:

- the systematic use of discrete time process algebra in the setting of interconnecting software components;
- the symbolic treatment of process expressions yielding "lazy" interpretation rules that are superior (regarding execution performance) over rules that fully normalize process expressions before interpreting them;
- various aspects of the ToolBus design itself, i.e., the specific form of terms and their use, among others, for matching and information transfer during communication, the monitoring of processes, and the reconfiguration of a ToolBus.
- the idea of randomized execution, that enforces more discipline on the writer of **T** scripts, but has as benefit that **T** scripts are insensible to changes in the scheduling strategy of the implementation by which they are executed.

There are two global views on the ToolBus that we now discuss in some detail.

## 8.1 The ToolBus as software interconnection architecture

The primary motivation for the ToolBus has always been the desire to create an open, flexible, architecture for interconnecting heterogeneous software components. The design presented here already incorporates improvements and extensions based on experiments with the previous version of the ToolBus as described in [BK94]. This has led to the addition of primitives for dynamic execution/termination and connection/disconnection of tools, simple operations on the built-in data type of terms, conditions, and time. Other primitives, e.g., monitors, were added anticipating various classes of usage of the ToolBus. For instance, we expect that *loggers* will play a role in gathering performance data and statistics about the execution of a ToolBus. They may also be used for creating automatic "replay" facilities that record all steps executed by a system based on the ToolBus. Later on, this record can be used to replay the previously recorded session. *Viewers* have clear applications in interactive, non-intrusive, debugging. *Controllers* can be used for intrusive debuggers, but have other applications that are described below. Finally, the reconfiguration primitive is a first step in the direction of providing support for the maintenance of **T** scripts.

## 8.2 The ToolBus as generic run-time environment

A different view on the ToolBus is to consider it as a generic run-time environment that can be used by tools. Controllers play a crucial role in this approach. Recall that controllers can modify both the process expression and the local state of the process they are controlling. One scenario is then to implement some application in a tool $T$ and to attach it as controller to one ToolBus process $P$. Whenever the tool needs to interact with its environment (for instance, to interact with a user-interface or to use some other facility available in the ToolBus), it can send the appropriate process expression to $P$. After executing it, $T$ gets again control and can continue its own computation. In this way, one can build, for instance, interpreters for arbitrary languages and connect them as controllers to the ToolBus, thus re-using existing facilities.

## 8.3 Concluding remarks

The ultimate yardstick for judging the architecture presented here is in its use. Although initial experiments are encouraging, we anticipate further developments in extending and refining the current design.

However, when using the ToolBus the largests investments will be made in the development of *tools* that will be connected to it rather than in the **T** scripts used to connect them. This implies that the stability of the Tool Interface Protocol and the representation of terms should have the highest priority. Unfortunately, "stability" and "further development" are not necessarily good friends of each other. Hereby, we commit ourselves to a further development strategy that ensures maximal (albeit not absolute) stability for tool developers.

## Acknowledgements

Norbert Kajler provoked the second author to come up with generic debugging primitives. This has led to our current notion of "monitors". Simon Gray commented on drafts of this paper. Comments by Steven Klusener on the specification in [BK94] are included in the current specification.

# Bibliography

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

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

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

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

[BB92] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra (extended abstract). In *Proceedings of CONCUR'92*, LNCS 630. Springer Verlag, 1992.

[BB93] J.C.M. Baeten and J.A. Bergstra. Real space process algebra. *Formals Aspects of Computing*, 5(6):481–529, 1993.

[BB95] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. 1995. to appear.

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

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

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

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

[BHK89a] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

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

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

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

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

[BK94] J.A. Bergstra and P. Klint. The ToolBus—a component interconnection architecture. Technical Report P9408, Programming Research Group, University of Amsterdam, 1994.

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

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

[Bri87] E. Brinksma, editor. *Information processing systems–open systems interconnection–LOTOS–a formal description technique based on the temporal ordering of observational behaviour.* 1987. ISO/TC97/SC21.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

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

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

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

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

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

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

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

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

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

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of $\mu$CRL. Technical Report CS-R9076, CWI, 1990.

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

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

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

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

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

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

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

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

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

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

[KPvW95] G. Kok, A. Ponse, and J van Wamel. Grid protocols based on synchronous communication. Technical report, Programming Research Group, Univeristy of Amsterdam, 1995. to appear.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, pages 85–139, 1990.

[MV93] S. Mauw and G.J. Veltink, editors. *Algebraic specification of communication protocols*, volume 36 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

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

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

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

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

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

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

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

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

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

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

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

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

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

# Appendix A

# Primitives available in T scripts

| Primitive | Description | § |
|---|---|---|
| delta | deadlock | 5.1 |
| tau | internal step | 5.1 |
| + | choice | 5.1 |
| . | sequential composition | 5.1 |
| * | iteration | 6.1 |
| \|\| | free merge | 6.2 |
| if ... then ... fi | guarded command | 6.6 |
| if ... then ... else ... fi | conditional | 6.6 |
| create | process creation | 6.7 |
| let ... in ... endlet | variable introduction | 6.3 |
|  | expressions | 6.5 |
| := | assignment | 6.5 |
| snd-msg | send a message (binary, synchronous) | 6.4 |
| rec-msg | receive a message (binary, synchronous) | 6.4 |
| snd-note | send a note (broadcast, asynchronous) | 6.8 |
| rec-note | receive a note (asynchronous) | 6.8 |
| no-note | no notes available for process | 6.8 |
| subscribe | subscribe to notes | 6.8 |
| unsubscribe | unsubscribe from notes | 6.8 |
| delay | relative time delay of atom | 6.10 |
| abs-delay | absolute time delay of atom | 6.10 |
| timeout | relative timeout of atom | 6.10 |
| abs-timeout | absolute timeout of atom | 6.10 |
| rec-connect | receive a connection request from a tool | 7.3 |
| rec-disconnect | receive a disconnection request form a tool | 7.3 |
| execute | execute a tool | 7.4 |
| snd-terminate | terminate the execution of a tool | 7.4 |
| shutdown | terminate ToolBus | 7.8 |
| reconfigure | reconfigure ToolBus | 7.8 |
| attach-monitor | attach a monitoring tool to a process | 7.7 |
| detach-monitor | detach a monitoring tool from a process | 7.7 |
| snd-eval | send evaluation request to tool | 7.5 |
| snd-cancel | cancel an evaluation request to tool | 7.5 |
| rec-value | receive a value from a tool | 7.5 |
| snd-do | send request to tool (no return value) | 7.5 |
| rec-event | receive event from tool | 7.6 |
| snd-ack-event | acknowledge a previous event from a tool | 7.6 |

# Appendix B

# Annotated list of sorts

| Sort | Description | Module | § |
|---|---|---|---|
| AP-FORM | Action prefix forms for process expressions | ActionPrefixForm | 5.4 |
| ATOM | Atomic actions in process expressions | Tscript | 5.1 |
| ATOMIC-FUN | Function symbols of atomic actions | Tscript | 5.1 |
| BOOL | Boolean datatype | Booleans | 4.1 |
| BOOL-CON | Boolean constants (true and false) | Booleans | 4.1 |
| BUS | Complete | StatRepr | 5.3.1 |
| BUS-ASG | Bus state assignments: one (field, value) pair | StateRepr | 5.3.1 |
| BUS-STATE | Complete state | StateRepr | 5.3.1 |
| BUS-VAL | Values to be assigned to fields of bus state | StateRepr | 5.3.1 |
| CONTEXT | Context of one process | BusState | 5.3.2 |
| DEF | Process and tool definitions | Tscript | 5.1 |
| ENTRY | (identifier, value) pair in environment | Environments | 4.4 |
| ENV | Environments representing bindings of variables | Environments | 4.4 |
| ENV-PAIR | Pair of environments | Environments | 4.4 |
| E-PROCESSES | Process creation (E) applied to merge of processes | ToolBus | 5.5 |
| FEATURES | Features in tool definition | ToolDefs | 7.1 |
| FEATURE-ASG | Feature asg. in tool definition: one (id, string) pair | ToolDefs | 7.1 |
| FIELD | Field in bus state | StateRepr | 5.3.1 |
| FORMALS | Formal parameters of process definition | Tscript | 5.1 |
| GEN-VAR | Generalized variables | Terms | 4.1 |
| ID | Identifiers (names of function symbols in terms) | Terms | 4.1 |
| INT | Integers | Integers | 4.1 |
| L-AP-FORM | State operator ($\lambda$) applied to action prefix form | ToolBus | 5.5 |
| NAME | Names of variables and processes (in **T** script) | Tscript | 5.1 |
| NAT | Natural numbers | Integers | 4.1 |
| NAT-CON | Natural constants (numerals) | Integers | 4.1 |
| NOTES | List of notes received by process | Notes | 6.8 |
| NEXT-INFO | Information for the next operator ($\nu$) | ToolBus | 5.5 |
| PNAME | Process names | Tscript | 5.1 |
| PROC | Process expressions | Tscript | 5.1 |
| PROC-APPL | Application of named process expression | Tscript | 5.1 |
| PROC-ID | Process identifications | StateRepr | 5.3.1 |
| PROC-REPR | Representation of one process | ToolBus | 5.5 |
| PROCESSES | Parallel merge of processes | ToolBus | 5.5 |
| PB-PAIR | (Process expression, Bus) pair | Reconfigure | 7.8 |
| STRING | String constants | Terms | 4.1 |
| SUBSCRIPTIONS | The subscriptions of one process | Notes | 6.8 |
| TB-CONFIG | configurations | Tscript | 5.1 |
| T-SCRIPT | **T** scripts | Tscript | 5.1 |
| TERM | Terms (prefix tree structures) | Terms | 4.1 |
| TERM-LIST | List of terms | Terms | 4.1 |
| TIMER | Delay or timeout | Delay-Timeout | 6.10 |
| TIMER-FUN | Function symbols for delay or timeout | Delay-Timeout | 6.10 |
| TOOLS | List of processes representing tools | BusState | 5.3.2 |
| TYPE | Types (of variables) | TypedTerms | 4.2 |
| VAR | Variables in **T** script | Terms | 4.1 |

# Appendix C

# Functions defined by equations in several modules

| Function | Description | Defining equations |
|---|---|---|
| args | Arguments of atom | Tscript (5.1), Conditionals (6.6), Create (6.7), Delay-Timeout (6.10) |
| simple-atomic-step | Atomic step in one process | ToolBus (5.5), Expressions (6.5), Notes (6.8), Connect-Disconnect (7.3), Monitoring (7.7), Reconfigure (7.8) |
| atomic-steps | Atomic step in several processes (like communication) or involving a transformation of a process expression | ToolBus (5.5), Messages (6.4), Conditionals (6.6), Create (6.7), Notes (6.8), Delay-Timeout (6.10), Execute-Terminate (7.4), Monitoring (7.7), Reconfigure (7.8) |
| expand | Expansion of action prefix form | ActionPrefixForm (5.4), Iteration (6.1), FreeMerge (6.2), Let (6.3), Conditionals (6.6), Connect-Disconnect (7.3), Execute-Terminate (7.4), Delay-Timeout (6.10) |
| fun | Function symbol of atom | Tscript (5.1), Expressions (6.5), Conditionals (6.6), Create (6.7), Delay-Timeout (6.10) |
| $\gamma_1$ | Communication function | Tscript (5.1), Messages (6.4), Connect-Disconnect (7.3), Execute-Terminate (7.4), Eval-Do (7.5), Events (7.6), Monitoring (7.7), Reconfigure (7.8) |
| info | Information for "next" function ($\nu$) | ToolBus (5.5), Monitoring (7.7) |
| interpret | Interpret expression | Expressions (6.5), DiscreteTime (6.9) |
| is-enabled | Is atom enabled in a context | BusState (5.3.2), Conditionals (6.6) |
| $\nu$ | Next step in action prefix form | ToolBus (5.5), Monitoring (7.7) |
| prep-proc | Prepare process expression | Prepare (5.2), Iteration (6.1), FreeMerge (6.2), Let (6.3), Expressions (6.5), Conditionals (6.6), Create (6.7), Delay-Timeout (6.10), Connect-Disconnect (7.3), Execute-Terminate (7.4) |
| proc2term | Convert a process expression to a term | Tscript (5.1), Env (4.4), Iteration (6.1), FreeMerge (6.2), Let (6.3), Expressions (6.5), Conditionals (6.6), Create (6.7), Delay-Timeout (6.10) |
| term2proc | Convert a term to a process expression | *See* proc2term |

# Appendix D

# Fields of the bus state

| Field | Sort | Description | Module | § |
|---|---|---|---|---|
| control-tool | TERM | Internal and external id of the tool controlled by a process (per TCP) | ToolDefs | 7.1 |
| monitor | TOOL-ID | Attached monitor (per process) | Monitoring | 7.7 |
| name | TERM | Name of process definition (per process) | BusState | 5.3.2 |
| new-proc-id | PROC-ID | Counter for unique process identifiers | BusState | 5.3.2 |
| new-tool-id | TOOL-ID | Counter for unique tool identifiers | ToolDefs | 7.1 |
| time | INT | Absolute time | DiscreteTime | 6.9 |
| max-time | INT | Maxiaml absolute time | DiscreteTime | 6.9 |

**Notes.**

- All fields marked as "(per process)" or "(per TCP)" are indexed with a process identifier, e.g., name(Pid).
- The control-tool field has as value a list consisting of an (internal) tool identifier followed by an integer representing the external identification of the tool, i.e., [Tid, Int].
- The name field has as value a term of the form proc-name(*String*).

# Appendix E

# Discrete time process algebra for the ToolBus

## E.1 Preliminaries

With $a^{\vee}(n+1)$ we denote a process that may perform the action $a$ during the course of time slice number $n+1$ or, alternatively, it may idle indefinitely. The $n+1$-time slices takes time from $time = n$ to $time = n+1$. Within a slice actions take place in interleaved fashion.

We follow discrete time process algebra in [BB95] in our explanation. In the syntax used there, we have $a^{\vee}(n+1) = \underline{a}(n+1) + \delta$ where $\underline{a}(n+1)$ is a process that *must* perform $a$ in time slice $n+1$ and $\delta$ is a process that idles forever.

To model **T** scripts, it suffices to work in a subalgebra of the parametric time process algebra of [BB95]. This subalgebra is generated by actions $a^{\vee}(n)$ rather than $\underline{a}(n)$. Parametric discrete time processes allow initialisation at any time $t \in N$. With $n \gg P$ we denote the process that $P$ develops into $P$ after initialization at $n$. $n \gg P$ itself is a so called absolute discrete time process. Its actions are all timed with reference to the same initial time 0

Using time spectrum abstraction we introduce parametric time processes: $P = \surd_d x.F$. When initialised at $n$ this $P$ behaves as $P[n/x]$ (or, more precisely, as $n \gg P[n/x]$). Time spectrum abstraction in the real time case was introduced in [BB93]. The discrete time case occurs in [BB92].

Two parametric time processes are equal if they are equal after all possible initializations. In this way equality is reduced to the simple notion of strong bisimulation for absolute discrete time transition systems. The Extensionality for Parametric Discrete Time rule

$$\frac{\text{for all } n : n \gg X = n \gg Y}{X = Y} \quad \text{(EPDT)}$$

embodies this version of process equality in parametric time. Notice that relative time notation is easily obtained on the basis of time spectrum abstraction: $a^{\vee}[n] = \surd_d x.a^{\vee}(n+x)$. Hence, $a^{\vee}[n]$ can perform $a$ in slice "$n$ after initialization" or idle.

Using $a^{\vee}(x)$, time spectrum abstraction, and infinite sums $\sum_{i \in N} P_i$ we can define the meaning of the actions *script*($a$) that occur in the ToolBus by

$$script(a) = \sum_{i \in N} a^{\vee}(i+1).$$

This means that to explain $a$ as an action in a **T** script in the discrete time setting, we replace it by an infinite sum of $a^{\vee}$'s. We will write $a$ for *script*($a$) when no confusion arizes.

Timed atoms are now described in the following table. Note that $e$ is an expression with free variable *time*, and the empty sum equals $\delta$.

| | |
|---|---|
| absdelay$(a, e)$ | $= \surd_d \cdot \sum_{i\in N}(i \geq e[n/time]) :\rightarrow a^{\vee}(i)$ |
| abstimeout$(a, e)$ | $= \surd_d \cdot \sum_{i\in N}(i < e[n/time]) :\rightarrow a^{\vee}(i)$ |
| absinterval$(a, e_1, e_2)$ | $= \surd_d \cdot \sum_{i\in N}(i \geq e_1[n/time]) \wedge (i < e_2[n/time]) :\rightarrow a^{\vee}(i)$ |
| reldelay$(a, e)$ | $= \surd_d \cdot \sum_{i\in N}(i \geq e[n/time] + n) :\rightarrow a^{\vee}(i)$ |
| reltimeout$(a, e)$ | $= \surd_d \cdot \sum_{i\in N}(i < e[n/time] + n) :\rightarrow a^{\vee}(i)$ |
| relinterval$(a, e_1, e_2)$ | $= \surd_d \cdot \sum_{i\in N}(i \geq e_1[n/time] + n) \wedge (i < e_2[n/time] + n) :\rightarrow a^{\vee}(i)$ |

In the axiomatization below we will avoid time spectrum abstraction and infinite sums, thus obtaining equations that are closer to an implementation though (perhaps) less intuitive.

## E.2 Untimed process algebra axioms

For reference purposes, we include here first a table of standard (untimed) process algebra axioms as it was given in [BK94]. (But see the notes below).

| | | | | | |
|---|---|---|---|---|---|
| Basic Process Algebra (BPA) | | | Encapsulation operator | | |
| $x + y$ | $= y + x$ | A1 | $\partial_H(a)$ | $= a$, if $a \notin H$ | D1 |
| $(x + y) + z$ | $= x + (y + z)$ | A2 | $\partial_H(a)$ | $= \delta$, if $a \in H$ | D2 |
| $x + x$ | $= x$ | A3 | $\partial_H(x + y)$ | $= \partial_H(x) + \partial_H(y)$ | D3 |
| $(x + y).z$ | $= x.z + y.z$ | A4 | $\partial_H(x.y)$ | $= \partial_H(x).\partial_H(y)$ | D4 |
| $(x.y).z$ | $= x.(y.z)$ | A5 | | | |
| | | | Renaming operator | | |
| Deadlock (BPA$_\delta$) | | | $\rho_f(\delta)$ | $= \delta$ | RN0 |
| $x + \delta$ | $= x$ | A6 | $\rho_f(a)$ | $= f(a)$ | RN1 |
| $\delta.x$ | $= \delta$ | A7 | $\rho_f(x + y)$ | $= \rho_f(x) + \rho_f(y)$ | RN2 |
| | | | $\rho_f(x.y)$ | $= \rho_f(x).\rho_f(y)$ | RN3 |
| Free merge operator | | | $\rho_{id}(x)$ | $= x$ | RR1 |
| $x \parallel y$ | $= x \mathbin{\parallel\!\!\lfloor} y + y \mathbin{\parallel\!\!\lfloor} x$ | M1 | $\rho_f \circ \rho_g(x)$ | $= \rho_{f\circ g}(x)$ | RR2 |
| $a \mathbin{\parallel\!\!\lfloor} (0 \gg x)$ | $= a.(0 \gg x)$ | M2 | | | |
| $a.x \mathbin{\parallel\!\!\lfloor} (0 \gg y)$ | $= a.(x \parallel (0 \gg y))$ | M3 | Process creation operator | | |
| $(x + y) \mathbin{\parallel\!\!\lfloor} z$ | $= x \mathbin{\parallel\!\!\lfloor} z + y \mathbin{\parallel\!\!\lfloor} z$ | M4 | $E_\phi(a)$ | $= a$, if $a \notin cr(D)$ | CR1 |
| | | | $E_\phi(cr(d))$ | $= \overline{cr}(d).E_\phi(\phi(d))$, for $d \in D$ | CR2 |
| Merge operator | | | $E_\phi(a.x)$ | $= a.E_\phi(x)$, if $a \notin cr(D)$ | CR3 |
| $a \mid b$ | $= \gamma(a, b)$, if $\gamma$ defined | CF1 | $E_\phi(cr(d).x)$ | $= \overline{cr}(d).E_\phi(\phi(d) \parallel x)$ for $d \in D$ | CR4 |
| $a \mid b$ | $= \delta$, otherwise | CF2 | $E_\phi(x + y)$ | $= E_\phi(x) + E_\phi(y)$ | CR5 |
| | | | | | |
| $x \parallel y$ | $= x \mathbin{\parallel\!\!\lfloor} y + y \mathbin{\parallel\!\!\lfloor} x + x \mid y$ | CM1 | State operator | | |
| $a \mathbin{\parallel\!\!\lfloor} (0 \gg x)$ | $= a.(0 \gg x)$ | CM2 | $\lambda_S(\delta)$ | $= \delta$ | SO1 |
| $a.x \mathbin{\parallel\!\!\lfloor} (0 \gg y)$ | $= a.(x \parallel (0 \gg y))$ | CM3 | $\lambda_S(a)$ | $= a(S)$ | SO2 |
| $(x + y) \mathbin{\parallel\!\!\lfloor} z$ | $= x \mathbin{\parallel\!\!\lfloor} z + y \mathbin{\parallel\!\!\lfloor} z$ | CM4 | $\lambda_S(a.x)$ | $= a(S).\lambda_{S(a)}(x)$ | SO4 |
| $a.x \mid b$ | $= (a \mid b).x$ | CM5 | $\lambda_S(x + y)$ | $= \lambda_S(x) + \lambda_S(y)$ | SO5 |
| $a \mid b.x$ | $= (a \mid b).x$ | CM6 | | | |
| $a.x \mid b.y$ | $= (a \mid b).(x \parallel y)$ | CM7 | Iteration operator | | |
| $(x + y) \mid z$ | $= x \mid z + y \mid z$ | CM8 | $x * y$ | $= x.(x * y) + y$ | I |
| $x \mid (y + z)$ | $= x \mid y + x \mid z$ | CM9 | | | |
| | | | Conditional control | | |
| | | | $T :\rightarrow x$ | $= x$ | C1 |
| | | | $F :\rightarrow x$ | $= \delta$ | C2 |

**Notes:**

- BPA consists of A1 ... A5.
- PA consists of BPA plus free merge operator.
- ACP consists of BPA$_\delta$ plus merge and encapsulation.
- Axioms M1, M2, (CM2), M3 (CM3) have been modified using $0 \gg x$ instead of $x$. Here, $0 \gg x$ denotes the initialization of $x$ in 0. In the absence of time bound actions, $0 \gg x$ is just $x$. This modification guarantees consistency with the setting involving time bounds.

In the following sections we extend the above table with additional axioms for discrete time process algebra.

## E.3 Axioms for timed atoms and initialization

**Timed atoms**

| | |
|---|---|
| $a^{\vee}(0)$ | $= \delta$ |
| $\delta^{\vee}(k)$ | $= \delta$ |
| $a^{\vee}(k+1).x$ | $= a^{\vee}(k+1).k \gg x$ |

**Initialization**

| | |
|---|---|
| $k \gg a$ | $= a^{\vee}(k+1) + (k+1) \gg a$ |
| $k \gg (x+y)$ | $= k \gg x + k \gg y$ |
| $k \gg (x.y)$ | $= (k \gg x).y$ |
| $k \gg (x \parallel y)$ | $= (k \gg x) \parallel (k \gg y)$ |
| $k \gg (x \mathbin{\lfloor\!\lfloor} y)$ | $= (k \gg x) \mathbin{\lfloor\!\lfloor} (k \gg y)$ |
| $k \gg (x \mid y)$ | $= (k \gg x) \mid (k \gg y)$ |
| $k \gg \partial_H(x)$ | $= \partial_H(k \gg y)$ |
| $k \gg \rho_f(x)$ | $= \rho_f(k \gg x)$ |
| $k \gg \lambda_S(x)$ | $= \lambda_S(k \gg x)$ |
| $k \gg E_\phi(x)$ | $= E_\phi(k \gg x)$ |
| $k \gg l \gg x$ | $= max(k,l) \gg x$ if $k \leq l$ |
| $k \gg l \gg m \gg x$ | $= max(k,l) \gg m \gg x$ |

## E.4 Initialization axioms for atoms with time constraints

**Delays and timeouts**

| | |
|---|---|
| $k \gg absdelay(a,e)$ | $= max(k, e[k/time]) \gg a$ |
| $k \gg abstimeout(a,e)$ | $= e[k/time] > k :\!\rightarrow (a^{\vee}(k+1) + (k+1) \gg abstimeout(a, e[k/time]))$ |
| $k \gg absinterval(a,e_1,e_2)$ | $= max(k, e_1[k/time]) \gg abstimeout(a, e_2[k/time])$ |
| $k \gg reldelay(a,e)$ | $= (k + e[k/time]) \gg a$ |
| $k \gg reltimeout(a,e)$ | $= e[k/time] > 0 :\!\rightarrow (a^{\vee}(k+1) + (k+1) \gg reltimeout(a, e[k/time]-1))$ |
| $k \gg relinterval(a,e_1,e_2)$ | $= (k + e_1[k/time]) \gg reltimeout(a, e_2[k/time])$ |

**Extensionality**

$$\frac{\text{for all } n : n \gg x = n \gg x}{x = x}$$

## E.5 Operator evaluation axioms for timed atoms

**Left merge**

| | | |
|---|---|---|
| $a^\vee(k) \mathbin{\Vert\!\!\lfloor} (l \gg x)$ | $= a^\vee(k).(l \gg x)$ | M2$^\vee$ CM2$^\vee$ |
| $a^\vee(k).x \mathbin{\Vert\!\!\lfloor} (l \gg y)$ | $= a^\vee(k).(x \Vert l \gg y)$ | M3$^\vee$ CM3$^\vee$ |

**Communication merge**

| | |
|---|---|
| $a^\vee(k) \mid b^\vee(l)$ | $= \delta$ if $k \neq l$ |
| $a^\vee(k) \mid b^\vee(k)$ | $= (a \mid b)^\vee(k)$ |

**Encapsulation**

| | |
|---|---|
| $\partial_H(a^\vee(k))$ | $= a^\vee(k)$ if $a \notin H$ |
| $\partial_H(a^\vee(k))$ | $= \delta$ if $a \in H$ |

**Renaming**

| | |
|---|---|
| $\rho_f(a^\vee(k))$ | $= (f(a))^\vee(k)$ |

**State operator**

| | |
|---|---|
| $\lambda_S(a^\vee(k))$ | $= a(S)^\vee(k)$ |
| $\lambda_S(a^\vee(k).x)$ | $= a(S)^\vee(k).\lambda_{S(a)}(x)$ |

Here $a(S)^\vee(k)$ is the action that takes place if within the scope of $\lambda_S$ $a$ takes place in time slice $k$. $S(a)$ is the new state after performing $a$ in state $S$ during time slice $k$.

**Process creation**

| | |
|---|---|
| $E_\phi(a^\vee(k))$ | $= a^\vee(k)$ if $a \notin \mathrm{cr}(D)$ |
| $E_\phi(\mathrm{cr}(d)^\vee(k))$ | $= \overline{\mathrm{cr}}(d)^\vee(k).E_\phi(\phi(d))$ |
| $E_\phi(\mathrm{cr}(d)^\vee(k).x)$ | $= a^\vee(k).E_\phi(x)$ |
| $E_\phi(\mathrm{cr}(d)^\vee(k).x)$ | $= \overline{\mathrm{cr}}(d)^\vee(k).E_\phi(x \Vert \phi(d))$ |

**Remark** Observe that all axioms where some operator distributes over $+$ can also be applied to infinite sums, e.g.,

$$(\sum_i X_i).Y = \sum_i (X_i.Y)$$

This is helpful to rewrite the expressions that emerge if **T** scripts are assigned a meaning by means of the definitions given in Section E.1.

# Index