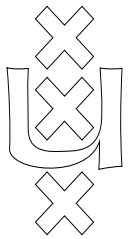
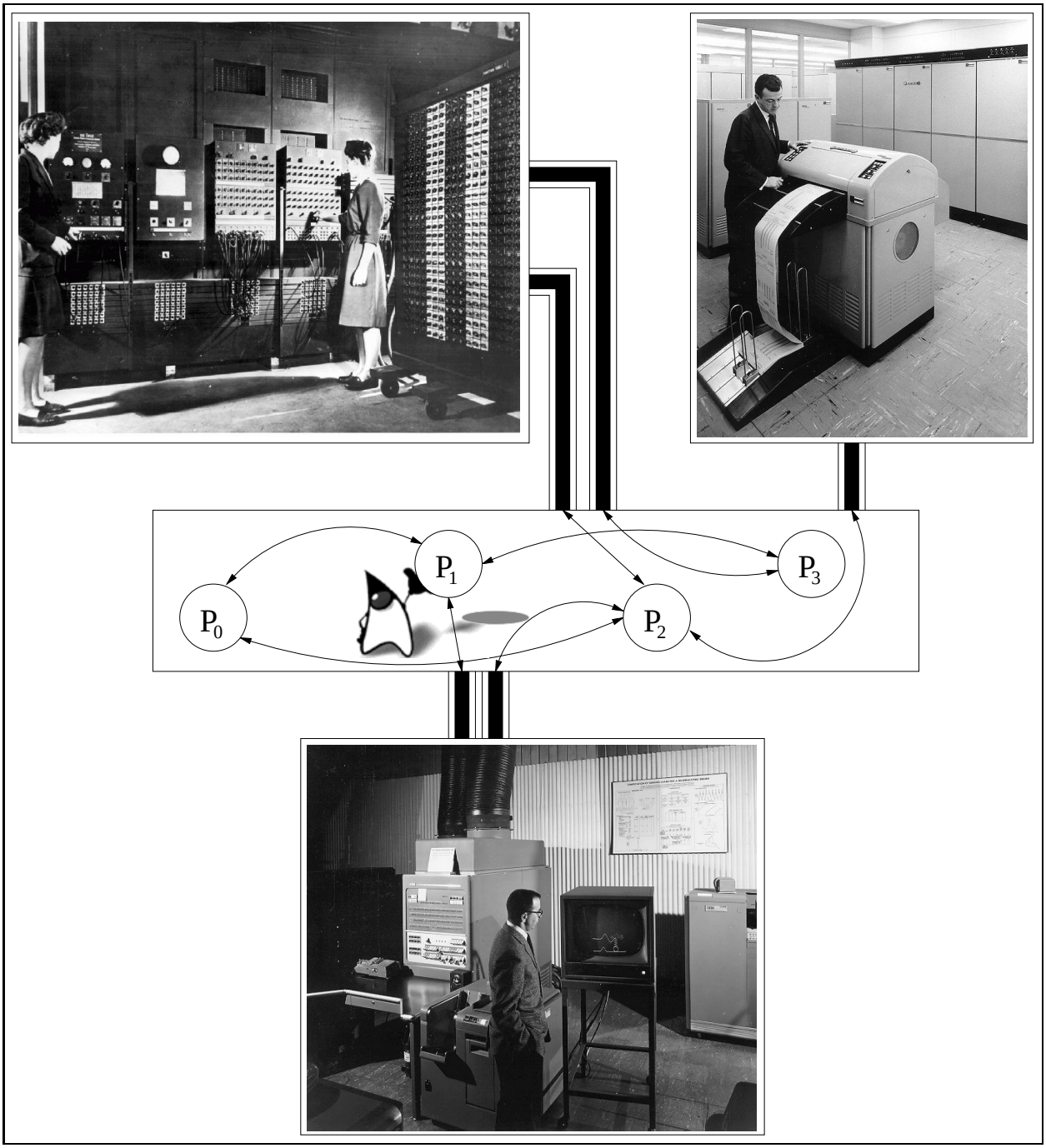


University of Amsterdam
Programming Research Group

A ToolBus Prototype Written in Java

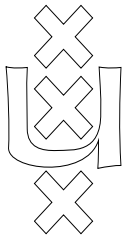


Coen L.S. Visser

Master's thesis June 2000

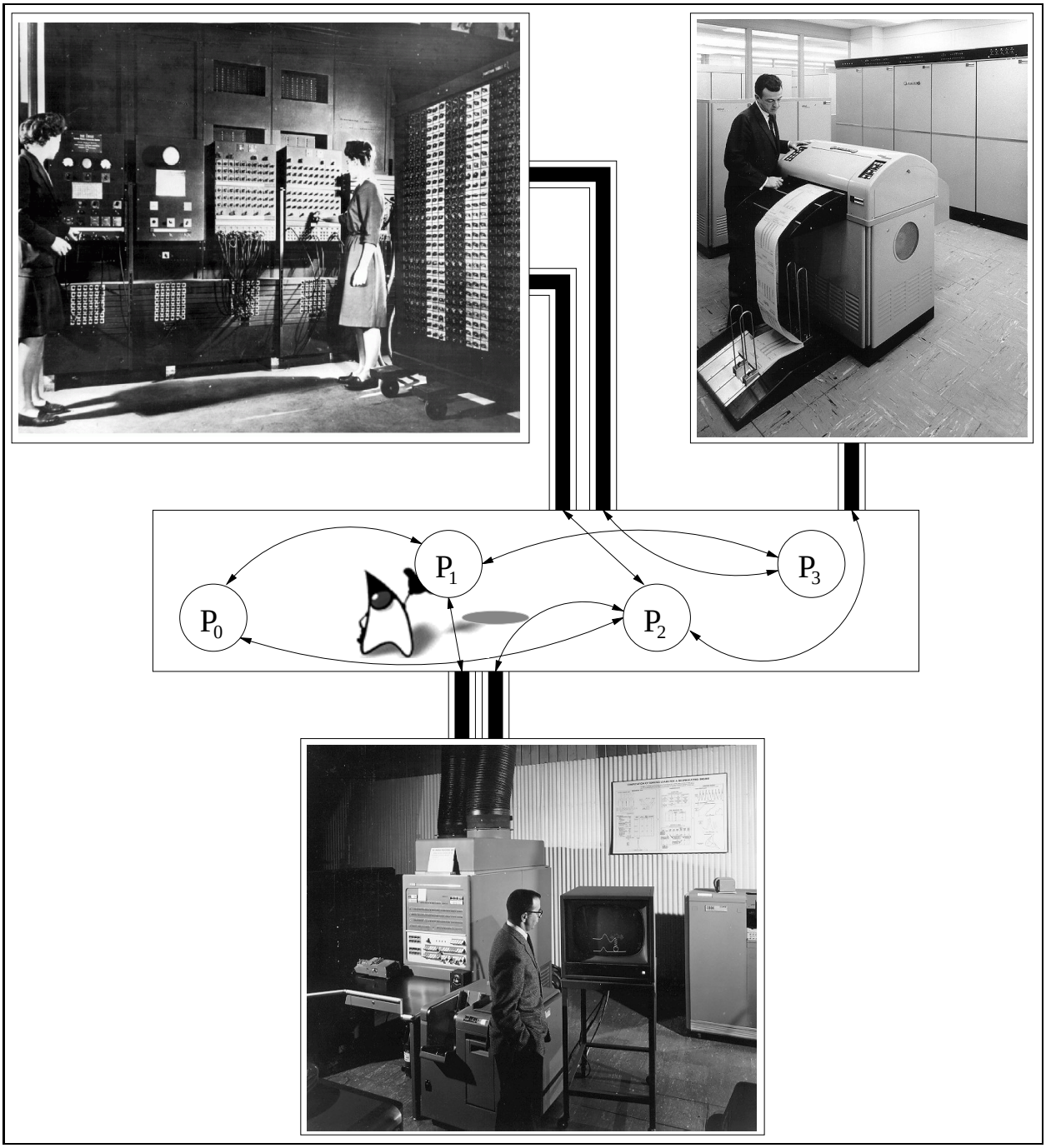
University of Amsterdam
Programming Research Group

Supervisors: Prof. dr. P. Klint and dr. P. Olivier



University of Amsterdam
Programming Research Group

A ToolBus Prototype Written in Java



Supervisors: Prof. dr. P. Klint and dr. P. Olivier

Coen L.S. Visser

Master's thesis June 2000

Acknowledgments

First of all I like to thank my parents and Pascale for their love, patience, and support. I want to thank Jurg van Vliet for a pleasant time studying together. Many thanks to Jurgen Vinju for the music, inspiration, and lots of fun. Chris Verhoef was a great resource for references and good advice. No riddle was too difficult for Pieter Olivier. Hayco de Jong was an unexpected but excellent proof reader. My colleagues at TNO-FEL where very supportive with their continuous harras^H^H^H^H^H encouragements. And last but not least many thanks to Dante for being asleep at night.

Contents

Chapter 1

Introduction

The construction and deployment of large software systems has more in common with organic growth than with structured and controlled development. Successful software systems need constant revisions, bug-fixes and extensions due to ever changing organizational demands. Additions and changes are often applied in an ad-hoc manner by adding or replacing application programs and libraries. After years of maintenance the modularity of systems disappears in a forest of remote procedure calls, call-backs and other constructs that break interface and module boundaries. Such legacy systems are a nightmare for the software engineers that have to maintain them and a thankful study object for computer scientists. Through the years computer scientists and software developers have proposed several solutions for the problem of maintaining and developing large software systems. One of them is the ToolBus [BK95].

We will continue this introduction with a very short overview of the history of software engineering followed by the motivation behind the development of the ToolBus. The chapter is concluded with the aims of our project. In chapter two we will give an overview of the ToolBus architecture including its formal foundations. In chapter three we will discuss our design and the implementation of the individual elements of the ToolBus. The concluding chapter shows the integration of the separate ToolBus parts and what the consequences are of the specific design choices we made.

1.1 A short history of software engineering

One of the great insights in theoretical computer science is that to express an algorithm in a program it does not matter which programming language is used [LV97] (page 98 and further). This suggests that all reasonable programming languages, like C, Java, LISP, and ASF+SDF [vD94], have the same computational power. However this way of reasoning places program development completely in the domain of formal abstractions, but one of the major problems in software engineering is the translation of real-world phenomena to a formal description. As the history of programming shows there is no single programming language that is preferable above all the others. Instead of converging to a small set of increasingly powerful languages we see the opposite development. Thanks to progress in the field of compiler generators and formal languages there has been an explosion of the number of programming languages. Many new languages are domain specific, small languages that are well-suited for a specific task. Using domain specific languages has obvious advantages. Programs can be shorter and easier to understand because the abstractions they contain are closely related to the real world notions they describe. Another, parallel development in programming languages coming from the software industry rather than academic research is object-orientation. The notion of an object is closely related to the way in which we make abstractions of real-world phenomena. So object-orientation makes it easier for a programmer to think about problems and solutions in a natural way.

Despite all the improvements in the way we can model the world developing large software

systems has not become much easier. Although the proliferation of programming languages has its advantages, it has also complicated the interoperability of software. In the increasingly network oriented environment in which our software lives it is not uncommon that even a small application consists of three programs written in three different languages. Consider for example a World Wide Web application using a custom made web server module, written in C, providing data from a relational database, queried in SQL, presented on a HTML-page generated from a Perl script. This is a common scenario that can easily get more complicated if there is JavaScript code on the HTML page or a Java applet.

It was in the context of computer networks that the interoperability issue was first recognized by researchers. The developers of the Internet provided the solution to the problem of how to let different computers exchange raw data. It took some time after computer networks and multi-tasking operating systems had gained popularity that people became aware of the problem of how to make different applications cooperate. This time it was the software industry that provided a number of, not surprisingly, incompatible solutions. Key to the solutions is the concept of a component [Szy97]. Components are configurable building blocks from which larger programs can be constructed. Using a well-specified interface, components can use a dedicated communication channel to exchange information. The communication facilities provide for the exchange of complex data types and even objects. There are a few popular infrastructures that are used to connect components. The most well known are COM [WK94] developed by Microsoft and CORBA [Gro99] of the Object Management Group. Sun's JavaBeans [Mic97] is an example of a component architecture with emphasis on the components instead of the communication infrastructure. All the component software frameworks are designed for maximal flexibility because their success depends on the adoption of the framework by developers.

In [Szy97] a potential market of software components is described in which component creators can sell generally usable components as third party building blocks that are used by component assemblers to build software. A prerequisite is that the delivered components are properly specified and behave according to their specification. But even if a correct formal specification of components is supplied there is no guarantee that software build out of them works as expected. Interaction between otherwise correctly behaving components can lead to subtle errors especially when there is no single party responsible for the order of events. The ToolBus is designed to avoid this situation by providing a central control structure for distributed components. Each component is an independent part of the distributed system but communication with other components is strictly regulated by the ToolBus. This is done without making any assumption about the order in which (communication) events happen so the distributed character of the system is preserved.

1.2 Aims of this project

The main goal of this project is to experiment with an alternative implementation of the ToolBus. Before the project started there were already two complete implementations of the ToolBus, so why build another one? In order to answer this question we have to tell something about the existing implementations. The first one is a semi-executable specification written in the ASF+SDF formalism. ASF+SDF is an algebraic specification formalism that is used among other things for the automatic generation of software engineering tools. The ASF+SDF specification is executable in principle, but it serves more as a formal specification of the ToolBus architecture than as an application program. The second implementation, written in C, is a complete ToolBus system. It consists of a stand alone **T** script interpreter and several adapters that make it possible to write components in different programming languages (C, TCL/Tk, Java, SQL, ...) and connect them to the ToolBus. The C implementation follows the ASF+SDF specification for a large part. The approach taken by the C implementation is to transform a **T** script into one large term. This term is then interpreted following the rewrite rules of the ASF+SDF specification.

The C implementation of the ToolBus has the advantage that it is a straightforward implementation of the ASF+SDF specification, translating rewrite rules from ASF+SDF to C leaves little room for implementation errors. A major drawback is that the C implementation does not allow

for threaded evaluation of **T** scripts. Any parallelism has to be simulated by the term rewriter. It would be nice to have another implementation that allows for experiments with multi-threading. Another problem is that the absence of other functional implementations makes it hard to make claims about the performance of the current ToolBus implementation. Although the performance of the ToolBus interpreter is quite good, the question remains if another design of the interpreter can improve on it. This motivated us to experiment with a radically different implementation. An implementation that would use as little of the current specification and implementation as possible.

For this project we chose Java [JGS96] as implementation language instead of C. The support for object-oriented programming made it an attractive choice. Especially because we wanted to approach the implementation of the ToolBus in a way that differs as much as possible from the term rewriting paradigm. The idea was to make optimal use of the mapping between the basic ToolBus concepts and Java classes. Other useful language features that supported the decision to choose Java for implementation were the built-in garbage collection and multi-threading support. Last but not least two important parts of the ToolBus architecture were already available in Java: the ATerm library [vdBdJKO99] and an adapter for connecting Java to the ToolBus. The language features and availability of supporting tools made Java look like a good choice for building a prototype implementation of the ToolBus. It is interesting to note that at the end of this project we can conclude that most of our motivations for choosing Java as implementation language were not as valid as we initially thought.

Chapter 2

The Discrete Time ToolBus

In this chapter we present an overview of the ToolBus architecture, the general ideas behind it and its formal foundations. In chapter one we have used the word 'ToolBus' in two different ways: to describe an architecture and to describe a specific instantiation of a ToolBus in a software system. We shall now use a more precise terminology to prevent confusion. When using the term ToolBus we will be talking about the general architecture. A specific instance of a ToolBus within the setting of a larger software system will be called a ToolBus configuration. The ToolBus architecture is designed to facilitate the construction of software systems consisting of cooperating components. The exact definition of a component is still matter of discussion (see [Szy97]). We will use the term component in a liberal manner, denoting parts of a software system that interact with other parts using their respective interfaces. Using this "definition" a component can be as small as a single function and as large as a database engine. When components are used in the context of the ToolBus we will call them *tools*.

The organization of this chapter is as follows. First we place the ToolBus in perspective by comparing its approach with other software engineering structures. Second we discuss the most abstract features/foundations of the ToolBus, terms and process algebra. Finally we give a short overview of the ToolBus scripting language with an example of a ToolBus configuration.

2.1 Context of the ToolBus architecture

The need to bring structure in software development was recognized by computer scientists in the late sixties [Dij68]. A result of this was the introduction of structured programming as a software development technique. The idea of structured programming has led to new programming concepts such as procedures, modules, objects and components. While many program development tools support these concepts there is not much tooling that supports structure on the level above single components. When components have to cooperate there are only a few choices to provide structure and preserve the sanity of software engineers.

An often used approach to maintain structure between components is to use a strict hierarchical client-server architecture, like in database management systems. The advantage is that the interaction between the clients and the server is very clear. A disadvantage is that the communications between clients is not as clearly regulated. In figure 2.1 a fictitious example is shown of a software system consisting of a corporate database and several client applications. The database server has a central role and several clients connect with it in a strictly organized way.
However, the interaction between the clients is organized in a very ad hoc manner for example using remote procedure calls between components. This is a very flexible approach, but it has the disadvantage that the dependencies between components can become very complex. Furthermore it is very dependent on the programming language/remote invocation mechanism used.

The ToolBus provides structure at the highest technical level of a software system while still maintaining flexibility. The approach taken by the ToolBus is to use an independent programmable

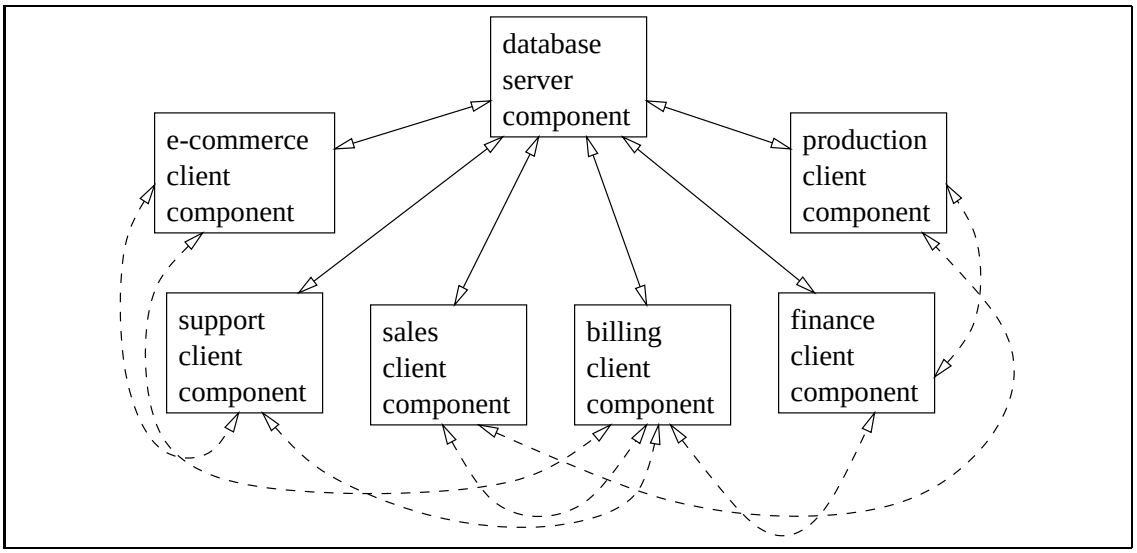


Figure 2.1: Software system in a client server setting

coordination facility to control all interaction between different tools (components). In figure 2.2 the previous example is reworked into a system based upon the ToolBus architecture. All commu-

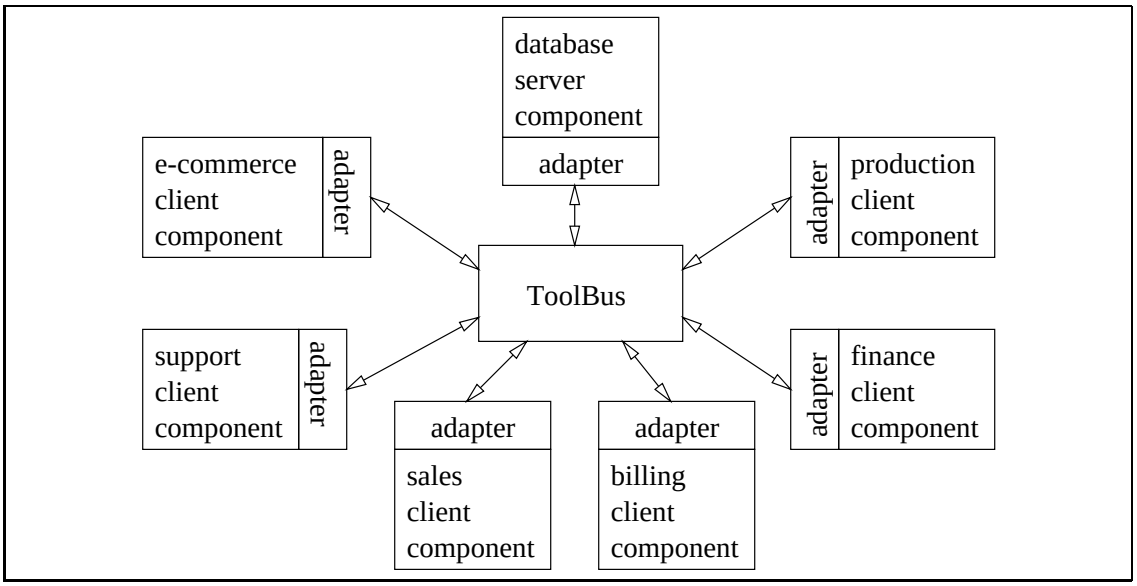


Figure 2.2: Software system using the ToolBus

nication between tools and the ToolBus takes place through a network interface and is facilitated by small tool interfaces called ToolBus *adapters*. A ToolBus adapter implements the protocol defined for the exchange of messages between tools and the ToolBus. For the communication between the ToolBus and tools a protocol is involved that distinguishes between:

- A process [1] requesting an action from a tool: after sending the request the process continues with its own business. The tool performs the action that was requested.

[1] The word "process" will be reserved for describing internal ToolBus processes. Processes that are outside the ToolBus context will be referred to as operating system processes, or tools when appropriate.

- A process requesting a tool to perform some function: after sending the request the process waits for the tool to return a result.
- A tool sending an event to a process: after an event is received by a process it must be acknowledged.

To be able to connect tools written in different programming languages to the ToolBus adapters convert the native data format of the tool to the native data format of the ToolBus and vice versa. This data format is also used internally by the ToolBus. The communication between internal processes is done through the synchronous sending and receiving of messages. Additionally asynchronous communication between processes is also possible. The script language of the ToolBus **T** is specifically suited to define the behavior of distributed systems. In a ToolBus configuration a **T** script contains the definition of a number of processes, including all communications between different internal processes and tools.

PVM [GBD+94] and MPI [MPI95] are two other major software architectures that are designed to coordinate distributed components. However their use lies more in the field of scientific computing than information systems. They support a number of fixed system topologies like a ring topology and a Cartesian topology to implement distributed algorithms. PVM and MPI are ported to many platforms but the implementations focuses on programming languages like FORTRAN and C that are traditionally used in scientific computing.

2.2 Terms

The native datastructure of the ToolBus is called a *term*. The ToolBus uses terms to transport values internally and to communicate with tools. The terms used in the C implementation of the ToolBus are the predecessor of annotated terms, called ATerms [vdBdJKO99], which is the fundamental datastructure of the ASF+SDF Meta-environment[2]. This is not surprising as the ASF+SDF Meta-environment uses term rewriting [BK87], [BKM89] to execute its specifications. We will give an introduction to the terms as used in the C implementation of the ToolBus. For the description of terms we will use the syntax that is used in **T** scripts.

2.2.1 Basic terms

Basic terms are simple values like numbers and strings. The basic terms available in the ToolBus are:

- **Boolean values**: The values true and false.
- **integers**: $\ldots, -1, 0, 1, \ldots$
- **strings**: `"This is a string."`
- **real values**: $3.141592653589, -3.2\text{E}{-}16$ et cetera.
- **binary values**: For example the contents of a JPEG image.

For our ToolBus prototype we only use Boolean values and integers.

2.2.2 Term variables

Apart from the basic terms the ToolBus also needs variables. In a **T** script ToolBus variables start with an uppercase character and have a type (Boolean, integer, string, real, binary) associated with them. We distinguish between:

- **(plain) variables**: Num, Name, TruthVal, ...

[2]The ASF+SDF Meta-environment is the "programming environment" based on the ASF+SDF formalism

- **result variables**: InVal?, FileName?, Result?, ...

A plain variable is used by the ToolBus to store and manipulate values. A result variable is denoted by appending a question mark to a variable name. Result variables are used to communicate values between the ToolBus and tools, and between different ToolBus processes (see 2.3).

2.2.3 Recursive terms

More complex terms can be build by using function application and/or lists. Functions are terms that consist of an identifier followed by zero or more terms enclosed in brackets:

- `pow(2, 4)`
- `f(X, add(2, 4))`

These functions have no meaning of themselves, so `pow(2, 4)` will not automatically be interpreted as 2^4. Lists are terms that consist of one or more terms enclosed in square brackets.

- `[A, Simple, List]`
- `[On, "a clear disk", you(can, seek, ["forever."])]`

In the second example we have used the convention to leave out the brackets of the function applications `can` and `seek` because they have zero arguments. Using function applications and lists most complex datastructures occurring in tools can be represented.

2.3 Process algebra

The behavior of a ToolBus configuration is determined by its **T** script. One of the advantages of the ToolBus is that its scripting language has a formal foundation. This facilitates automatic verification of desired properties like the absence of deadlocks, although this is still quite hard and an interesting study in itself [Rom99]. The **T** script language is based on discrete time process algebra; in fact it has been called syntactically sugared discrete time process algebra [BB95]. Usually process algebra [BW90] is used to model an existing process after which the presence of desired properties can be proved. In the ToolBus architecture process algebra is used in a reverse manner. There is no existing process that we want to describe using process algebra. We rather describe, using process algebra, the behavior of a process as we want it to be. This model is then used directly to simulate a system with the described behavior.

The prototype ToolBus does not implement all process algebra features that are available in the C implementation. While the C implementation of the ToolBus implements a discrete time process algebra the notion of time is lacking from our implementation. In the rest of this text we will not regard the discrete time properties of process algebra.

2.3.1 Example specification

The key elements of process algebra are atomic actions and operators. Atomic actions are operations that are either completely evaluated or not evaluated at all. The operators determine in what order atomic action are evaluated. Some operators define a strict order, others define non-deterministic behavior. To illustrate the use of process algebra to model processes we give a small example of the finite recursive[3] specification of a stack that can push either 0 or 1 in basic process algebra, BPA [BW90]. First we define the atomic actions needed for our specification:

$push(0)$	Add a 0 to the top of the stack.
$push(1)$	Add a 1 to the top of the stack.
$pop(0)$	Delete a 0 from the top of the stack.
$pop(1)$	Delete a 1 from the top of the stack.

[3]The process algebra used for the ToolBus architecture does not allow recursive specifications.

For the example we need the following process algebra operators:

- $\cdot$ Sequential composition: first the left operand is evaluated, after completion the right operand is evaluated.
- $+$ Non-deterministic choice: either the left operand or the right operand is evaluated.

Now we can define the stack process:

$$\begin{aligned} S &= T \cdot S \\ T &= push(0) \cdot T_0 + push(1) \cdot T_1 \\ T_0 &= pop(0) + T \cdot T_0 \\ T_1 &= pop(1) + T \cdot T_1 \end{aligned}$$

Unlike what is the case for an algorithm[4], a process does not always have to terminate after a finite number of steps giving some result as output. This is also the case in our example, the recursive specification has no termination condition. The specification begins by telling that a stack S process begins with a process T followed by the recursive execution of stack process S. T is a process that either may push the number 0 onto the stack followed by process T_0 or it may push the number 1 onto the stack and behave like process T_1. Processes T_0 and T_1 pop their respective values of the stack or proceed with pushing another value on the stack followed by popping their value of the stack. One of the properties of our model is that it is not possible to pop an empty stack, which is often a source of errors in a design of a stack. If we can make a model of a design of a stack and prove that it is equivalent to the above model we have the guarantee that this error can not occur.

2.3.2 Atoms

In process algebra atoms are abstract concepts. In the stack specification the atoms were given a concrete role of which the meaning was intuitively clear. In the ToolBus atoms have a concrete function to perform that can have an influence on the state of the ToolBus or its environment. The functionality of atom however does not interfere with process algebra semantics. In our experimental ToolBus prototype we have implemented the following subset of the ToolBus atoms:

- Assignment.
- Process creation.
- Print functionality.
- Synchronous communication (send).
- Synchronous communication (receive).
- ToolBus shutdown.

This subset is sufficient to run a number of non-trivial testing scripts. The atoms are 'atomic' in the context of the ToolBus although their evaluation may use multiple processor instructions to complete. For the evaluation of an assignment atom it might be necessary to calculate some mathematical function before updating the target variable, this is considered part of the evaluation of the atom.

Two other important process algebra elements we encounter in **T** scripts are the constants δ (`delta`) and τ (`tau`). The constant δ represents the atomic step that once started never completes its action but idles forever. The effect of δ on a process expression can be seen in appendix B table B.2, axioms A6 and A7. The constant τ is treated as an atomic action that, when it is evaluated, always terminates successfully but which has no other visible effects.

[4] We use the definition from [Knu97a].

2.3.3 Operators

The ToolBus performs its coordinating task by specifying which tools may communicate and in what order this happens. However it is often not possible to compute statically which tools want to communicate in what order. Most of the time the behavior of tools is governed by events that are not under control of the ToolBus like user input or network events. In order to handle this unpredictability a number of process algebra operators contain a non-deterministic element. We have implemented all process algebra operators that are used in the C implementation of the ToolBus:

- $\cdot$ Sequential composition.
- $+$ Non-deterministic choice.
- $*$ Iteration or binary Kleene star: the left operand is evaluated zero or more times before the right operand is evaluated.
- $\lfloor\!\lfloor$ Left merge: interleaving parallelism, starting with left operand.
- $\|$ Free merge: interleaving parallelism.

The behavior of the operators follows from their action relations. A summary of the action relations is given in appendix B table B.1. An operator that does not occur in **T** scripts but plays an important role behind the scenes is the communication merge operator $|$. An essential part of the ToolBus, synchronous communication (between processes), is defined by the communication merge axioms CF1 and CF2. The axioms can be found in appendix B table B.2.

2.4 Special ToolBus features

Evaluation of atomic actions can have an effect beyond the change of the process expression they are part of. The effect of atomic actions can be grouped in the following categories:

- manipulation of data
- direct input/output
- ToolBus control
- tool interaction
- asynchronous communication

Although data can be integrated in the definition of a process algebra, μCRL [PVV94] is an example, this is not the case for discrete time process algebra used by the ToolBus. Incorporating data and datastructures in a process algebra makes it much more complex. Furthermore it makes the use the ToolBus programming language less flexible as all datatypes have to be defined explicitly. The purpose of the ToolBus interpreter[5] is not executing large calculations or performing operations on data but rather the controlled distribution of complex data from one tool to another. Despite the fact that it would be most elegant if no operations on data (terms) would be performed inside the ToolBus it proved to be very convenient to provide some capabilities to manipulate data. The operations on data take place in the assignment atom. For each type of term there are familiar operations, like arithmetic operations for the numeric types and logical operations for Boolean terms.

A quick and dirty way for the ToolBus to communicate with the outside world is using 'direct input' and 'direct output'. Direct input is read from the standard input and direct output is written by the ToolBus to the standard output of the operating system. For debugging purposes or simple status messages direct I/O suffices. More sophisticated I/O can be done using tools specially written for this purpose.

ToolBus control is meta-functionality like shutting down the ToolBus or monitoring the internal state of a process. Additional functionality could be defined like saving the state of a running ToolBus and starting a ToolBus using a saved state. Saving the state of a running ToolBus

[5] The ToolBus interpreter is the part of the ToolBus responsible for interpreting a **T** script.

configuration is not (yet) implemented in the C implementation of the ToolBus. For our prototype we have only implemented facility to shut down the ToolBus.

The two most important ToolBus features that have not been implemented in our prototype are interaction with tools and asynchronous communication between processes. Normally tools are used to provide a means of interaction with the user. In our tests all input is provided by **T** scripts and direct output is used to provide feedback. Asynchronous communication is implemented in the ToolBus using a messaging system called notes. ToolBus processes can state that they want to receive notes of a specific format by "subscribing" to them. When a process sends a note it is distributed to all processes that have subscribed to it and stored in a queue until it is processed. This gives processes the possibility to do low-priority communication. We have concentrated on synchronous communication because it is more challenging to implement than asynchronous communication.

2.5 Example of a ToolBus configuration

To test our prototype we have used several **T** scripts that are included in the C implementation of the ToolBus. We have used the scripts that tested the internal workings of the ToolBus interpreter and did not use tool interaction. As an example of a ToolBus configuration we will discuss one of the **T** scripts used for testing. We will start with a presentation of the **T** script language, followed by an explanation of the specific purpose of the example **T** script and we finish with describing how the script is interpreted by a ToolBus interpreter.

2.5.1 T script syntax

Because a complete syntax definition of **T** scripts is available in [BK95] we present the syntax of **T** scripts in a rather informal way. We have restricted our definition to the features of the ToolBus that we have implemented. Words shown in a `fixed-width` font are reserved keywords.

A **T** script consists of the definition of a number of processes followed by an instruction to start evaluating a number of those processes in parallel:
`process` P_1 `is` *process_definition*
`process` P_2 `is` *process_definition*
...
`process` P_N `is` *process_definition*
`toolbus`$(P_{i_0}, \ldots, P_{i_M})$
The definition of a process starts with the name of the process and possibly some formal parameters followed by a variable declaration and ends with a process expression:
`process` *Name opt_formal_parameters* `is` `let` *variable_declaration* `in` *process_expression* `endlet`
When no variables have to be declared the keywords `let`, `in`, and `endlet` do not appear. A process expression is built from ToolBus atomic actions, possibly with additional evaluation conditions, and process algebra operators. The following list shows the reserved ToolBus keywords relevant for our project:

- `process`
- `is`
- `let`
- `toolbus`
- `snd-msg`
- `rec-msg`
- `create`
- `printf`
- `shutdown`
- `delta`
- `tau`
- `if`
- `then`
- `else`
- `fi`
- `equal`
- `greater-equal`
- `less`
- `add`
- `mod`

The first column contains the keywords that are used to define processes and define which processes have to be started in parallel. The second column shows the keywords of atoms that we have implemented. Some atoms that we have implemented, assignment and process call, have no

keywords. In the third column we encounter the two process algebra constants and the keywords that are used to define conditional evaluation of atoms. The last column shows the keywords that can be used to define a Boolean condition and keywords for arithmetic operations within an assignment atom.

2.5.2 An example T script

As an example of a ToolBus script we consider the problem of finding all prime numbers smaller than some specified number, max, using the sieve of Eratosthenes. The sieving algorithm works as follows:

1. Start with the number $n = 2$.
2. Print prime number n.
3. Mark all multiples m of n with $m \leq max$ as non-prime.
4. Assign to n the next number smaller or equal to max that is not marked non-prime. If there is no such number we are done.
5. If $n^2 \geq max$ we are done else go to step 2.

Some small variations on the algorithm exist, for example in [Knu97b] only odd numbers are considered.

The **T** script that implements the sieve of Eratosthenes tests the creation of ToolBus processes and the communication between them. Below is the script that we used for testing our implementation. The only difference between this script and the original script from the ToolBus distribution is that our simpler print atom, which just prints terms and appends a newline, has no formatting capabilities.

```
process SIEVE(Max : int) is
  let N : int, Fid : int
  in
     create(FILTER(2), Fid?) .
     N := 3 .
     if less(N, Max) then snd-msg(2, N) . N := add(N, 1) fi *
     delta
  endlet

process FILTER(P : int) is
  let Z : int, N : int, Fid : int
  in
     printf(P) .
     ( rec-msg(P, Z?) .
       if equal(mod(Z, P), 0) then tau
       else
         create(FILTER(Z), Fid?) .
         ( rec-msg(P, N?) .
           if equal(mod(N, P), 0) then tau
           else snd-msg(Z, N) fi
         ) * delta
       fi
     ) * delta
  endlet

toolbus(SIEVE(400))
```

When this script is interpreted the line

```
toolbus(SIEVE(400))
```

instructs the ToolBus interpreter to start the process `SIEVE` with 400 as actual parameter. We can describe the structure of the process expression of `SIEVE` as[6] $create \cdot assign \cdot block_{condition} * \delta$. When *create* is evaluated it creates a new instantiation of a `FILTER` process with as argument the value 2; *assign* is an assignment atom, assigning 3 to variable `N`.

From axioms A6 and I it follows that *block* will be repeatingly evaluated. If *block* will actually be evaluated depends on *condition* which in this case is $N < Max$. If *condition* evaluates to the Boolean value *true* *block* will be evaluated else it becomes equal to δ and process `SIEVE` idles forever. *block* consists of two atoms that are sequentially evaluated, a synchronous send action and an assignment atom that adds 1 to `N`. The atom `snd-msg` is one half of a communication action, in order to complete its action it needs to communicate with a matching receive atom. We will return to the matching of communication atoms later on. Summarizing, *block* sends messages of the form $(2, M_i)$, with M_i ranging from 3 to 399 included.

The process `FILTER` has one formal parameter. When it is created its first action is to print the value with which is was created. So obviously only `FILTER` processes should be created with a prime number as actual parameter. The first `FILTER` process is created by $atom_1$ of `SIEVE` with actual parameter value 2. After the print statement there are two loops in a nested construction:

$(receive \cdot (create \cdot block * \delta)_{condition}) * \delta$

inner loop

outer loop

What might be counter-intuitive, if we rely on our experience with imperative programming languages, is that the loops are evaluated consequetively. While *condition* is not true we keep doing *receive* actions. And when *condition* is true we evaluate *create* and start looping *block* indefinitely. So after we start evaluating *block* we won't return to *receive*.

The receive atom can only communicate with a send atom in another process that has matching arguments. Communication can be used to synchronize processes and to transfer values from one process to another. In table 2.1 we give a few examples of matching send and receive atoms.

send atom	receive atom	result
`snd-msg(2)`	`rec-msg(N)`	If $N \neq 2$ there is no communication.
`snd-msg(2)`	`rec-msg(2)`	Successful communication.
`snd-msg(2, 3)`	`rec-msg(2, Z?)`	Successful communication, variable Z gets value 3.
`snd-msg(2, 3)`	`rec-msg(3, Z?)`	No communication.

Table 2.1: Communication between send and receive atoms

The function of *condition* is to discard all (non-prime) multiples of the prime value of the process, i.e. the value with which the process was created. Because the numbers received by *receive* are ordered from low to high the first value that is received that can not be divided by the prime number of the process is also a prime number. This number is then used by *create* to create a new `FILTER` process. Because each `FILTER` process can only identify one prime number it can only create one new filter. So after it has created a new filter a process starts evaluating *block*, which does not contain a create atom. *block* has the same filtering function, it checks if a number that is received is a multiple of its own prime value or not. However, if a possible prime number is received it does not create a new filter but it communicates the value to the `FILTER` process it has created. This way a chain of filters is created as depicted in figure 2.3.

[6]The $*$ operator binds stronger than ".".

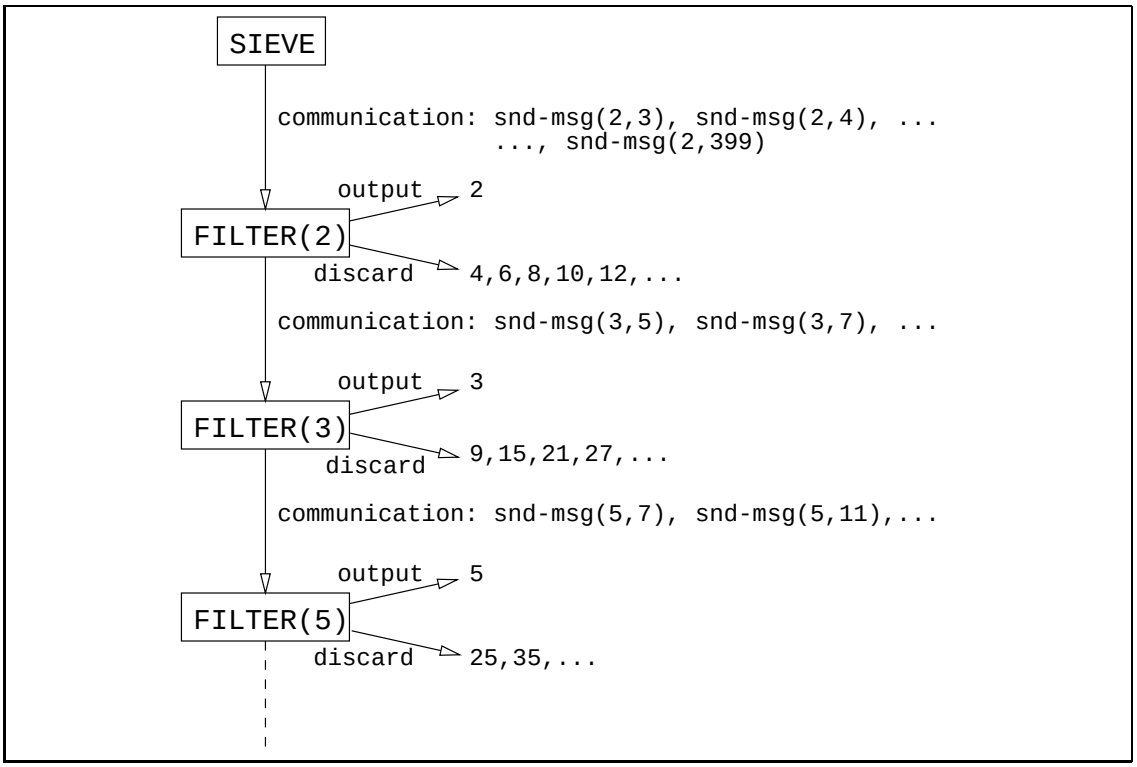


Figure 2.3: Chain of filters

The algorithm is slightly different from the sieving algorithm described above. First of all it does not check if the square of the last prime number found is greater than the maximum number that has to be checked. Second, the algorithm does not halt properly. After the last value is send by process `SIEVE` and checked for being prime the ToolBus interpreter should be halted with a shutdown command. However, these are only cosmetic differences.

Chapter 3

A Java Framework for the ToolBus

Now that we have described the theoretical foundations of the ToolBus we can proceed towards the design and implementation of the prototype. Due to the scope of our project we were not able to implement all of the ToolBus features available in the C implementation. The focus of our project lies with the internal operation of the ToolBus, the interpretation of a **T** script according to its process algebra semantics. Fortunately the ASF+SDF specification of the ToolBus has a number of modules that are optional. That makes it possible to define a specific ToolBus just by including (or excluding) certain modules. The optional modules are:

- the iteration operator "$*$" from process algebra
- the free merge operator "$\|$" from process algebra
- variable declaration with the let-construction
- expressions
- conditionals
- creation of processes
- synchronous communication between processes using messages
- asynchronous communication with notes
- discrete time
- delay and timeout
- tool definitions
- connection and disconnection of tools
- execution and termination of tools
- evaluate and do-commands
- events
- monitoring
- reconfigure
- tool control processes

Some modules depend on others ("delay and timeout" depends on the discrete time module) and inclusion of one implies inclusion of the other, but most optional modules are independent. In our Java prototype we have implemented iteration, free merge, variable declaration, expressions, conditionals, creation of processes and synchronous communication. While our design too allows to make some features optional it is not possible to attain the same flexibility as the ASF+SDF specification. One of the reasons is that not all optional modules are on the same level of abstraction. For example, the inclusion of discrete time in our implementation can not be done without changing a large number of classes at different places of the class structure.

Using a retrospective view we recognize the following important stages in the development of the prototype:

- The design of the top-level ToolBus structure.
- Definition of the parser.

- The design and implementation of the abstract process algebra features.
- Design and implementation of atomic actions.

Beforehand we had the idea that we could design and implement our prototype in two separate phases, the first one would be concerned with the top-level structure that represents the general structure of a ToolBus configuration and the second one was concerned with the process algebra elements that are used to specify process behavior. The parser plays the role of man-in-the-middle because it interacts with both parts of our design. However some abstract concepts like process expressions and their normal forms could not be ignored and their design and implementation falls between that of the top-level structure and the process algebra primitives. So we needed another design phase.

Following common principles of object-oriented design we started with identifying the elements of the ToolBus that could be translated into a class. There are different ways in which a decomposition can be made of the ToolBus into different classes, each with its own consequences for the implementation. For the first design and implementation phase we used the description of the ToolBus implementation framework from the ToolBus report [BK95] . In the second phase of the design we chose for a detailed decomposition in which we recognized the individual process algebra concepts like atomic actions and operators. This gave a rich class structure which also added some complexity to the design not found in the C implementation. During the implementation of the atomic actions and operators the need was recognized to define classes for higher level structures like process expressions. So the development cycle of the process algebra elements had to be interrupted by the design of the higher level process algebra concepts. As it turned out, most complexity of our implementation originated from the mutual dependencies between the low-level and the high-level process algebra elements. And it is hard to make a sharp distinction between these two levels development cycles. There was no added complexity due to unexpected interaction between our top-level ToolBus structure and the process algebra features.

Building the first part of the parser, which recognizes a syntactically correct **T** script, was relatively straightforward. The syntax of **T** scripts is completely defined by the ASF+SDF specification of the ToolBus. Furthermore we could reuse parts of the Lex and Yacc definitions that were available from the C implementation. Adding the action rules was more difficult because that was the part of the parser that had to connect with our design and implementation. However, it was possible to follow a top-down approach by first defining the parser for the top-level structures of **T** scripts and working out the smaller parts at a later time.

In section 3.1, we will describe the parser which is a relatively independent part of our ToolBus prototype. The top-level design is described in section 3.2 showing the parts of the ASF+SDF specification from which it was derived. In section 3.3 we describe the general design of process expressions and their normal form called *action prefix normal form*. Because it is difficult to discuss the action prefix normal form without any references to more detailed process algebra concepts we have included the discussion of the design and implementation of the process algebra operators. In the final section of this chapter, section 3.4, the atomic actions and their supporting features are described as they are implemented in our prototype.

3.1 The T script parser

Our first intention was to not build the parser and build our example ToolBus configurations by hand. After some discussion Paul Klint presented convincing arguments why a proper **T** script parser should be included in the project. Without a parser all examples have to be build by hand. This is an error prone process and the examples tend to be small and simple. Having a parser makes it possible to test much larger process expressions.

Before we started building the parser we already had a partly working ToolBus prototype. During implementation of the parser it proved necessary to modify the ToolBus implementation. This resulted in a more sophisticated ToolBus prototype that was not working anymore. This situation continued until the parser was finished. Ultimately it can be said that the implementation

of the parser was the driving force of the implementation of our **T** script interpreter. About 60%-80% of the changes in our initial design of the ToolBus resulted from the development of the parser. Many of these changes were corrections of design errors. Another advantage of developing the parser has been the possibility of using larger examples to test our implementation. No implementation of a software system should be considered complete without a set of validation tests.

For creation of the **T** script parser we have used the JavaCC tool[1]. It is a tool that resembles the well known scanner and parser generators Lex and Yacc [LMB92]. An experiment with a JavaCC generated parser for ATerms showed that it was approximately ten times slower than a hand-crafted parser. These performance characteristics are not of any influence on our implementation because parsing only takes place before starting the ToolBus interpreter. Our experience with JavaCC is that it is an easy to use tool, especially when one is familiar with Lex and Yacc, and Java.

After we finished our initial implementation of the top-level classes we started building the parser. As the basis of our scanner and parser definition we used the Lex and Yacc files from the C implementation. First we tried to make a direct translation from the C parser generator. This was very easy because the syntax used by the two parser generators is very close. As an example we show the syntax definition of a term and a term list. On the left-hand side the Yacc definition, on the right-hand side the JavaCC definition.

```
term:
    INT
  | REAL
  | STRING
  | var
  | result_var
  | IDENT
  | IDENT '(' term_list ')'
  | '[' term_list ']'
  | '<' type '>'
  ;

term_list:
    /* empty */
  | term
  | term ',' term_list
  ;
```

```
void term():
{ }
{   <INT>
  | <REAL>
  | <STRING>
  | var()
  | result_var()
  | <IDENT> ( "(" term_list() ")" )?
  | "[" term_list() "]"
  | "<" type() ">"
}

void term_list():
{ }
{
  ( term() ( "," term_list() )? )?
}
```

When we wanted to add the action rules we encountered some problems that were due to fundamental differences in our approach and the C implementation. In the C implementation **T** scripts are converted to a single term and all parts of **T** scripts are represented by terms one way or another. For example we have the Yacc definitions of a communication action, the assignment atom, and a conditional expression.

```
SND_MSG '(' term_list ')'
NAME ASSIGN term
IF term THEN proc ELSE proc FI
```

In each of these parse rules terms are used in a different context in which they can represent different things. The argument of a send atom for example can contain an identifier, a Boolean variable, and a string value. The right-hand side of an assignment expression always represents some kind of expression of the same type as the variable on the left-hand side. And in the conditional process expression, `term` always represents some Boolean expression. One of our design decisions is the extensive use of the object oriented features of Java. In our decomposition

[1] This parser generator was originally developed by Sun Microsystems. It is now distributed and supported by the company Metamata and can be downloaded from http://www.metamata.com.

of the ToolBus in classes we ended up with different classes for variables, Boolean expressions, constant values. This had its influence on the parser because we had to create different objects where in the Yacc parser only terms are created. The definitions we use instead of (the direct translation of) the Yacc definitions are[2]:

```
<SND_MSG> "(" term() ("," term() )* ")"
<NAME> <ASSIGN> value_expr()
<IF> bool_expr() <THEN> proc_expr() <ELSE> proc_expr() <FI>
```

We see that the definitions for the assignment and conditional expression use different non-terminals where previously only terms were used. The rule for assignment uses the construction `value_expr()` that creates an object that returns a value of some type when it is evaluated. The construction `bool_expr()` is more restricted and returns an object that returns only Boolean values when it is evaluated. Maybe a bit surprisingly the send atom is defined in the same way as in the Yacc definition even though it uses terms in the most various ways. The reason for this is that for our specific implementation of the send atom extensive processing of the arguments is necessary. The arguments of communication atoms are parsed by the action rules using hand-crafted Java code. The construction of communication atoms is described in more detail in subsection 3.4.5.

Even after redesigning the parser to match our Java class structure our parsing troubles where not over. The process call feature, which is actually a macro mechanism, required elaborate transformations on the created parse tree. This included substituting parts of the parse tree by other trees and renaming multiple elements of parts of the parse tree. Because of our dedicated datastructure we had to build several rather large tree traversal functions just to provide for a single ToolBus feature. This was one of the times the term rewriting facilities of ASF+SDF were sorely missed. The expansion of process calls is described in subsection 3.4.7.

3.2 The top-level design

The design of the top-level structure of our ToolBus prototype is derived from following ASF+SDF description of a ToolBus configuration:

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

This description contains all the elements that we have used to build the framework of our implementation. We will start from the top (λ_{BS}) to the lowest level (AP_{ij}) describing the meaning of the element in the ASF+SDF specification and how it fits in our design.

The top-level ASF+SDF operator λ_{BS} represents the global state of the ToolBus, the bus state. This state contains information like the current time, the last process identifier, and the last tool identifier. Because of the differences in the approach we have taken towards the implementation we have no direct equivalent for the λ_{BS} operator. The top-level class of the ToolBus is the `TBInterpreter` class which must keep track of all processes defined in a **T** script and is responsible for the post-processing of the process expressions that the parser produces. However we have chosen to put the emphasis on the process algebra elements instead of the interpreter and therefore we move as much information as possible from the interpreter downwards to its processes and even further down. So the `TBInterpreter` class contains little dynamic information that is needed for the actual evaluation of a **T** script.

The E_{Script} operator contains the **T** script that is being executed. In the ASF+SDF specification it is necessary to keep the **T** script available in the case a new process has to be created. We wanted to remove this level which in our view contained mostly redundant information. The useful information of a **T** script after it has been parsed are the process expressions of processes that are dynamically created during evaluation of the **T** script. In most cases that is just the process expression of one or two processes. That information might as well be put in the global

[2]Note that the argument of the send atom uses an alternative definition of a term list.

bus state although the bus state has a dynamic character and E_{Script} is rather static. In fact in the ASF+SDF specification E_{Script} is always used together with the bus state. Because of our decision to put as little information as possible in the bus state we had to put the information in lower level classes.

Below E_{Script} we find the renaming operators $\rho_1, \ldots, \rho_m$. The renaming operators make it possible to link each atom occurring in a process expression with a unique process, so in a way they represent virtual processes. We have defined a class `TBProcess` to represent actual processes. This approach seems more suitable for creating a multi-threaded implementation than using virtual processes. It is very natural to associate a ToolBus process with a single thread of execution. Introducing threads on a higher level does not make sense because on the level above there is only a single interpreter. Using threads on a lower level can be too costly because thread creation is expensive and the lower execution levels are very dynamic and require frequent thread creation.

In the `TBProcess` class we store the information contained in E_{Script} that we might need for dynamic process creation. A consequence we had not foreseen was that by doing so we have mixed the dynamic and the static character of process expressions. When a process is evaluated its process expression changes with every step. However, every time a process is dynamically created it should be initialized with the original process expression. So we cannot put the information of E_{Script} in a process and just consume it because we might need it in its original state. A solution could be to keep a copy of the initial process expression in a process apart from the evaluated expression. This means some overhead in memory use when multiple instances of a specific process are created because we only need one original process expression but each instance contains one. More problematic would be the removal of a permanently idle process from memory because we might need the original process expression it contains. To prevent these complications we added a new layer representing the dynamic part of a process: the process instance. A process implemented in the class `TBProcess` contains the static process expression, called `processDefinition`, and is created only once for each process defined in a **T** script. A process instance contains the changeable copy of the process expression. Multiple process instances can be created by copying the process expression from the fixed version of the process. To represent instances of processes we use the class `TBProcessInstance`. Each process instance contains all the information that is necessary for evaluation. Besides a process expression it contains a storage class for the value of its variables λ_{Env_i}. The association between the top-level classes of our ToolBus design is depicted in figure 3.1.

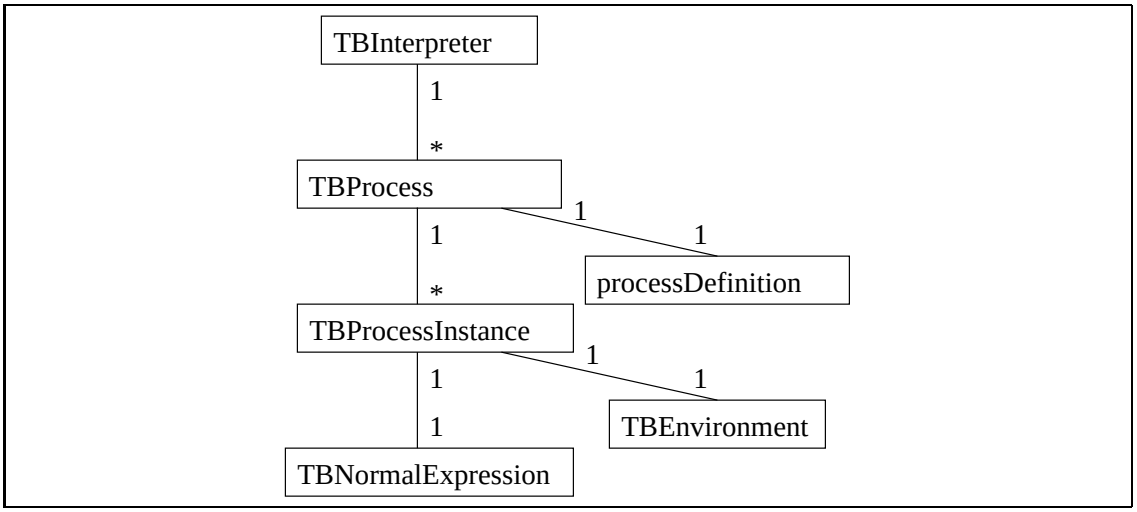


Figure 3.1: The top-level class associations

The classes `TBInterpreter`, `TBProcess`, and `TBProcessInstance` form the high level structure of our Java ToolBus design. The class `TBNormalExpression` is not a part of the top-level structure

but part of the intermediate level. It is shown here because it is the equivalent of the action prefix expressions in the ASF+SDF ToolBus characterization above. The design and implementation of action prefix expressions will be discussed in section 3.3.

The C implementation of the ToolBus and our implementation use a different strategy for interpreting a **T** script. Both implementations have a main loop that keeps the interpreter running as long as there are atomic actions that can perform an action. But that is where the similarity between the two interpreters ends. In the C implementation a **T** script is first transformed into a term. The elements of this term are then used by the interpreter as passive identifiers to determine what action it has to perform. After an atomic action is executed it is the interpreter that initiates the required normalization of the process expression. In our implementation the only responsibility of the interpreter is to select a random process to perform an action. A process in its turn selects a random process instance to perform an action. In this way the responsibility for executing an atomic step trickles down until it reaches the bottom of the ToolBus structure where the objects representing individual atoms reside. The idea behind this approach is that by spreading out the complexity between different classes there would be no single class which would be very complex.

3.3 Process expressions

The behavior of a process is determined by its process expression which consists of atoms and operators. In process algebra the atomic action is used as an abstract notion, the specific function of an atomic action is disregarded most of the time. The only thing important about an atom is the fact that it can perform some unspecified action after which the structure of a process expression changes. More important are the operators that determine the structure of the process expression and thereby the behavior of the process. The situation is more equal in the ToolBus, with emancipated atoms performing actions that can result in a change of state of the process instance to which they belong. In our development we have disregarded this fact at first and implemented a framework to represent abstract process expressions as they are defined in process algebra. For the design and implementation of process expressions we used the process algebra constants τ and δ as dummy atoms so we could test our implementation. This approach worked well until we had to implement the more complex atoms. This is discussed in section 3.4.

When considering the way process expressions are constructed, it is not hard to imagine that a binary tree is a natural representation, with the operators as parental nodes and the atoms as the leaf nodes. The most important function of the process expression framework is the correct implementation of the process algebra axioms as shown in table B.2. This is discussed in the first part of this section. The second part of this section discusses the special normal form used for evaluating process expressions.

3.3.1 Process expression operators

The `TBOperator`*Name* classes are the subclasses of `TBExpression` representing the process algebra operators with the same name or character representation. The operators are defined in the following classes:

- `TBOperatorStar` representing the iteration operator "*"
- `TBOperatorDot` representing the sequential composition operator "."
- `TBOperatorPlus` representing the non-deterministic choice operator "+"
- `TBOperatorLeftMerge` representing the left merge operator "$\lfloor\!\lfloor$"
- `TBOperatorFreeMerge` representing the free merge operator "||"

Despite what their name might suggest ToolBus operators do not operate directly on their operands. At least not in the same way as arithmetical operators do. One can even say that

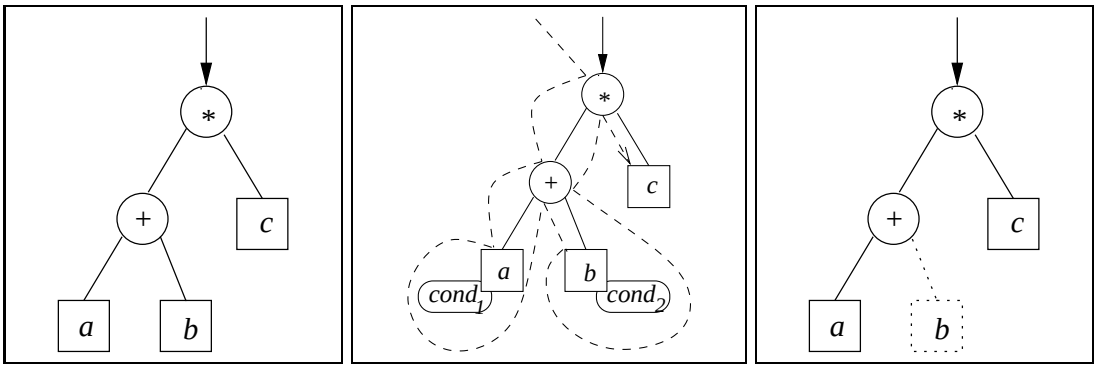


Figure 3.2: Process expression tree of $(a + b) * c$

the operands have more influence on the operators than the reverse. The main function of an operator is to determine which of its operands may perform an action and to define how to transform the process expression tree after one of its operands has finished a single atomic action.

Not every process expression is equally suitable for evaluation. Consider the process expression $(a + b) * c$. It is not immediately clear what atom may be executed. In figure 3.2a the tree representation of the process expression is depicted, the root of the expression tree is marked with a solid arrow. When the expression is evaluated one of its atoms has to be selected for execution. Which atom gets selected depends in the first instance on the operators appearing in the tree. Because the process expression tree has a rather unfavorable form it is possible that there are many steps necessary to select an atom.

In figure 3.2b an example trace is shown of the selection process. Selection of the atom to evaluate starts at the root of the tree, the $*$ operator gives a non-deterministic choice between evaluating its left operand (possibly multiple times) or evaluating its right operand. Suppose the choice is made to evaluate the left operand then the + operator has to be evaluated. From table B.1 Rel4 and Rel5 we have that either one of the operands of "+" that can perform an action may be evaluated. In this example the non-deterministic choice of evaluating atom a has been made but because condition $cond_1$ could not be met the evaluation fails. So we have to backtrack and try the other operand, atom b. Unfortunately condition $cond_2$ associated with b can not be met either so we have to backtrack to the root of the tree so we can select c. All the backtracking increases the complexity of the implementation and has a negative influence on performance.

Another problem with arbitrary process expressions is determination of the resulting process expression after one of its atoms has completed its action. Suppose that atom b was selected and has finished its action, see figure 3.2c, it is not clear how the new process expression must be determined without backtracking to the root of the expression tree. Fortunately using the process algebra axioms process expressions can be transformed into an equivalent action prefix normal form. This representation of a process expression facilitates the choice of atom to evaluate and after evaluation the transformation into the new normal form. Normalization is implemented by the method `normalize` in classes representing ToolBus operators and atoms. Using axioms I and A4 we can transform the process expression tree from figure 3.2a into the tree depicted in figure 3.3a. The following code is the implementation of axiom I in `TBOperatorStar`.

```
public TBNormalExpression normalize() {
  TBNormalExpression normalExpr = leftChild.normalize();
  normalExpr.seqCompose(this);
  normalExpr.addNormalExpression(new TBNormalExpression(rightChild));
  return normalExpr;
}
```

We start with creating, recursively, a normalized expression (`TBNormalExpression`) of the the left sub-expression of the iteration operator "$*$". Then the current process expression (with the itera-

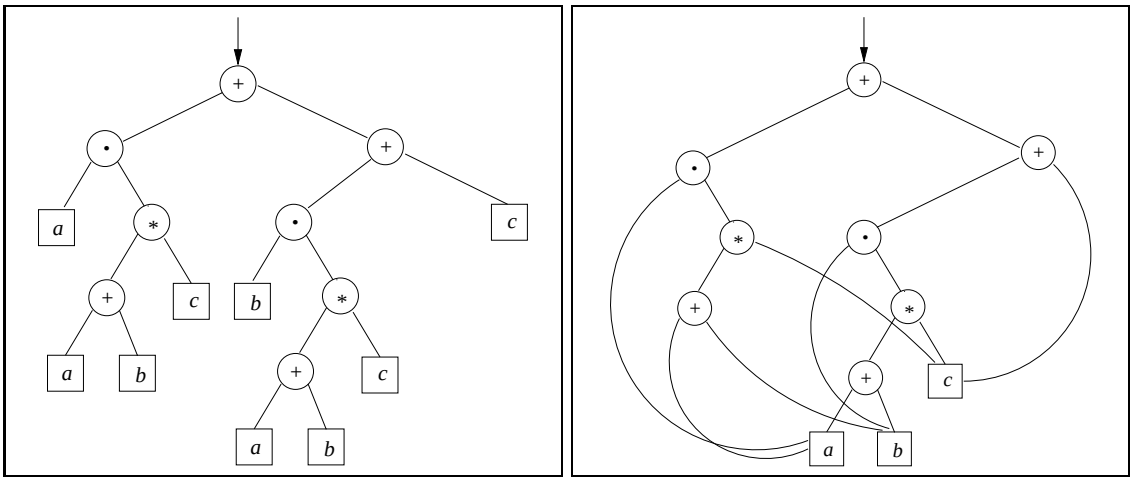


Figure 3.3: Process expression in action prefix normal form

tion operator as root) is added to the normalized left sub-expression using sequential composition (the sequential operator "."). At last the right sub-expression is added to the normal expression using the choice operator "+". The result is shown in figure 3.3a.
The cost of transforming the process expression into its normal form is (apart from the transforming process) an expansion of the expression. Although the expansion might seem considerable when the process expression trees of figure 3.2 and figure 3.3a are compared the atoms can be shared and do not have to be recreated. The actual situation in memory would resemble figure 3.3b. So by sharing the atoms we save a lot of (object creation) time and memory.

The two most important advantages of representing a process in action prefix normal form are that it is easier to select an atom to evaluate and that the process expression remaining after evaluation can easily be determined. One problem with process expressions in action prefix form is that the number of action prefixes is not immediately known. This is important if we want to select one prefix atom at random. So in order to exploit the advantages of the action prefix normal form we designed a specific class for it.

3.3.2 Action prefix normal form

A process expression in action prefix normal form consists of the choice composition of one or more action prefix expressions that start with the sequential composition of an atom and an arbitrary process expression. So we are dealing with expressions of the form $a_0 \cdot P_0 + a_1 \cdot P_1 + \ldots + a_n \cdot P_n$. In our ToolBus prototype objects of the class `TBNormalExpression` represent action prefix normal forms.

In the class `TBNormalExpression` we use the variable `actionPrefixTable` to store the different action prefix expressions. We use the `java.util.Vector` class to represent `actionPrefixTable`. The `java.util.Vector` class behaves like an array that can grow (and shrink) dynamically. Figure 3.4 shows the normalized form of expression $(a+b)*c$. Because a `Vector` knows its own size and it is possible to select arbitrary entries it is now easy to pick a random atom to evaluate. When the `eval` method of a `TBNormalForm` object is called there are different strategies to handle it. A simple and effective one is to create a random number r between 0 and the size of `actionPrefixTable`. The prefix of the process expression in the table with index r is evaluated. Because we have given atoms and process algebra constants a great deal of autonomy `TBNormalExpression` has very little information about the internal workings of the atoms. The only information returned by an atom is the status of its evaluation. After successful evaluation all action prefix expressions in the entries that were not selected are discarded and the expression of the selected prefix is normalized. We will return on the evaluation status of atoms in sections 3.4.1 and 3.4.8.

In contrary to what is suggested in section 3.3.1 a process expression in action prefix normal

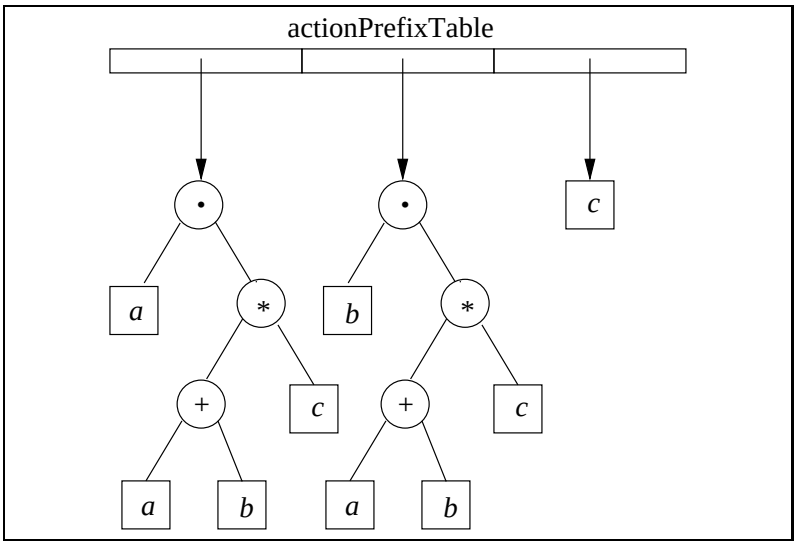


Figure 3.4: Data structure for storing action prefix expressions

form can not always be normalized directly without some additional constraints on the normal form. The process expression $a.b.c$ demonstrates this. This expression has two equivalent action prefix expressions as shown in figure 3.5. When atom a is evaluated only the second expression

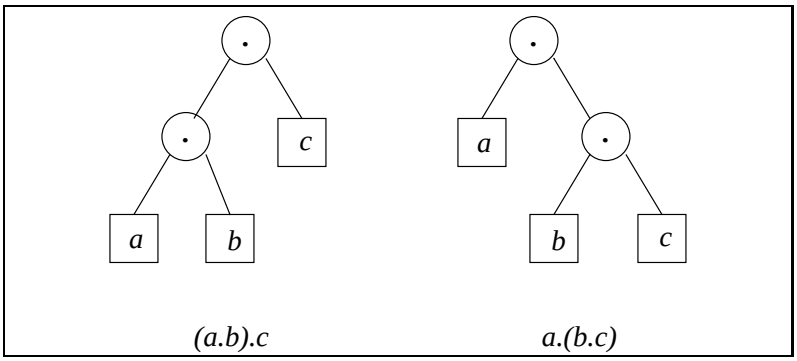


Figure 3.5: Two equivalent action prefix expressions

can be normalized in one step by selecting the "$\cdot$" operator with operands b and c as root of the new process expression. The first expression has to be cleaned-up first by attaching atom b to "$\cdot c$". After normalization the control is returned to the calling method.

3.4 Atomic actions, conditions and values

On the lowest level of our top-down design we find the atomic actions, constants, and data facilities of the ToolBus. The implementation of atomic actions forced many changes in higher levels. Most of the time, additional functionality was needed to support the atomic actions and sometimes a structural change had to be made to the design. The extent of the necessary changes was somewhat of a surprise considering the abstract and rather neutral nature of atoms in process algebra.

The names of the classes of ToolBus atoms follow the same scheme as those of the operators. Each atom is implemented in a class called `TBAtom`*Name*, where *Name* is one of `Printf`, `Shutdown`, `Receive`, `Send`, `Assign`, `ProcCall`, `Tau`, or `Delta`. The purpose of an atom is to perform an action

that changes the state of the process expression, the process instance, or the external environment of the ToolBus. The evaluation of an atom i.e. the performing of its action is influenced by its evaluation condition. Below we will describe the implementation of the process algebra constants, the different atoms, and the evaluation conditions.

3.4.1 ToolBus constants

The two process algebra constants `tau` and `delta` are implemented in the classes `TBAtomTau` and `TBAtomDelta`. The successful evaluation of `tau` and `delta` has no effect on the state of the ToolBus other than a change of the process expression tree. The evaluation of `tau` is completely neutral for the external and internal environment of the ToolBus. When timing constraints are present this could be different, but the implementation of `tau` is so simple that its evaluation time will not have much influence.

In general when an atom is evaluated it returns a result status to its calling class, which is normally[3] `TBNormalExpression`. When no conditions are present an atom returns the constant value C_s after evaluation denoting that is was evaluated successfully. The caller can conclude from this information that the process expression of which the atom is a part can be normalized. The successful evaluation of a `delta` constant has a different effect on the process expression. After evaluation of a `delta` a process expression should not be normalized, but idle indefinitely. This should be arranged by `TBNormalExpression`. To be able to distinguish the evaluation of `delta` the constant value C_d is used to indicate that the evaluated atomic action was `delta`.

3.4.2 I/O atoms

In the original implementation of the ToolBus there are two atoms provided for direct input and output. We have provided a limited implementation of the output atom in the `TBAtomPrintf` class. There is no implementation of the input atom `read` in the prototype, instead we provide all the data we need in our **T** scripts. A `TBAtomPrintf` atom prints the arguments with which it is initialized to the standard output.

3.4.3 ToolBus control atoms

The atom `TBAtomShutdown` prints a message to the standard output and quits the ToolBus interpreter. In our prototype there are no tools connected to the ToolBus, hence there is no extensive clean-up to do.

3.4.4 Assignment, variables and value expressions

The main purpose of the ToolBus is the coordination of tools through the manipulation and control of information flows. Most of the time it just distributes values unmodified between tools. It is the task of the individual tools to perform operations on these values. There are times however when it is convenient to perform some basic operations on variables and values in the ToolBus. This is done using the assignment atom `TBAtomAssign` which associates a value with a variable. Values can be represented by constants or by more complex expressions. In the **T** scripts and the ASF+SDF specification we find the following concepts related to assignment:

- variables
- values
- functions producing values
- a type system
- an environment to associate values with variables

[3]The only exception is the evaluation of communication atoms which we will describe in more detail below.

The ASF+SDF specification represents variables using terms of the form $Name\$Process : type$ and result variables are represented by $Name\$Process : type?$. A variable in itself is nothing more than an identifier within a specific context that has a type associated with it. Values can originate from tools or be defined in a **T** script. In the ASF+SDF specification terms are used to represent values. Data coming from a tool is transformed into a term by a ToolBus adapter, a term sent to a tool is transformed by a ToolBus adapter into a data representation native to the application. Manipulation of data inside a ToolBus is done with only a limited number of functions that can be grouped into logical functions, arithmetic functions and list operations. The functions operate on terms representing a value of a proper type (e.g. the logical function `and` expects two Boolean terms) and produces terms as result. Assignment of a value to a variable results in the creation of a variable-value pair. This variable-value pair is stored in an environment which keeps track of all assignments in a specific context.

We have approached the handling and manipulation of data by focusing on value expressions. Value expressions[4] are simply expressions that represent a value of some type. The advantage of this approach is that we can place variables, values and functions all in the same part of the class structure. As it turns out this part of the class structure has no interdependencies with classes other than those defining value expressions and environments. All interaction with other classes takes place using a minimal class interface. The top-class of the value expressions is the abstract class `TBValue`. It is used to represent a value of an arbitrary type. As part of the design we used the class structure to distinguish between the types of value expressions. The abstract subclasses `TBValueBool` and `TBValueInt` of `TBValue` represent Boolean and an integer values. The concrete classes that produce values when being evaluated are for the Boolean type:

- `TBValueBoolAnd` implements the Boolean and-operator.
- `TBValueBoolNot` implements the Boolean negation.
- `TBValueBoolLess` compares two integers.
- `TBValueBoolConst` is used to represent the Boolean constants `true` and `false`.
- `TBValueBoolEqual` compares two integers.
- `TBValueBoolGreaterEqual` compares two integers.
- `TBValueBoolVar` implements the Boolean variable.

We did not need a Boolean or-operator for the **T** scripts which we used to test our prototype, but it would not be hard to extend the class with more functions. The subclasses of `TBValueInt` producing values are:

- `TBValueIntAdd` implements the integer addition.
- `TBValueIntMod` implements the modulo operator.
- `TBValueIntConst` is used to represent an integer constant.
- `TBValueIntVar` implements the integer variable.

A partial inheritance relationship is shown in figure 3.6. For the Boolean values a number of representative subclasses are shown, the subclasses of `TBValueInt` have been left out for conciseness. The *italicized* names represent abstract classes and methods.

Although the interface between the value classes and the other parts of the ToolBus is quite clean the use of value expressions is not without its problems. These problems result from the way we have designed the value classes. On the one hand the design results in an implementation with a clear structure. The abstract superclasses make it possible to use value expressions without specifying exactly what type they are and what kind of expression they represent. As an example of the benefits of the class structure we have a piece of code from the implementation of the assignment atom:

[4] We use the term value expressions to distinguish this type of expression from process expressions.

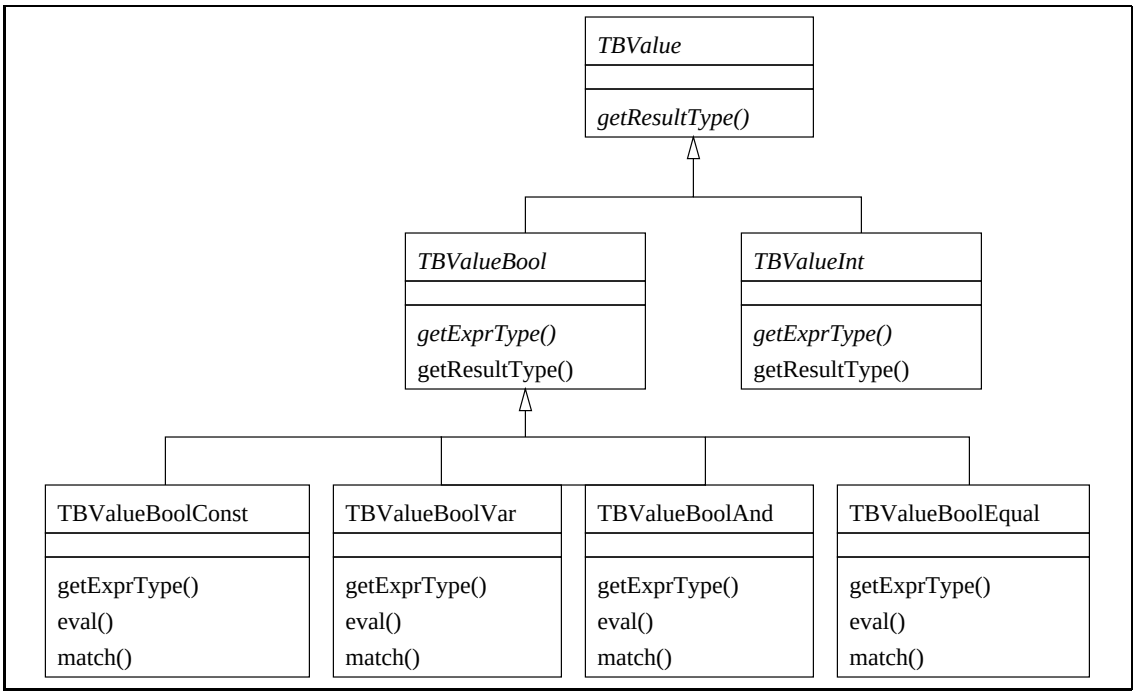


Figure 3.6: Inheritance relationship for value expressions

```
public class TBAtomAssign extends TBExpression {
  private TBValue       variable;
  private TBValue       value;

  public TBAtomAssign(TBValue v, TBValue val) {
    variable = v;
    value    = val;
  }
}
```

We use the abstract class `TBValue` to represent a variable `v` of an arbitrary type and a value `val` of an arbitrary type. This has the advantage that the creation of an assignment atom is very general. On the other hand is the use of an elaborate class structure in some situations a bit awkward. If we want to make a copy of an assignment atom we have to cast the abstract objects to their proper class so we can call their own `copy` method. In the following code from the class `TBAtomAssign` we see a part from the `copy` method. The copy method creates a new instance of an assignment atom with exactly the same arguments. In the example code below we see that making a copy of the variable requires casting `variable` to a variable object of the right result type.

```
  public TBExpression copy(TBProcess proc) {
    TBValue varCopy = null;
    if (variable.getResultType() == TBValue.BOOLEAN) {
      varCopy = ((TBValueBoolVar)variable).copy();
    } else
      if (variable.getResultType() == TBValue.INTEGER) {
      varCopy = ((TBValueIntVar)variable).copy();
    }
```

This is only the start of the trouble, the mere tip of the virtual iceberg. We also need to copy the value of that is assigned to the variable and the trouble is that we do not know what kind of

value expression `value` has. In the following example we see just part of the code that is needed to copy the value part.

```
TBValue valCopy = null;
if (value.getResultType() == TBValue.BOOLEAN) {
  switch(((TBValueBool)value).getExprType()) {
  case TBValueBool.CONSTANT :
    valCopy = ((TBValueBoolConst)value).copy();
    break;
  case TBValueBool.VARIABLE :
    valCopy = ((TBValueBoolVar)value).copy();
    break;
  case TBValueBool.FUNC_AND :
    valCopy = ((TBValueBoolAnd)value).copy();
    break;
  case TBValueBool.FUNC_EQU :
    valCopy = ((TBValueBoolEqual)value).copy();
    break;
  case TBValueBool.FUNC_GEQ :
    valCopy = ((TBValueBoolGreaterEqual)value).copy();
    break;
  case TBValueBool.FUNC_LESS:
    valCopy = ((TBValueBoolLess)value).copy();
    break;
```

This is only a part from the code necessary to copy a Boolean argument and there is another part for the case that we deal with an integer argument. We could use the type information gathered from the parsing phase to improve the situation. This leads to different implementations of assignment atoms for different types. However this does not solve the problem of dealing with the different expression types (variable, value, function).

3.4.5 Communication

Among the most important features of the ToolBus are the communication primitives. The communication facilities of the ToolBus can be divided into those used for communication with tools and those for internal communication between processes. A further subdivision of the internal communication can be made by separating synchronous from asynchronous communication. As the scope of our project was limited by time we were not able to implement all types of communication atoms. Because we did not implement the tool facilities of the ToolBus like tool control processes and adapters there was not much use for the external communication atoms to be implemented. So the decision was made to implement the atoms responsible for synchronous communication between ToolBus processes.

An essential part of communication is the data format that is used between the communicating partners. For our prototype to be useful it should at least in principle be able to communicate with external tools. The ASF+SDF ToolBus specification uses terms for external and internal communication and information flows naturally between atoms for external communication and internal communication. The same should be possible in our design, even if we used a different approach for exchanging information between ToolBus processes. Because the desire to approach the ToolBus with a fresh view we have not used the Java term library. Nevertheless, the difference between the way our ToolBus prototype and the original ToolBus implementation transports information is small. However, our implementation has not as much features and is not as flexible.

If we search for the origins of the synchronous communication atoms in process algebra we find axioms CF1 and CF2 (see the overview of the process algebra axioms on page 50). Axiom CF1 says that when the function γ is defined ($\downarrow$) for the two atoms a and b then γ is applied to a and b when they communicate. The axiom CF2 denotes a failure to communicate saying

that if the function γ is not defined for atoms a and b than the communication between the atoms results in a deadlock. In the ToolBus the function γ is only defined on atoms `snd-msg` and `rec-msg`. So communication can only happen between a `snd-msg` atom and a `rec-msg` atom. The behavior of the ToolBus differs slightly from the axioms. Consider the following two **T** scripts.

```
process P is
  printf("hello\n") . delta

toolbus(P)
```

```
process Q is
  printf("hello\n") . snd-msg(a)

toolbus(Q)
```

When the script on the left is interpreted the ToolBus prints "hello" on the standard output and then exits. But when the script on the right is interpreted the ToolBus prints "hello" on standard output and then remains active although nothing happens. There is no communication defined between atom `snd-msg(a)` and any other remaining atom because there are no remaining atoms so it should be interpreted as `delta` which is not the case. This is only a small deviation from the process algebra semantics and we have not found any strange behavior resulting from this deviation.

The interpretation of function $\gamma(a, b)$ is essential to define the behavior of the ToolBus with respect to communication. Axiom CF1 is very general, the only property of γ that follows directly from it is the synchronization of atomic actions because γ is applied to a and b simultaneously. This property is one of the major implementation difficulties because send and receive atoms can influence the state of two different processes at the same time.

In the ToolBus γ is defined as a matching function between the atoms `snd-msg`(L_1) and `rec-msg`(L_2). A send and a receive atom match only if the two atoms occur in different processes and if their argument lists L_1 and L_2 match. The argument lists, consisting of terms, match if $L_1 = t_{11}, t_{12}, \ldots, t_{1n}$ and $L_2 = t_{21}, t_{22}, \ldots, t_{2n}$ and for all $i = 1 \ldots n$ we have t_{1i} and t_{2i} match. Basic terms like Booleans, integers, strings, and variables match if they are of the same type (e.g. a Boolean value matches a Boolean value) and if they represent the same value. Two function applications match only if their function names are exactly the same and if all their arguments match pair-wise. The matching of a result variable is used to exchange values between processes. A result variable matches with any term of its type and the matching results in the binding of the value of the term to the result variable. Some examples of matching send and receive atoms (occurring in different processes) are:

- `snd-msg(5)` and `rec-msg(5)`
- `snd-msg("aap", "noot", "mies")` and `rec-msg("aap", "noot", "mies")`
- `snd-msg(a)` and `rec-msg(a)`
- `snd-msg(A)` and `rec-msg(B?)`

In the last example `A` and `B?` should have the same type. The result of this action is that after the communication action is completed the environment of variable `B` is updated with the binding of variable `B` to the same value as variable `A` has in its own context.

To facilitate efficient matching of arguments in communication actions we have used a binary encoding for the terms used in send and receive atoms. When the parser encounters a communication atom in a **T** script it encodes its parameters first and then creates a `TBAtomSend` or `TBAtomReceive` object using the encoded arguments as creation parameters. The encoding of terms is done using the following method. The argument list of a communication atom is split in two parts, one part describing the structure of the argument list and another part containing the values occurring in it. The encoding we used for the structural elements of terms are listed in table 3.1.

Because a communication atom always contains an argument list with one or more arguments the encoding always starts with 0. The elements of the argument list are parsed from left to right, depth first. Our test scripts contains only function applications with one character as identifier so we use the ASCII value of the identifier as function identifier. In a more complete implementation

element	$byte_0$	$byte_1$	$byte_2$
list	0	number of arguments	
function application	1	function name	number of arguments
integer value	2		

Table 3.1: Encoding of terms

a symbol table should be built associating all unique function names occurring in function applications with a unique value. Another restriction in our implementation is that we only use integer values in communication actions, however it is easy to extend the encoding when more types of values are used. The encoding of the values serves as a placeholder for the actual values that are stored in a separate list that is created by the parser. We give a few examples of the encoding of arguments of communication atoms in table 3.2.

communication atom	encoded arguments	value list
snd-msg(a)	0x0001016100	empty
snd-msg(g, X, 50)	0x00030167000202	integer variable, 50
rec-msg(g, 40, Y?)	0x00030167000202	40, integer result variable
rec-msg(h, g(40, 50))	0x0002016800016702020 2	40, 50

Table 3.2: Examples of encoding of terms

Using the values (here represented in hexadecimal base) of the encoded structure it is easy to perform a static check to determine which communication actions can possibly match. For example we see that the second and the third communication atom potentially match because the value of their encoded structure is the same. Whether two atoms really match can only be determined by comparing their value lists during interpretation of a **T** script. The second and third atom match during interpretation when the value of variable `X` in the environment of the send atom equals 40 else they do not match.

A special feature of our implementation of communication atoms is that each send atom keeps a link to all receive atoms with which it has a static match. In the same way does each receive atom keep a link to all send atoms with which it has a static match. This has the advantage that matching communication atoms can be found quickly. Because communication is the heart of the ToolBus and checking all other atoms to find a match can be slow it is important to find a match as quick as possible. The administration of matching atoms also has a disadvantage. The creation of new process instances becomes more complex, this will discussed in subsection 3.4.6. Another problematic issue concerning communication atoms comes from the fact that their influence is not restricted to the process instance to which they belong. Evaluation of a send atom has an effect on the state of its local process instance as well as on the state of the process instance of the receive atom with which it matches. The change in the state of a process instance can be two-fold. First there is the change of the process expression due to the evaluation of an atom and succeeding renormalization. The axiom for communication CF1 dictates that this happens atomically in the two processes that are communicating. Second, the definition of the matching function makes it possible to communicate values between process instances and thereby changing the state of the environment. We have implemented synchronous communication in such a way that only the send atom can initiate a communication action. This asymmetrical execution model may seem in contradiction with the definition of γ in the ASF+SDF specification, but in practice no problems arise from it. It has the advantage that the implementation stays smaller than when both communication atoms contain the same functionality.

3.4.6 Process creation

As we recall from the description of the top-level architecture of our prototype the process level is responsible for the creation and management of process instances. A process object keeps a list

of all active instances of the process it represents. When a new process instance is created it is added to the list, when a process instance can not perform any action anymore it is removed from the list. The `TBProcess` class contains the method `createInstance` for the creation of a new process instance, this method takes as argument a list containing the actual parameters of the process instance. New process instances are created by the ToolBus interpreter when starting the interpretation of a **T** script or by a process creation atom. While the context differs the creation procedure itself is executed in exactly the same way.

The process creation atom is implemented in the class `TBAtomCreate`. A completely initialized object of this class holds a reference to the process it has to create an instance of. Additionally it has a list containing the parameters with which to create a new process instance. When the atom is evaluated the parameter list is transformed into a list containing values. The following procedure describes the evaluation:

1. The `createInstance` method of the process is called with the value list as argument.
2. The process creates a new environment and adds the variables from the formal parameter list with the actual values.
3. It makes a copy of its (normalized) process expression.
4. It creates a new process instance using the new environment and the copy of the process expression.
5. The newly created process instance is added to the list of process instances a process keeps and the operation is completed.

The procedure is depicted in the following figure.

Things are more complicated when a new process instance is created and its process expression contains communication atoms, because they have to be registered with other, matching, communication atoms (see subsection 3.4.5). In figure 3.8 a part of the process instance creation procedure is depicted. This is a detailed view of the third step shown in figure 3.7 and it shows where the creation of a process instance with a communication atom differs from the creation of a process instance without a communication atom. The first action in the figure (3.1) is the copying of a single receive atom. This results (3.2) in a new `TBAtomReceive` object initialized with the same operation conditions and the same list of references, `sendList`, to matching send atoms as the original receive atom. In the example of this figure we assume that the receive atom has just one matching send atom. The copy of the receive atom now registers itself with the matching send atom (3.3). In the matching send atom a new reference (3.4) to the copy of the receive atom is created. This concludes the copying of the receive atom and the copy is returned to the method requesting it. A similar procedure is followed for the copying of send atoms.

Although the procedure seems quite straightforward it adds difficulties to the creation of a threaded implementation of the ToolBus. In a threaded implementation it would be logical to assign a thread to each process and have each thread/process run independently of the others. When, in a threaded implementation, two different instances of processes are created that contain matching communication atoms and two matching atoms are created at the same time the reference lists might not be completely updated unless this action is synchronized.

3.4.7 Process call

The process call is implemented as a macro mechanism that facilitates isolating and reusing specific functionality in **T** scripts. As we saw in our example of the wave equation a process call can be used in the same fashion as a procedure or function call. In the ASF+SDF specification process calls are expanded by replacing the call with the definition of the called process including declaration of local variables. We have used a similar approach in our implementation although we do not replace the call with the complete process definition, we use only the process expression. On a more detailed level more implementation differences can be found. In the ASF+SDF definition

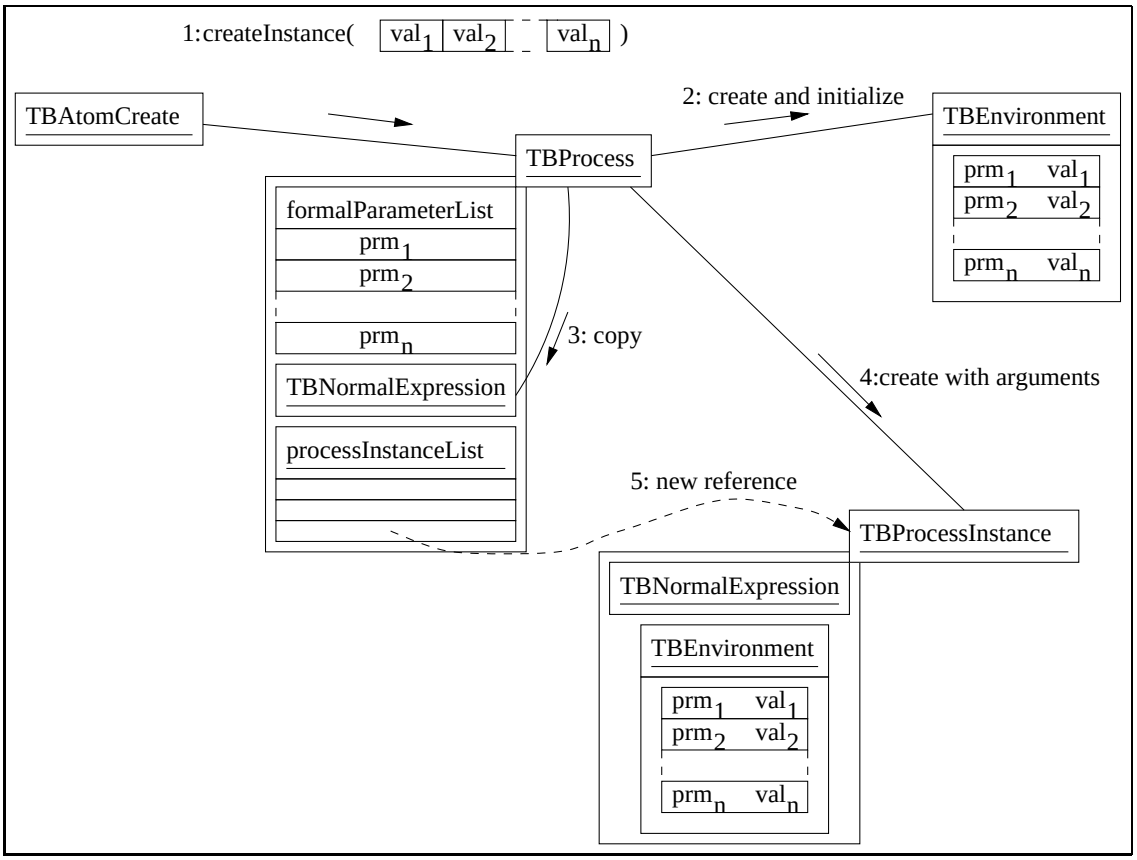


Figure 3.7: Creating a new process instance

a process call inside the context of another process gets its own environment for variables. The reason for giving an expanded process call its own environment is the prevention of a mix-up between variables when two or more calls to the same process are done in parallel. A potential source of trouble is illustrated by the following code from one of the **T** scripts we used for testing the prototype.

```
process E(N : int, X : int, Y : int) is
  let XX : int
    in XX := Y . CHK(N, XX, Y)
  endlet

process F is
  E(100, 3, 4) || E(101, 3, 5) || E(102, 3, 6)
```

We see here that process `E` is called in parallel three times from the same process. The process `CHK` of which the definition is not shown here checks equality between its second and third argument. If those arguments are not equal `CHK` prints an error message to standard output and terminates the ToolBus interpreter. When we expand the calls of process `E`, process `F` is transformed into[5]:

```
process F is
     ( XX$E:int := 4 . CHK(N$E, XX$E, 4) )
  || ( XX$E:int := 5 . CHK(N$E, XX$E, 5) )
```

[5]The expansion of process E is not exactly the way it is defined in the ASF+SDF specification, it was simplified to enhance the clarity of the example.

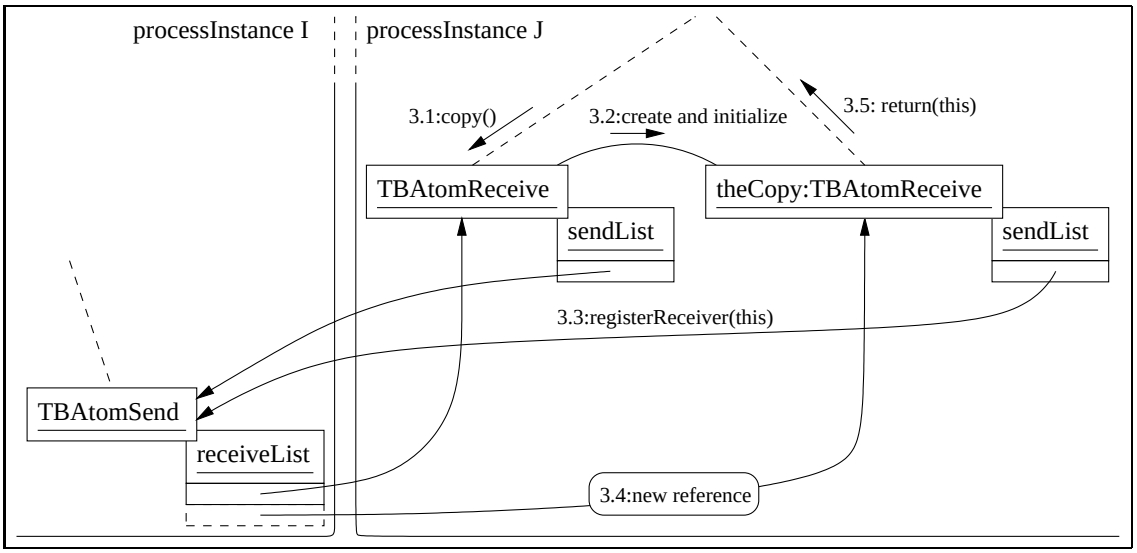


Figure 3.8: Updating references of communication atoms

```
|| ( XX$E:int := 6 . CHK(N$E, XX$E, 6) )
```

Because the process call is not evaluated as an atomic action its evaluation can be interleaved by the evaluation of other atoms. So one of the possible evaluation sequences of `F` might start with `XX$E:int := 4`, `XX$E:int := 5`, `CHK(N$E, XX$E, 4)`. This would result in an error message produced by `CHK` due to the failure of the test `4 == 5` and is not what we expected. The use of separate environments for variables in the expansion of each process call prevents this because the assignment `XX$E:int := 4` modifies a different environment than assignment `XX$E:int := 5` and the call `CHK(N$E, XX$E, 4)` uses the same environment as `XX$E:int := 4`. In the Java prototype this problem is solved in a different way. Instead of providing a different environment for each atom we use a single environment in one process. The name clashes are resolved by renaming the variables of the called processes using a counter. The procedure for expanding a process is as follows. First a counter is initialized with the value 1. Then for each process its parse tree is traversed and its the process expression is checked for the occurrence of process calls. When a process call is found the process expression of the called process is copied. In the copy of the process expression variables are renamed by appending the character representation of the counter to the name of the process. Then this modified process expression is inserted in the place of the originating process call and the counter is increased by one. In the Java prototype the expanded version of process `F` will become:

```
process F is
     ( XX$E1:int := 4 . CHK(N$E1, XX$E1, 4) )
  || ( XX$E2:int := 5 . CHK(N$E2, XX$E2, 5) )
  || ( XX$E3:int := 6 . CHK(N$E3, XX$E3, 6) )
```

The resulting code behaves the same because we have created a virtual environment inside the environment of `F` for each variable. Until now we have conveniently skipped over the initialization of a process call. In our implementation the initialization is done by prepending the process expression of the called process by assignment atoms. During expansion of the first process call of process `F` the process expression of `E` will be transformed into:

```
N$E1  :=  100 . X$E1 := 3 . Y$E1 := 4 .
XX$E1 := Y$E1 . CHK(N$E1, XX$E1, Y$E1)
```

A special case is the use of a result variable which can be used to return a value to the calling process as in the example of the wave equation. When a result variable is used in a process call and the corresponding formal parameter of the called process is also a result variable the variable of the calling process replaces all occurrences of the formal parameter in the process expression of the called process. So any updates to the result variable in a process call are noticeable in the calling process and in the other process calls. As an example we have the following **T** script:

```
process UpDown is
  let I: int in
    I := 0 .
    Up(I?) * delta || Down(I?) * delta
  endlet

process Up(Num: int?) is
  printf("Up   1: %d\n", Num) .
  Num := add(Num, 1) .
  printf("Up   2: %d\n", Num)

process Down(Num: int?) is
  printf("Down 1: %d\n", Num) .
  Num := add(Num, -1) .
  printf("Down 2: %d\n", Num)

toolbus(UpDown)
```

The output from a specific run from this program starts with:

```
Down 1: 0
Up   1: 0
Up   2: 0
Down 2: 0
Down 1: 0
Up   1: -1
```

In the **T** script variable `I` is initialized with the value 0. The process `Up` increases the value of `I` and the process `Down` decreases the value of `I` with one. As we can see in the run of the **T** script process `Down` is called first, it prints the initial value of `I`. Next we see that process `Up` is called and it prints the value of `I` which has not changed. The following line of output shows that process `Up` has finished, but it does not print an increased value of `I`. We can infer from this that before the second `printf` atom of `Up` was evaluated both `Up` and `Down` changed the value of `I` although we can not say in which order this happened. So it is clear that evaluation of an atom in a specific processes can influence the state of another process.

3.4.8 Conditional evaluation

All atom classes implement the `eval` method that is called when the atom has to be evaluated. The `eval` method returns a value representing the evaluation status. This is necessary because unlike in process algebra, which has no explicit execution model, the evaluation of an atom by the ToolBus interpreter can (temporarily) fail. The success or failure of evaluation depends on the evaluation condition associated with the atom. The evaluation condition is a Boolean function that is evaluated in the context of a process instance when the `eval` method of an atom is called. When the Boolean function evaluates to `false` the atom does not perform any action and the eval method returns the constant C_f, when it evaluates to `true` the atom completes the action it is supposed to perform and returns C_s (see also section 3.4.1). Initially all atoms have the evaluation condition set to `true`.

An evaluation condition is modified by the `if-then-else` expression. The conditional expression `if C then` $AP_1 + \cdots + AP_m$ `else` $AP_{m+1} + \cdots + AP_n$ with AP_i a process expression in action prefix form is transformed into the expression $AP'_1 + \cdots + AP'_m + AP''_{m+1} + \cdots + AP''_n$. The action prefix expression AP'_i, $1 <= i <= m$, is the expression AP_i with as evaluation condition the logical and of the condition of AP_i and C. The action prefix expression AP''_j, $m + 1 <= j <= n$, is the expression AP_j with as evaluation condition the logical and of the condition of AP_j and the negation of C. Although the evaluation of condition and atom is a two-step procedure it is important that the evaluation of a condition and the evaluation of the atomic action it is associated with should be performed as a single atomic step inside the ToolBus. If this is not the case, the following code could produce undesirable results.

```
if not(equal(A,0)) then a := div(2, A) fi || rec-msg(A?)
```

The condition ensures that `A` is not equal to 0 so a division by A would give no errors. However the value of `A` can get updated to any value by the communication action including the value zero. So if the condition is evaluated the atom which it guards has to be evaluated immediately.

Chapter 4

Results

The concrete result of this project is a partial implementation of the ToolBus in Java following an object oriented approach. Many important features are lacking but the implementation is complete enough to run a number of **T** scripts that are used to test the C implementation of the ToolBus. The interpretation of the test scripts by the Java ToolBus prototype gives the same results as the "real" ToolBus so at least the parts that are tested by these scripts function as intended. The performance as measured in speed of execution of the Java prototype is considerably slower than the C implementation, even on a Sun workstation using the JIT-compiler of the Java2 environment. This was a bit disappointing, although our implementation was not really optimized for speed we had hoped that the specific implementation of communication atoms would improve on the implementation model used in the ASF+SDF specification. There are however a number of implementation inefficiencies that can easily be improved upon and may bring the speed of the prototype on par with the C implementation.

Other, less concrete, results of the project are the software engineering lessons learned. At the start of this project we started not exactly with a Zen-like, empty mind. Being familiar with the ToolBus architecture shaped our ideas about how a ToolBus should be implemented. This resulted in many preconceptions about the resulting implementation. As we recall from the introduction our choice of implementation language was partially based on the reasoning that Java offered many features that would be useful for our project like garbage collection, object orientation and threads. Only garbage collection is the feature that we really profited from. Object orientation did not really make the implementation of the prototype easier. Identifying useful objects in the ToolBus was very easy, especially after studying the ASF+SDF specification and the process algebra foundations of the ToolBus. But the complexity of the implementation was not hidden in the class structure but in the specific execution model we chose for the interpreter of **T** scripts. And that was a part of the design process in which object orientation was not of any help.

4.1 Overview of the implementation

We will now present an overview of the implementation of the Java ToolBus prototype. Instead of focusing on the details we will show how the different elements described in chapter 3 fit together in the complete implementation.

The different classes from which the ToolBus prototype is build are arranged in four different packages. The `parser` package contains the JavaCC generated parser and some auxiliary classes. The `interpreter` package contains the top-level classes of the prototype and has two subpackages, `interpreter.expression` and `interpreter.value`. In `interpreter.expression` the classes are found that are used to build process expressions. The classes in `interpreter.value` are used to represent functions, variables, and values, i.e. value expressions. In figure 4.1 the packages and their contents are shown. The arrows from the classes to the packages show which packages are imported by which classes. It seems logical that the `TScriptParser` class

imports all classes from all other packages because it has to build a parse tree containing elements from the **T** script it parses. These elements are represented by objects from the `interpreter`, `interpreter.expression`, and `interpreter.value` classes. The classes from the `interpreter` package naturally depend on the lower level constructs which follows from the import dependencies. The class `TBProcessCall` is a small auxiliary class which is only used once when the interpretation of ToolBus processes is started. The dependencies of the classes in the `interpreter.expression` package on the `interpreter` package are a bit inelegant. It is a consequence of delegating responsibilities to the level of individual atoms. Single atoms sometimes need information about processes, process instances, and normal expressions for their proper operation. The classes in package `interpreter.value` have no dependencies on classes in other packages.

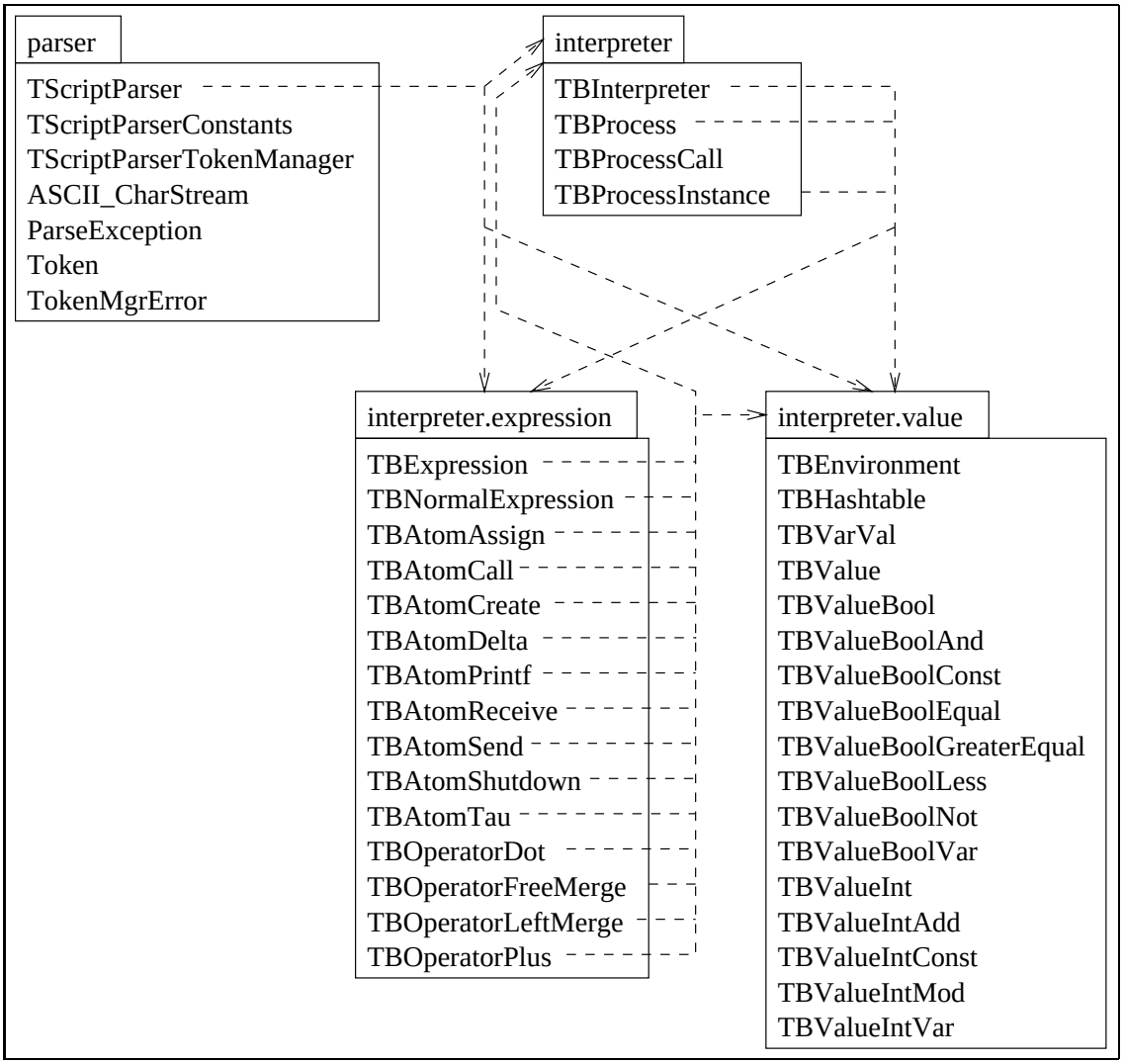


Figure 4.1: ToolBus Java packages with import relations

Within the packages `interpreter.expression` and `interpreter.value` there is another structure, the inheritance relationship between classes. In figure 4.2 the two inheritance trees present in the Java prototype are shown. The purpose of the class `TBExpression` is to build expression trees. An expression tree can of course be build from objects of a single class that represents a node of a tree. But the operators and atoms from which the process expression tree is build have specific places in the tree and very different functionality. Internal nodes of a tree can only

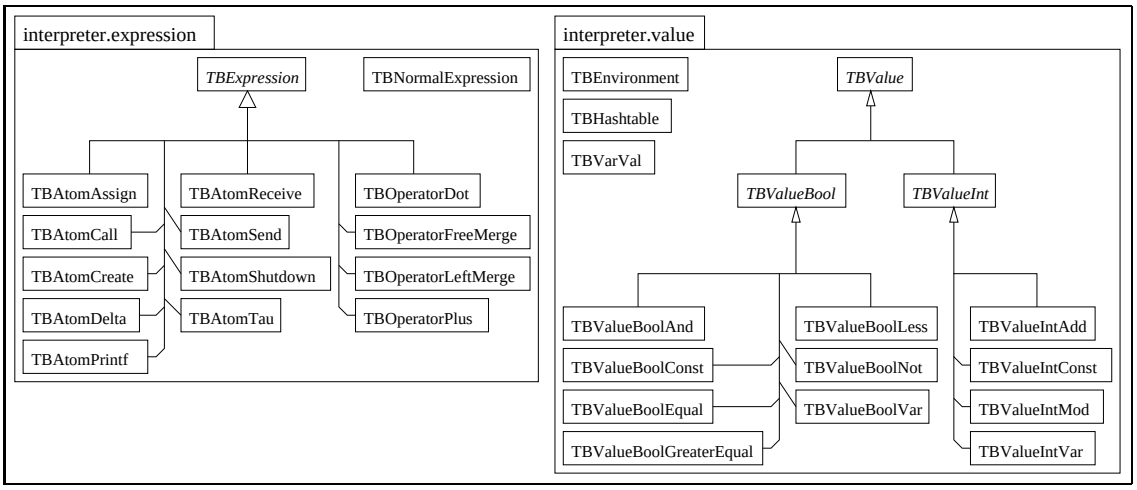


Figure 4.2: Inheritance relations

represent operators while leaf nodes can only represent atoms. At first, to emphasize their different roles in process expression trees, we created a single abstract class, `TBExpression`, with two abstract subclasses `TBAtom` and `TBOperator`. One subclass represented atoms and the other subclass represented operators. Concrete classes representing the different atoms inherited from the abstract atom class, concrete operator classes inherited from the abstract operator class. This approach resembles the same inheritance relationship as is used for value expressions. However the extra layer of abstract classes did not make our design much clearer to understand and made our implementation more complex so we decided to remove the abstract atom and operator subclasses. For a discussion of the inheritance relations between classes in the `interpreter.value` package were refer to subsection 3.4.4.

The evaluation of a **T** script is initiated by the a `TBInterpreter` object but the responsibility for performing the atomic actions lies with the individual atoms. In figure 4.3 the roles of the different objects during interpretation is shown. The `TBInterpreter` selects a process at random that may perform an action, after selection control is deferred to that process. The responsibility of a `TBProcess` object is to select one of its process instances at random and to instruct it to evaluate itself. A process instance, `TBProcessInstance`, just passes control to its (only) normal expression. In step 4 the `TBNormalExpression` selects one action prefix atom from multiple action prefix expressions and calls its `eval` method. The atom performs its actions and returns control to the normal expression, step 5. The evaluation of an atom in step 4 can fail if its operational conditions, set by an `if-then-else` construction, are not met. We have handled this situation by evaluating all prefix atoms in order until one evaluates successfully. After successful evaluation the `TBNormalExpression` object normalizes (step 6) the rest of the process and control is returned to the higher layers until it reaches `TBInterpreter` and the cycle is completed. This evaluation cycle is repeated until no actions can be performed anymore or until a shutdown atom is evaluated.

As might be expected after reading the previous part of this implementation report, the evaluation cycle for communication atoms is a bit different. Communication atoms as well as creation atoms have a non-local influence. For creation atoms the additional complexity is reasonably small, but it is worth elaborating a bit on the evaluation of a communication atom. As we recall from subsection 3.4.5 in our implementation of the ToolBus only a send atom can initiate a communication action. When a send atom is evaluated it uses its list of matching receive atoms and tries all receive atoms until it finds a suitable atom to communicate with. It calls the `eval` method of a receive atom with as arguments the two-part encoding of its term argument, its own environment and a reference to its process instance. The receive atom checks that the arguments really match and that its own process instance is different from the process instance of the send

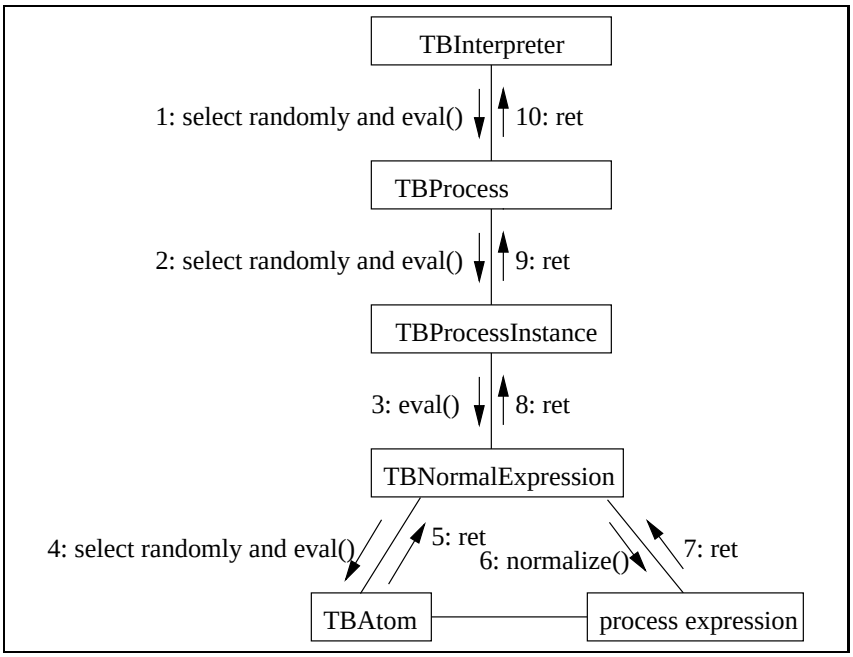


Figure 4.3: A single interpretation step

atom. If those conditions are met the environments of the send atom and that of the receive atom are updated appropriately. This may seem like a reasonably elegant evaluation model for communication atoms but it has some problems. The first problem is that an atom in general does not know if it is an action prefix or not. The second problem is that a receive atom does not know how to normalize the process expression of which it is the action prefix. The source of the problems is that the evaluation of a receive is requested by another atom and not by its normal expression. To solve these problems we had to modify the `TBNormalExpression` class. When a process expression is normalized all receive atoms that are action prefixes are 'activated'. This means that they get a reference to their `TBNormalExpression` and that they get their index number in the action prefix table (see figure 3.4) of the `TBNormalExpression`. Using these values a receive atom can determine if it is an action prefix. And after evaluation a receive atom can call its normal expression with the request to normalize its process expression. A simplified version of the implementation is shown in figure 4.4 below.

4.2 Design and implementation decisions

Before we even started designing our prototype we already had an idea what the result should look like: at the end of the project we would have a working prototype that consisted of many independent elements, each with its own specific responsibility. This would facilitate one of the project goals, namely building a threaded implementation. Using the fact that ToolBus processes were implemented as independent objects, the transition of non-threaded implementation to a threaded implementation would just consist of replacing the main loop of the ToolBus interpreter by the starting of process threads that would run simultaneously. This proved to be harder to achieve than we had imagined due to the non-local effects of communication atoms.

Another thing that influenced our design was that we wanted to approach the implementation of the ToolBus in a way that was completely different from the ASF+SDF specification. The C implementation has many term rewriting characteristics inherited from the ASF+SDF specification. We wanted to let go of the term rewriting and see what would happen. The drive to do things different let to certain design choices that complicated our project and made parts of our

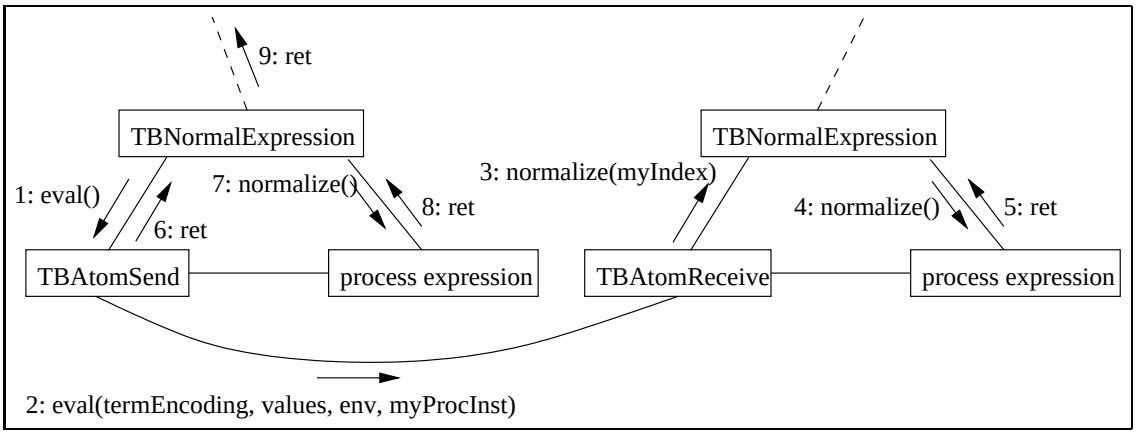


Figure 4.4: Successful evaluation of communication atoms

implementation a bit inelegant.

The resulting prototype was not exactly what we had in mind before we started. But even so we hope by describing our design choices and the consequences that this project will be useful for others who want to build a new implementation of the ToolBus even if it is just to point out the pitfalls we have encountered and sometimes fallen into.

4.2.1 Implementation language and tools

One of the first choices made was which programming language we would use to implement the prototype. For the actual implementation we used the following tools:

- Java Development Kit for Linux, version 1.1.6 (also called JDK), Blackdown port
- Java Development Kit for Solaris, version 1.2.1 (also called JDK2), Sun Microsystems
- JavaCC compiler generator, version 1.1, Sun Microsystems and Metamata company

The reasons for using these tools where the availability and the ease of use. The JDK is available for most platforms, although the newest versions are only available for Solaris and Windows platforms. The tools from the JDK have a very simple command line interface so that they can be used without having to get used to a specific user interface. The JavaCC compiler is very similar to Lex and Yacc, although specially designed for a Java development environment. Other compiler generators, like ANTLR, are also available but besides being more powerful they are more different from Lex and Yacc than JavaCC. The ToolBus does not require a very powerful parser so we used the simplest tool available.

One of the features of Java that was not mentioned as a motivation to choose it as implementation language, portability, was in fact very useful in our project. We developed on two different machine architectures, a PC running the Linux operating system and a Sun workstation running Solaris. The Solaris version of the Java Development Kit did run the prototype that was compiled with the Linux Java compiler without any problems. Due to the lack of a JDK version 1.2 for Linux we could not try the reverse situation but we were impressed with the portability of the Java byte code. The byte code produced by the lower version java compiler ran without problems on a higher version Java Virtual Machine on a different architecture. Due to a bug[1] in the JDK2 compiler it was not possible to compile our source code without some small modifications. But once the modifications where made in our source code it compiled without problems.

We had hoped that our implementation would not be much slower than the C implementation. Unfortunately it is quite a bit slower. On a Linux test machine the benchmark program `sieve.tb`

[1] The version of the Java compiler we have used did not parse empty comments correctly.

takes about 11 seconds to finish when evaluated by the ToolBus implemented in C while the Java prototype needs approximately 84 seconds. On a Sun SPARCstation the figures are 5 seconds for the C implementation and 24 seconds for the Java prototype. The better speed ratio is probably due to the Just In Time compiler in the Solaris Java environment. The difference in performance is not extremely bad especially when considering that many parts of our code are not optimized for speed. Furthermore Java has a disadvantage over C that it suffers some disadvantage because of the overhead of its virtual machine. However, the special way in which Java programs are executed makes it hard to compare the performance of our prototype with the C implementation. So to make an useful comparison between the performance of the C implementation and the performance of our approach Java is not the best language, C or C++ would be much better.

4.2.2 Delegating responsibilities

The design choice that has had probably the largest impact on the implementation of our Java ToolBus prototype was the specific class decomposition of the ToolBus interpreter and the assignment of responsibilities to each class.

In the C implementation all functions directly concerned with interpretation of a **T** script can be found in the file `interpreter.c`. The function `atomic_steps` loops through all processes and selects an atom to evaluate. Evaluation of an atom is delegated to the function `simple_atomic_step` which is basically a large switch statement with an action attached to each type of atom. In the next code fragment from the C implementation the implementations of the actions are edited for brevity.

```
static TBbool simple_atomic_step(atom *Atom)
  {
    switch(at_fun(Atom)) {
      case a_tau:
      case a_rec_note:
      ...
      case a_assign:
      case a_shutdown:
    default:
      return TBfalse;
    }
  }
```

All atoms are encoded as function applications with the function symbol determining the type of atom. A few atomic actions, like those for communication, are handled directly by the function `atomic_steps` because it has information about other processes available.

In our implementation we have identified all kinds of objects in the ToolBus like an interpreter, processes, tools, and atoms. We defined an abstract structure between the different objects, a hierarchical relationship with the interpreter on top and the individual atoms at the bottom. But instead of letting the interpreter control all the action inside of the ToolBus we have delegated much of the control towards the lower levels of the structure making each atom responsible for its own evaluation. While this approach works for many facilities of the ToolBus it is not suitable for all. The evaluation of communication atoms has an effect that is not restricted to their immediate surroundings. The impact of this was not immediately clear when we designed and implemented the other atoms, postponing the implementation of what proved to be the most complex atoms. The implementation of the other atoms was straightforward and we did not expect as much problems with the implementation of the send and receive atoms as we have encountered. By delegating most of the responsibility to the level of the individual atoms it becomes difficult to implement atoms that have a non-local influence. It took a lot of patching to let communication atoms handle their own evaluation.

Comparing the two implementations we see that except for the communication atoms they do not differ much. When we focus on the implementation of the communication atoms we see that

we have traded a more complex design for a more flexible way to communicate.

4.2.3 Matching of terms

Our alternative implementation of terms may seem superfluous when one considers that a mature term library already existed before our project started. The decision to make a new implementation was made because of the rather trivial desire to do things in a different way. As we already have shown, the representation of values in our ToolBus implementation might as well have been implemented using the existing ATerm class library. Still our alternative implementation has some benefits over an implementation using only ATerms.

Encoding the structure of terms separately from the parts denoting values leads to a compact way to represent terms and an efficient way to match them. We can take our idea further by recognizing that the structure of a communication term is static and that it can be determined at parse time. So instead of using a binary encoding to represent the term structure we can let the parser create a term structure table (just like a symbol table) and assign a single integer value to each unique term structure. If we revisit our example from subsection 3.4.5 we get the following encoding for the different terms.

communication atom	term structure identifier	value list
snd-msg(a)	0x0000	empty
snd-msg(g, X, 50)	0x0001	integer variable, 50
rec-msg(g, 40, Y?)	0x0001	40, integer result variable
rec-msg(h, g(40, 50))	0x0002	40, 50

Table 4.1: A more compact encoding of terms

We have used a two-byte value to represent term structure allowing for 65536 unique term structures, in all **T** scripts we have encountered this is more than enough.

A consequence of this alternative approach is that all tools must be compiled after the term structure table is created by the **T** script parser. And after (fundamental) changes in the term structure table all tools must be re-compiled. This is less flexible than the current implementation and may be a disadvantage. On the other hand the translation between native tool values and ToolBus terms becomes much easier.

4.2.4 Other design decisions

We give a short summation of some smaller design decisions in table 4.2 below.

design choice	**alternative**
Definition of a ToolBus process in the TBProcess class	Definition of a ToolBus process in the TBInterpreter class
Special class representing an action prefix form	Just one class representing process expressions.
Each atom has a reference to its environment	Passing the environment when an atom is evaluated.

Table 4.2: Some smaller design decisions

An immediate improvement on our current implementation would be the removal of a layer of our design while keeping the same functionality and complexity. We realized that this was possible after we finished our implementation. By moving the original process expression to the level of the interpreter we can let the interpreter handle the process instances and remove the level of processes (now implemented by `TBProcess`).

Another layer that can be removed is the one representing the action prefix normal form. This however can not be done without moving functionality to other places in our design.

A decision that seemed rather trivial when it was made was whether to give each atom a reference to its environment when it is created or to pass the environment in the `eval` method of an atom. Fortunately we chose to give each atom a reference to its environment. If we had not done this the evaluation of the receive atom would have been even more complicated to implement. The reason is that the evaluation method of the receive atom is not called by the object that manages action prefix normal expressions, but by a send atom from a different process instance. The send atom is not able to pass any other environment than its own, it knows nothing of the environment of the receive atom.

4.3 Bugs and non-intuitive features

During the design and implementation of our prototype we encountered situations where we found the behavior of the ToolBus odd or at least not intuitive. We branded one of those situations[2] as a bug in the C implementation while the implementation was in fact behaving exactly as specified in the ToolBus report. This was clearly due to a lack in our intuition. Still we found one situation in which the behavior of the C implementation deviated significantly from the ASF+SDF specification and another situation in that could be a bug in the implementation or another blind spot in our intuition. We will first discuss the "real" bug and then the "suspected bug".

4.3.1 Mismatch of result variables

When we implemented the matching of send and receive atoms we had to deal with the following situations that can occur in a **T** script:

- matching a value with another value
- matching a value with a result variable
- matching a result variable with a value

A question that had to be answered during implementation was: what happens when matching two result variables? In the C implementation the behavior of this communication action is not well defined although it is not prohibited. Consider the evaluation of the following **T** script. It can lead to two very different results.

```
process P is
  let A: int in
    A := 3 .
    snd-msg(A?) .
    printf("%d\n", A)
  endlet

process Q is
  let B: int in
    B := 2 .
    rec-msg(B?) .
    printf("%t\n", B)
  endlet

toolbus(P, Q)
```

Note that the printf atom of process `P` expects an integer and the printf atom of process `Q` a term. On the first run the output of this **T** script is:

[2] The last example of subsection 3.4.7 describing the interference between two process calls.

```
3
A$P:int?
```

If we assume that the assignment atoms are evaluated before the communication action is performed then `A` has the value 3 assigned and `B` has the value 2 assigned. During the communication action a matching is performed between `A?` and `B?`. Apparently the matching has resulted in the updating of the environment of process `Q` but not of process `P`. We know this because process `P` can only print integer values and the value 3 is printed. Furthermore the term `A$P:int?` is printed by process `Q` instead of the value 2. At another run the output resulting from interpreting this script is

```
2
ToolBus: Illegal type in printf '%d
', expected int, got var
```

Process `P` and process `Q` can both have printed the value 2. But the run-time error message was printed by process `P`, because only process `P` expects an integer argument in its `printf` atom. So apparently the value of `B` was not updated while the value of `A` changed to `A$P:int?`. This looks like an error, but to be sure we have to find out what the defined semantics according to the ASF+SDF specification.

The result of the communication between a compatible pair of send and receive atoms is the matching of their arguments. The matching function γ defines that matching two arguments arg_1, arg_2 can take place in either order $match(arg_1, arg_2)$ or $match(arg_2, arg_1)$. We have the following equation for the matching of two terms:

$$\frac{\begin{array}{c} T_1' = substitute(T_1, Env_1), \\ T_2' = substitute(T_2, Env_2) \end{array}}{match(T_1, Env_1, T_2, Env_2) = match1(T_1', T_2', nullEnvP)}$$

The equation for *substitute* rewrites to the value of the term in its environments. When no value for a term is defined in the environment *substitute* rewrites to the term itself. In our case $substitute(A\$P : int?, [A\$P : int \mapsto 3])$ rewrites to $A\$P : int?$ because $A\$P : int?$ and $A\$P : int$ are not exactly the same. For $B\$Q : int?$ the same is true. So the communication action `snd-msg(A?)|rec-msg(B?)` in our **T** script results in rewriting one of the following equations:

1. $match(A\$P : int?, [A\$P : int \mapsto 3], B\$Q : int?, [B\$Q : int \mapsto 2])$ under the condition $T_1' = A\$P{:}int?$ and $T_2' = B\$Q{:}int?$
2. $match(B\$Q : int?, [B\$Q : int \mapsto 2], A\$P : int?, [A\$P : int \mapsto 3])$ under the condition $T_1' = B\$Q{:}int?$ and $T_2' = A\$P{:}int?$

The relevant rewriting rule[3] for $match1$ is:

$$\frac{\begin{array}{c} [Entries_1] = env1(EnvP), \\ value(Var, [Entries_1]) = var, \\ Var = Vname : Type, \\ require\text{-}type(Type, T_2) = true \end{array}}{match1(Var?, T_2, EnvP) = ([Var \mapsto T_2, Entries_1], env2(EnvP))}$$

If we try to rewrite $match1(A\$P : int?, B\$Q : int?, ([], []))$ we get a mismatch of types. The equation $require\text{-}type(int, B\$Q : int?)$ rewrites to $false$. It is easy to see that the same holds for $match(B\$Q : int?, A\$P : int?, ([], []))$. This means that in the ASF+SDF specification two result variables do not match, even if they have the same type. So the send and receive atom in the **T** script behave not according to the ASF+SDF specification. Evaluation of the script should halt in a deadlock situation.

[3] Other rules are applicable too, but because of the symmetry of the arguments it is sufficient to consider just this one equation.

4.3.2 Confusing term semantics

In the C implementation of the ToolBus some terms have a special interpretation depending on the context in which they appear. For example the term `equal()` is considered a Boolean function when it appears in a condition. However it does not have a special meaning when it is used as an argument of an communication atom. Boolean values, like integer values, are special terms and their interpretation should be independent of their specific ToolBus context. Now consider the following **T** script:

```
process Send is
  snd-msg(f(10)) .
  snd-msg(g(false))

process Receive is
let
  I: int,
  B: bool
in
  rec-msg(f(I?)) .
  if equal(I, 10) then
    printf("number is equal\n")
  else
    printf("number is not equal\n")
  fi .
  rec-msg(g(B?)) .
  if equal(B, false) then
    printf("boolean value is equal\n")
  else
    printf("boolean value is not equal\n")
  fi
endlet

toolbus(Send, Receive)
```

The output when running this script is:

```
number is equal
boolean value is not equal
```

The first line of output is what we expect, an integer value is matched with an integer result variable. The value of this result variable is then compared to a value equal to that what was send and as expected it was found to be the same. The same procedure is followed with a Boolean value, but to our surprise the two Boolean values are not the same. What happens is that `g(false)` is not parsed as a function application with a Boolean value as argument, but as a function application with a identifier with value 'false' as argument. This behavior is a bit counter-intuitive.

4.4 Conclusion

To summarize the results of this project we can say that the ASF+SDF specification of the ToolBus allows for very different implementations. The advantage of having a formal specification is that it is easy to check if an implementation is behaving correctly. A downside is that the high level of abstraction of the ASF+SDF specification hides many complexities that one might encounter when building an actual implementation. In our project we have explored the possibility to create a ToolBus implementation that uses multiple execution threads using an object oriented approach. Due to unexpected complexity issues and time constraints we were not able to complete a multi-threaded prototype. On a positive note we picked up a few interesing ideas during design and

implementation that we like to mention. And we hope that these topics can be of interest for future development of the ToolBus.

Although process algebra, upon which **T** scripts are based, is primarilly concerned with parallel execution of processes one can wonder if building a parallelized implementation of the ToolBus is worth the effort. While it is certainly possible to build a multi-threaded implementation it is not certain that the benefits would outweigh the disadvantages. By distributing threads equally over multiple processors it is possible to increase execution speed by just adding an extra CPU. The possibility of automatic speeding up the interpretation of a **T** script simply by adding hardware to the computer system running the interpreter is very attractive. However allowing for different threads complicates the design and implementation of the ToolBus considerably. In large **T** scripts the synchronization of many different threads could become a major bottleneck which would nullify the performance increase that multiple processors provide.

The design and implementation of our prototype has shown us two directions for ToolBus development that might be of interest:

- refinement of the implementation of synchronous communication in the C implementation
- building a lightweight Java ToolBus interpreter.

We have not studied the C implementation of the ToolBus in detail, but we did look at the implementation of the communication atoms. Finding a matching communication atom is done by looping through all processes and all action prefixes of each process. Our design improves on this method by providing all communication atoms with a reference to matching atoms. A similar construction could be used to speed up synchronous communication in the C implementation.

One of our less successful experiments was the design and implementation of our alternative data format. It is very similar to the original ATerm format although it lacks features that the ATerm library offers. Especially when one considers that there is a binary ATerm format that uses a very compact encoding a less flexible solution is not desirable. What our experiment did show was that the data that is transported through the ToolBus is very independent of the rest of the ToolBus. We could use the original ATerm data format instead of our own data format with changes in only a few specific classes.

One of the features of Java which we did not take into account when selecting an implementation language for our prototype, portability, proved to be very convenient. Portability of Java byte-code is not without its problems especially when complex graphical tasks have to be performed. However in the case of the ToolBus only the most basic Java language features are used which generally do not give portability problems. As there is a general trend today for commercial software to support three types of computing platforms (server, desktop, and portable[4] computing) it may be interesting for the ToolBus to explore the same expansion path. As Java Virtual Machines have been developped for many handheld computing devices, a Java version of the ToolBus would be suitable for low-end computing devices.

[4]The term *portable* in portable computing refers to the small physical appearance of computing devices and not to the property of software to run on different platforms. As an example of portable computing platforms we have the PalmPilots, Psions, and the various Windows CE based handheld devices.

Appendix A

Implementation of the send atom

```
package interpreter.expression;

import java.util.*;
import interpreter.*;
import interpreter.value.*;

/**
 * This class represents the send part of a communication atom. It can not
 * evaluated on its own, only together with the corresponding receive atom.
 */
public class TBAtomSend extends TBExpression {

  private TBValueBool enableCondition;
  private TBEnvironment env;
  private byte[] termEncoding;
  private Vector values;
  private Vector receiveList;
  private TBProcessInstance myProcInst;

  /**
   * Create a new send atom.
   */
  public TBAtomSend(byte[] encoding, Vector valueEncoding) {
    enableCondition = new TBValueBoolConst(true);
    termEncoding = encoding;
    values = valueEncoding;
    receiveList = new Vector();
  }

  /**
   * Set the process instance for this expression.
   */
  public void setProcessInstance(TBProcessInstance procInst) {
    myProcInst = procInst;
  }

  /**
   * Evaluate an expression.
   * @returns 0 if an expression could not be evaluated.
```

```
 * @returns 1 if an expression is evaluated.
 */
public int eval() {
  if (enableCondition.eval(env) == true) {
    TBAtomReceive recAtom;
    for (int i = 0; i < receiveList.size(); i++) {
      recAtom = (TBAtomReceive)receiveList.elementAt(i);
      if (recAtom.eval(termEncoding, values, env, myProcInst) == 1) {
        return 1;
      }
    }
  }
  return 0;
}

/**
 * Return the static encoding of this communication term.
 */
public byte[] getTermEncoding() {
  return termEncoding;
}

/**
 * Register a matching receive atom with this send atom.
 */
public void registerReceiver(TBAtomReceive rec) {
  // These should actually be WeakReferences.
  for (int i = 0; i < receiveList.size(); i++) {
    if ( (TBAtomReceive)receiveList.elementAt(i) == rec ) {
    // Receive atom is already in the list.
      return;
    }
  }
  receiveList.addElement(rec);
}

/**
 * Set the list with references to matching receive atoms.
 */
public void setReceiveList(Vector list) {
  receiveList = list;
}

/**
 * Set the environment for an expression and its subexpressions.
 */
public void setEnvironment(TBEnvironment environment) {
  env = environment;
}

/**
 * Rename any occuring variables in the expression.
 */
public void renameVar(String oldVarName, String oldProcName,
```

```
                                String newVarName, String newProcName) {
    enableCondition.renameVar(oldVarName, oldProcName, newVarName, newProcName);
    TBValue argument;
    for (int i = 0; i < values.size(); i++) {
      argument = (TBValue)values.elementAt(i);
      argument.renameVar(oldVarName, oldProcName, newVarName, newProcName);
    }
}

/**
 * Make a new instance of this tree.
 */
public TBExpression copy(TBProcess proc) {
    TBValue val;
    TBValue valCopy;
    byte[] termEncodingCopy;
    Vector valuesCopy = null;
    termEncodingCopy = new byte[termEncoding.length];
    for (int i = 0; i < termEncoding.length; i++) {
      termEncodingCopy[i] = termEncoding[i];
    }
    if (values ≠ null) {
      valuesCopy = new Vector();
      for (int i = 0; i < values.size(); i++) {
        valCopy = null;
        val = (TBValue)values.elementAt(i);
        valCopy = val.copy();
        valuesCopy.addElement(valCopy);
      }
    }
    TBValueBool condCopy = null;
    condCopy = (TBValueBool)enableCondition.copy();
    TBAtomSend theCopy = new TBAtomSend(termEncodingCopy, valuesCopy);
    theCopy.addCondition(condCopy);
    theCopy.setReceiveList(receiveList);
    if (proc ≠ null) {
      TBAtomReceive recAtom;
      for (int i = 0; i < receiveList.size(); i++) {
        recAtom = (TBAtomReceive)receiveList.elementAt(i);
        recAtom.registerSender(theCopy);
      }
    }
    return theCopy;
}

/**
 * Set the conditions under which this atom may be evaluated.
 */
void addCondition(TBValueBool cond) {
    enableCondition = new TBValueBoolAnd(enableCondition, cond);
}

/**
 * Return the type of sub tree.
```

```
    */
    public int getTreeType() {
      return TBExpression.ATOM_SEND;
    }

    /**
     * Bring the tree into normal form and return it.
     */
    public TBNormalExpression normalize() {
      return new TBNormalExpression(this);
    }

    public String toString() {
      StringBuffer strbuf = new StringBuffer("send");
      return strbuf.toString();
    }

}
```

Appendix B

Action relations and axioms for process algebra

The action relations show which actions can be performed and how a process expressions transforms when an atomic action is performed. We see the strict order imposed by the sequential composition operator in Rel2 and Rel3 (we have used the notational convention to leave out "."). The non-determinism of "+" and "||" follows from the fact that the left and the right operand can be exchanged in Rel4, Rel5, Rel6 and Rel7.

(Rel1)	$a \xrightarrow{a} \surd$		
(Rel2)	$x \xrightarrow{a} x'$	$\Rightarrow$	$xy \xrightarrow{a} x'y$
(Rel3)	$x \xrightarrow{a} \surd$	$\Rightarrow$	$xy \xrightarrow{a} y$
(Rel4)	$x \xrightarrow{a} x'$	$\Rightarrow$	$x + y \xrightarrow{a} x'$ and $y + x \xrightarrow{a} x'$
(Rel5)	$x \xrightarrow{a} \surd$	$\Rightarrow$	$x + y \xrightarrow{a} \surd$ and $y + x \xrightarrow{a} \surd$
(Rel6)	$x \xrightarrow{a} x'$	$\Rightarrow$	$x \parallel y \xrightarrow{a} x' \parallel y$ and $y \parallel x \xrightarrow{a} y \parallel x'$
(Rel7)	$x \xrightarrow{a} \surd$	$\Rightarrow$	$x \parallel y \xrightarrow{a} y$ and $y \parallel x \xrightarrow{a} y$
(Rel8)	$x \xrightarrow{a} x'$	$\Rightarrow$	$x \lfloor\!\lfloor y \xrightarrow{a} x' \parallel y$
(Rel9)	$x \xrightarrow{a} \surd$	$\Rightarrow$	$x \lfloor\!\lfloor y \xrightarrow{a} y$

Table B.1: Action relations for process algebra.

For a complete interpretation we need the axioms of process algebra, which are given in table B.2. The axioms show the relations between operators and make it possible to rewrite every process expression so that application of the action relations becomes possible. For example we can rewrite the expression $a * b$ to $a.a * b + b$ (the operator "$*$" binds stronger than "." which in turn binds stronger than "+"). From action relations Rel1, Rel3, and Rel5 it follows that the possible actions for this expression are:

- $a \xrightarrow{a} \surd \Rightarrow a.a * b + b \xrightarrow{a} a * b$
- $b \xrightarrow{b} \surd \Rightarrow a.a * b + b \xrightarrow{b} \surd$

In many process algebras there exists a conditional operator like `if-then-else-fi`, sometimes denoted by $s \lhd C \rhd t$, in the original implementation as well as in our prototype the if-then-else construction is implemented using conditionally evaluated atoms. Much more can be said about the process algebra used in the ToolBus, for more information we refer to [BBP94] and [BB95].

(A1)	$x + y$	$=$	$y + x$		(CM1)	$x \parallel y$	$=$	$x \between y + y \between x + x \mid y$
(A2)	$x + (y + z)$	$=$	$(x + y) + z$		(CM2)	$a \between x$	$=$	ax
(A3)	$x + x$	$=$	x		(CM3)	$ax \between y$	$=$	$a(x \parallel y)$
(A4)	$(x + y)z$	$=$	$xz + yz$		(CM4)	$(x + y) \between z$	$=$	$x \between z + y \between z$
(A5)	$(xy)z$	$=$	$x(yz)$					
(A6)	$x + \delta$	$=$	x		(I)	$x * y$	$=$	$x.x * y + y$
(A7)	δx	$=$	δ					
(CF1)	$a \mid b$	$=$	$\gamma(a, b)$	if $\gamma(a, b) \downarrow$				
(CF2)	$a \mid b$	$=$	δ	otherwise				

Table B.2: The process algebra axioms.

References

[BB95] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. Technical report, Programming Research Group, University of Amsterdam, 1995.

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

[BK87] J.A. Bergstra and J.W. Klop. *Semi-complete termherschrijfsystemen.* Kluwer Bedrijfswetenschappen, 1987.

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

[BKM89] J.A. Bergstra, J.W. Klop, and A. Middeldorp. *Termherschrijfsystemen.* Kluwer Bedrijfswetenschappen, 1989.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[Gro99] Object Management Group. The common object request broker: Architecuter and specification. Technical report, Object Management Group, October 1999. Available at: `http://www.omg.org`.

[JGS96] Bill Joy James Gosling and Guy Steele. *The Java Language Specification.* Addison-Wesley, 2nd edition, 1996.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, third edition, 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, third edition, 1997.

[LMB92] John Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly, second edition, 1992.

[LV97] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer, 1997.

[Mic97] Sun Microsystems. Javabeans. Technical report, Sun Microsystems, July 1997. Available at: `http://java.sun.com`.

[MPI95] Mpi: A message-passing interface standard. Technical report, Message Passing Interface Forum, june 1995. Available at: `http://www.mpi-forum.org/docs/docs.html`.

[PVV94] A. Ponse, C. Verhoef, and S.F.M. Vlijmen, editors. *Algebra of Communicating Processes*. Springer-Verlag, 1994.

[Rom99] J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. PhD thesis, Universiteit Twente, October 1999.

[Szy97] Clemens Szyperski. *Component Software*. Addison-Wesley, 1997.

[vD94] A. van Deursen. *Introducing ASF+SDF*. PhD thesis, University of Amsterdam, 1994. Available at: `http://www.cwi.nl/~gipe/asf+sdf.html`.

[vdBdJKO99] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. Technical report, Programming Research Group, University of Amsterdam, August 1999.

[WK94] Sara Williams and Charlie Kindel. The component object model: A technical overview. Technical report, Microsoft Corporation, October 1994. Available at: `http://www.microsoft.com`.