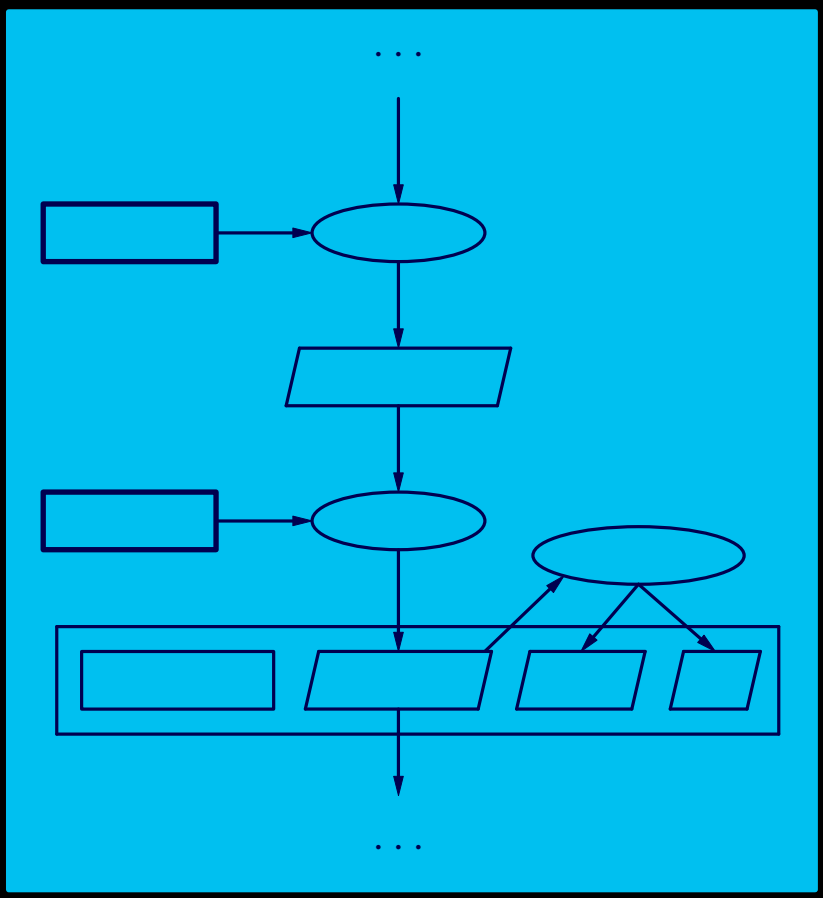
Software Engineering with Process Algebra



Bob Diertens

Software Engineering with Process Algebra

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op donderdag 29 oktober 2009, te 12.00 uur

door

Bob Diertens

geboren te Rotterdam

promotor:	prof. dr. J.A. Bergstra
co-promotor:	dr. A. Ponse
faculteit:	Natuurwetenschappen, Wiskunde en Informatica

with help from friends

Contents

Figures

Tables

Part I

Introduction

Chapter 1

Aims and Scope

Making a small piece of software is relatively easy for a person having some programming skills. For constructing a larger piece of software also the skill is needed to divide it into small pieces that cooperate together to perform the task of the larger piece. Dividing the larger pieces into smaller pieces is called design. In this design phase, it is not necessary to think about how the small pieces can be implemented. So design abstracts from the implementation of the small pieces. The larger pieces of software need several design decisions to divide them into smaller pieces. In order to not make all these design decisions at once, several levels of design are necessary. Designing large software systems can thus be done step by step, where each step consists of some design decisions that have to be made at a certain level of design and that can not be pushed to a lower design level. With every step there can be feedback to the higher levels, which can lead to improvements or changes in the higher level designs.

Usually, testing of software systems starts with the testing of the small pieces followed by the testing of the larger pieces and finally by the testing of the whole system. If the design contains errors or has some shortcomings, this only becomes clear at the later stages of the testing. It would be better if the design can be tested to detect errors at an early stage. If the design is put down in writing it can only be tested by inspection. If the design is specified in a formalism it can be tested with the use of tools, and with the right tools some properties can even be verified.

1.1 Research Questions

This thesis describes the project of using process algebra as a formalism in software engineering and software re-engineering. We aim at making process algebra specifications of software design at various abstract levels. Tools that work on process algebra specifications should make it possible to test these specifications.

As specification language we use the process algebra based language Process Specification Formalism (PSF) developed at the University of Amsterdam, for which a toolkit is freely

available. Furthermore, we make use of the ToolBus as target for the coordination of software components. The ToolBus is a coordination architecture developed at the University of Amsterdam and CWI. It utilizes a scripting language based on process algebra to describe the communication between software tools.

Our work is motivated by applications of process algebra in software and software engineering. In software the ToolBus is used for coordination of components using scripts based on process algebra. These scripts for the ToolBus can get quite large and complex for larger software systems and the ToolBus provides limited possibilities for debugging the scripts. If we specify the ToolBus scripts in PSF with abstractions for the tools coordinated by the ToolBus, we can validate the specifications of the scripts with the use of the PSF Toolkit. It is interesting that the idea for the ToolBus originates from [65] that gives a PSF specification describing the interaction of the components for a distributed editor consisting of a user-interface, text editor, and structure editor. This specification uncovered several communication problems and potential deadlocks in the implementation. The size and complexity of this specification led to the idea for a ToolBus providing a built-in communication protocol. In software engineering process algebra is applied on the level of the architecture. There exist several Architecture Description Languages (ADLs), some of which are based on a form of process algebra. Using PSF for the specification of software architecture makes it possible to use the tools available for PSF to validate the specifications of the architecture.

Our goal is to find out how useful PSF is in the field of software engineering and if the PSF Toolkit is adequate to support validation of the specifications. We can make a specification of the behaviour of a software system on a certain abstract level. By applying algebraic laws for abstraction we must be able to obtain a specification of the system on a higher abstract level and with enough abstraction we should get a specification of the architecture of the system. We want to support the specification of software on various abstract levels of design, making it possible to use the same formalism and tools available for this formalism on different levels of design. Furthermore, we want to support the process of developing a lower level specification of design (implementation) from a higher level specification.

It is not our intention to change the software engineering process, nor do we advocate such a change. We only want to find out if process algebra can be of any help in the software engineering process, or even improve some parts of this process. We do not describe software engineering processes here, instead we refer to [56] for an overview. It is also not our goal to verify our specifications or to verify some properties of the specifications. In this context verification is the process of proving a product correct and validation is the process of testing that a product functions properly.

1.2 Why using Process Algebra?

To answer the question of the usefulness of PSF in the field of software engineering we first have to make clear why we use process algebra as a formalism in the design of software systems. Process algebra in the style of the Algebra of Communicating Processes (ACP) was designed to describe and analyse the behaviour of processes in concurrency, which communicate with each other. It is mostly used in the setting of communication protocols.

The exchange of information between objects, whether in architecture, implementation, or at some intermediate level, is usually prescribed by some protocol. The exchange on the implementation level can be done in many forms, such as a function call with possible return values, remote procedure call, invocation (message to a mechanism that can invoke a particular object), etc. On the higher levels we abstract from the form of communication, but on all levels we have components that interact. Process algebra is suitable for the specification of these components and the interaction between them, and can thus be used for the specification of the design at the various levels.

1.3 Validation of Process Algebra Specifications

Using process algebra in software design is only useful if we can validate the specifications. Validation can be done by inspection but for larger specifications this is not sufficient. At least some tools are needed for syntactic and semantic analysis. In order to validate the behaviour of a specification a simulator is needed. The PSF Toolkit described in Chapter 3 contains a compiler and a simulator. Although the simulator is perfectly capable of simulating the behaviour of (compiled) specifications, for complex specifications it can be difficult to keep track of what is going on. Visualization of the current state and of transitions between states is necessary or at least very useful for a better understanding. Adding animation facilities to the simulator can make the validation of (complex) specifications easier, and thus can be useful in the software engineering process.

1.4 Outline of this Thesis

In the remainder of Part I of this thesis we give an introduction to the various elements we use in our work. We start with an introduction to the process algebra based language PSF, followed by a description of the PSF Toolkit. Next, we give an introduction to the ToolBus that we use as coordination architecture for the software design described in this thesis.

In Part II we present a platform for coupling animation to the simulator from the PSF Toolkit. This platform makes it possible to animate actions executed by the simulator from the PSF Toolkit. The simulation can either be controlled through the simulator or through the animation. We show how to use this platform by describing how to make animations for two small specifications. We also describe how animations can be generated from PSF specifications. The generation of an animation from a specification keeps the animation consistent with the specification and therefore it is useful in the development process of the specification.

In Part III we apply PSF in the software engineering process. As test cases, we (re-)engineer tools from the PSF Toolkit. This is convenient since we are familiar with the software and at the same time the (re-)engineering can improve the Toolkit. First, we re-engineer the compiler from the PSF Toolkit. We present a PSF library to support the specification of ToolBus applications. We transform an existing specification for the compiler into a ToolBus application specification using the PSF ToolBus library. From the ToolBus application specification we make an implementation of the compiler as ToolBus application. A specification of the architecture of the re-engineered compiler is extracted from its specification. The relation between the specifications and application is depicted in

Figure 1-1. The specification of the architecture is used as starting point for developing a parallel compiler.

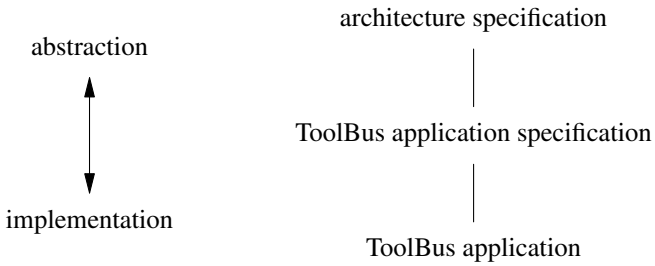


Figure 1-1. Relation between specifications and application

Next, we present a PSF library for specifying software architecture. We introduce implementation techniques for refinement and constraining, which can be used to refine architecture specifications into ToolBus application specifications. We make a new implementation of the simulator from the PSF Toolkit, starting with an architecture specification for the simulator using the PSF Architecture library. From the ToolBus application specification we develop an implementation of the simulator as a ToolBus application. With the technology thus obtained we engineer a new tool, namely an integrated development environment (IDE) for PSF. An IDE is a software application that provides the facilities for the development of specifications. In the final chapter of this part we describe our software development process more formally by presenting it in a Computer-Aided Software Engineering (CASE) setting. We present a software engineering environment based on the development of architecture specifications and ToolBus application specifications. Generalization of a part of this environment leads to a process algebra software engineering workbench.

We evaluate our work in Part IV, starting with conclusions on our work, followed by our thoughts on how to promote software engineering with process algebra to industry. We end with some thoughts on further work.

Chapter 2

PSF

PSF has been designed as the base for a set of tools to support ACP (Algebra of Communicating Processes) [4]. ACP is an axiom based mathematical theory for concurrency. An ACP specification starts from a set of objects called atomic actions, atoms, or steps. Process expressions are built up from atomic actions by means of operators. The most important operators are sequential composition, alternative composition, parallel composition, communication, encapsulation, and abstraction. Infinite processes are specified by one or more recursive equations. The syntax of the processes in PSF is very close to the informal syntax normally used in denoting ACP-expressions. The part of PSF that deals with the description of data is based on ASF (Algebraic Specification Formalism) [5]. PSF is mainly used in the specification of communication protocols, but it also used in the specification of, e.g. traffic control, bank account, and model factories. Publications on this usage can be found through the PSF website [16].

A description of PSF can be found in the PhD thesis of Sjouke Mauw [37]. For an extensive description and its use in the specification of some communication protocols we refer to [38]. In this chapter we give an informal description of PSF. A small example of a specification in PSF is given in the last section of this chapter. A larger example can be found in Appendix A.1. That example specifies the Alternating Bit Protocol, a simple communication protocol often used as a test case.

2.1 Modules

PSF has two types of modules, data and process modules. The structure of data modules is borrowed from ASF, and process modules have a similar structure. A module consist of sections that have a predefined order. Below we give the structure for both types of modules. We use *italics* to indicate identifiers to be filled in, and $\cdots$ for a list of elements. The order of the sections is relevant. Each section starts with a keyword followed by the elements of that section. If a section does not have any elements, the associated keyword can be omitted.

data module *Module*
begin
  sorts
    ...
  functions
    ...
  variables
    ...
  equations
    ...
end *Module*

process module *Module*
begin
  atoms
    ...
  processes
    ...
  sets
    ...
  communications
    ...
  variables
    ...
  definitions
    ...
end *Module*

In the following we give a description for each section.

Sorts

Sorts are to be declared as a comma separated list, as below

```
S,
DATA
```

Functions

Function declarations have a list of types for the arguments separated by a '#', and a result type indicated with '->' (pretty printed as →).

```
f : → S
f : DATA → S
f : S # DATA → S
```

Functions can also be declared as infix- or prefix-operators. The name of an operator consists either of one or more operator symbols or of a normal function name enclosed in '.'. The places for the arguments of the operator are indicated in the declaration with a '_'.

```
_&_ : B # B → B
_@*$_ : B # B → B
_.and._ : B # B → B
!_ : B → B
.not._ : B → B
```

These operators can be used as shown below.

```
x & y
x @*$ y
x .and. y
! x
.not. y
```

Variables

Depending on the type of the module, the variable section lists the variables used either in the equations or in the process definitions. The type of variables is indicated with '->'.

```
x : → S
```

Equations

The definition of a function is given by means of equations. An equation is interpreted as a rule for a term rewrite system, in which the left hand side is rewritten to the right hand side. In the example below we give equations for the boolean `and` function.

```
[and1] and(false, false) = false
[and2] and(false, true) = false
[and3] and(true, false) = false
[and4] and(true, true) = true
```

Tags (`[`*x*`]`) on the left side of the equations are for documentation purposes only.

It is possible to use variables in an equation that are declared in the variables section. The variables get a value from the matching of terms with the left hand side in the rewrite process. These values are then substituted in the right hand side.

```
[and1] and(false, x) = false
[and2] and(true, x) = x
```

Equations can have conditions which have to be fulfilled before such equations can be applied. A condition is indicated with the keyword **when** followed by a comma separated list of equations.

```
[and1] and(x, y) = false when x = false
[and2] and(x, y) = false when y = false
[and3] and(true, true) = true
```

Atoms

Atom declarations have a list of types for the arguments separated by a '#'.

```
a
a : S
a : DATA # S
```

Processes

Process declarations have a list of types for the arguments separated by a '#'.

```
P
P : S
P : DATA # S
```

Sets

A sets section consists of sub-sections, each indicating the type of the sets declared in this sub-section.

of atoms
    H = *set-expression*
of S
    D = *set-expression*
    E = *set-expression*

A set expression can be one of the following constructions in which S and T denote set expressions.

sort S
indicating all elements of the sort S

set S
indicating all elements of the set S

enumeration $\{e_1, e_2, \cdots, e_n\}$
An enumeration can contain placeholders.

```
H = {a(x), b(y) | x in S, y in DATA}
```

union $S + T$

intersection $S \cdot T$

difference $S \setminus T$

Communications

A communication consist of two communication partners separated by a 'l' and the resulting action of that communication. Communication declarations can contain placeholders, in which case a communication can only take place if the communication partners have the same value for each placeholder. The values for the placeholders are substituted in the resulting communication action.

```
a | b = c
a(x) | b(x) = c(x) for x in S
a(x) | b(y) = c(x, y) for x in S, y in S
```

Definitions

Process definitions consist of a process on the left hand side and its definitions on the right hand side. The process can have terms as arguments in which variables, defined in the variables section, may occur.

```
P = process-expression
P(x) = process-expression
P(f(b), b(y)) = process-expression
```

Upon execution of a process, the process will be matched with the left hand side of the process definitions. The process will be replaced by an alternative composition of all the definitions for which the left hand side matches. If there is no matching process definition, a *deadlock* results. In the matching of the process, values for the variables will be determined, which will be substituted in the right hand side of the definition.

In the following we list possible constructions for process expressions and describe their behaviour.

atomic action a
Execution of the atomic action a.

internal step skip
Execution of an internal step not visible to the environment. In ACP based extensions it is known as τ.

deadlock delta
A deadlock cannot be executed.

process P
The process P will be replaced by an alternative composition of all the process definitions of which the left hand side matches with P.

sequential composition $x \cdot y$
Process expression x is executed and upon termination followed by the execution of process expression y.

alternative composition $x + y$
A non-deterministic choice between process expressions x and y is made. Choosing a deadlock is forbidden.

parallel composition $x \parallel y$
The process expressions x and y are executed in parallel in which possible communications can take place.

generalized alternative composition $sum(v\ in\ S, x)$
An abbreviation of the alternative composition of, for every value of v in the sort or set S, the process expression x in which v is replaced by the value.

generalized parallel composition $merge(v\ in\ S, x)$
An abbreviation of the parallel composition of, for every value of v in the sort or set S, the process expression x in which v is replaced by the value.

encapsulation $encaps(H, x)$
Can only execute actions from process expression x that are not an element of set H.

hiding $hide(I, x)$
Let the executed actions from process expression x behave as the internal step *skip*.

conditional expression $[t = u] \rightarrow x$
If the terms t and u are equal, the conditional expression evaluates to the process expression x, and to deadlock otherwise.

In addition to the above described process constructions, PSF also has the following constructions, that are described in [17] and [18].

interruption $interrupt(x, y)$
The executions of process expression x can be interrupted by process expression y at any time. After the execution of process expression y has finished, the execution of process expression x resumes.

disruption $disrupt(x, y)$
The execution of process expression x can be interrupted by process expression y at any time. After the execution of process expression y has finished, instead of continueing the execution of process expression x, the execution of the whole process expression is finished.

priority $prio(S_1 > \cdots > S_n, x)$
: Gives priority of actions in process expression x that appear in set S_i over actions that appear in set S_j for $j > i$. This means that when there is a choice between actions only the actions with the highest priority and the actions without a priority can be chosen. With the priority operator, the use of the keyword *atoms* is extended to not only denote the set type *atoms*, but also to denote the set of all atomic actions. The process expression $prio(S, x)$ is an abbreviation of $prio(S > atoms, x)$.

iteration $x * y$
: Chooses between process expressions x and y. If x is chosen, upon termination of the execution of x the iteration is repeated. If y is chosen, upon termination of the execution of y the iteration finishes.

nesting $x \# y$
: Chooses between process expressions x and y. If x is chosen, upon termination of the execution of x the nesting is repeated. If y is chosen, upon termination of the execution of y, x is executed the number of times x has been chosen.

Parentheses may be used to group process expressions.

2.2 Overloading

The names of functions, atoms, and processes, can be overloaded. This means that the same name can be used to denote different functions, atoms, or processes. Overloaded functions must have unique input types, consisting of the types for the arguments. The same also applies for atoms and processes, which at the same time must be distinguishable from each other.

2.3 Imports and Exports

All objects defined in a module are only visible within that module. To make objects specified in one module visible in other modules, PSF uses an import/export mechanism. Every module can have an export and an import section. The layout for these sections of data and process modules is given below.

```
data module Module
begin
    exports
    begin
        sorts
            ...
        functions
            ...
    end
    imports
        ...
    ...
```

```
process module Module
begin
    exports
    begin
        atoms
            ...
        processes
            ...
        sets
            ...
    end
    imports
        ...
    ...
```

All objects defined in an export section of a module are made visible to modules that import

this module. Objects imported by a module are also exported by this module, and thus are visible in other modules that import this module.

The imports section consists of a comma separated list of module names. A data module can only import data modules, and a process module can import both data and process modules.

2.3.1 Renamings

Upon import of a module, visible objects in this module can be renamed. A renaming construct specifies a list of pairs consisting of an old visible name and a new visible name. It has the following layout.

```
imports
    Module {
        renamed by [
            a → b,
            ...
        ]
    }
```

If a renaming is applied to an overloaded name, all instances of this name will be renamed.

2.4 Parameters

A parameters section must appear as the first section in a module and has the following layout.

```
parameters
    Parameter
    begin
        sorts
            ...
        functions
            ...
        atoms
            ...
        processes
            ...
        sets
            ...
    end
    ...
```

Upon import of a module with parameters, a parameter can be bound to a module while all objects listed in the parameter are bound to actual objects from this module. Unbound parameters are inherited by the importing module and become parameters of this module. A parameter binding has the following layout.

```
imports
    Module1 {
        Parameter bound by [
            a → b,
            ...
        ] to Module2
    }
```

2.5 Example

As an example of a specification in PSF we specify a system consisting of the processes P and Q that communicate with each other. Process P can either send a 'message' to process Q and then wait for an acknowledgement from Q, or it can send a 'quit' after which the system stops.

We start with specifying a data module containing the definitions for the different data to be send between the processes P and Q.

```
data module Data
begin
    exports
    begin
        sorts
            DATA
        functions
            message : → DATA
            ack : → DATA
            quit : → DATA
    end
end Data
```

We continue with a process module that specifies the system consisting of the processes P and Q. This module imports the data module.

```
process module System
begin
    imports
        Data
    atoms
        snd : DATA
        rec : DATA
        comm : DATA
    processes
        P, Q
        System
    sets
        of atoms
            H = { snd(x), rec(x) | x in DATA }
    communications
        snd(x) | rec(x) = comm(x) for x in DATA
    definitions
        P = snd(message) . rec(ack) . P + snd(quit)
        Q = rec(message) . snd(ack) . Q + rec(quit)
        System = encaps(H, P || Q)
end System
```

The process System merges the processes P and Q and enforces the communication between them by encapsulating this merge. The encapsulation prevents the execution of the actions in set H, which now can only be executed as part of the communication action.

Chapter 3

The PSF Toolkit

PSF is accompanied by a toolkit consisting of several tools and libraries that form a specification environment for PSF. The tools operate around the Tool Interface Language (TIL) [39] as shown in Figure 3-1. There are several advantages in the use of an intermediate language. The most important one is that the tools do not have to parse and type check the PSF code, but use a simple parser to read the intermediate language. Another advantage is that languages similar to PSF can be compiled to the intermediate language and make use of large parts of the tools.

An overview of the PSF Toolkit is given in [62], and a more detailed description of the tools is given in the PhD thesis of Gert Veltink [63]. In this chapter we describe briefly some of the tools in the PSF Toolkit.

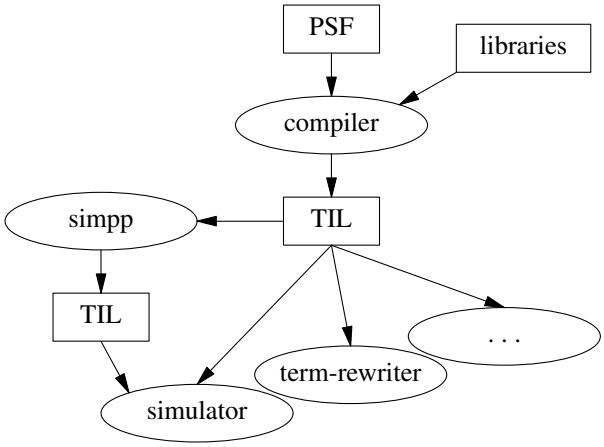


Figure 3-1. PSF-Toolkit

3.1 Compiler

The PSF compiler translates a group of PSF modules to a specification in the Tool Interface Language (TIL) that is suitable for tools to operate upon. The translation process from PSF to TIL is described in [64]. Here, we give a brief description of this process.

The compilation takes place in several phases. First each PSF module is parsed and converted to an MTIL (modular TIL) module. Then each MTIL module is normalized into an ITIL (intermediate TIL) module. In this normalization step all imports are resolved by combining the MTIL module with the ITIL modules corresponding to the imported modules. The resulting ITIL module no longer depends on any imports. The main ITIL module is then flattened to a specification in TIL. An overview of these steps is shown in Figure 3-2.

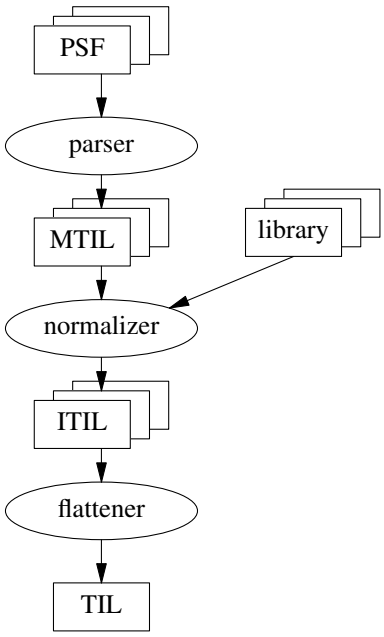


Figure 3-2. Translation from PSF to TIL

The implementation of the PSF compiler is built up from several independent components, controlled by a driver. The compiler driver consists of the following phases.

1. **collecting modules**
   The modules are collected from the files given to the compiler, and missing imported modules are searched for in the libraries.

2. **sorting modules**
   The modules are sorted according to their import relation.
3. **splitting files**
   Files scanned in phase 1 that contain more than one module are split into files containing one module each.
4. **parsing** (from PSF to MTIL)
   All modules that are out of date, that is the destination file does not exist, or the source file (with extension .psf) is newer than the destination file (with extension .mtil), are parsed.
5. **normalizing** (from MTIL to ITIL)
   All modules that are out of date, that is the destination file does not exist, or the source file (with extension .mtil) is newer than the destination file (with extension .itil) or one of its imported modules (ITIL) is newer, are normalized.
6. **flattening** (from ITIL to TIL)
   The main module is translated from ITIL to TIL.
7. **converting sorts to sets**
   The simulator preprocessor is invoked for converting sorts to sets so that the simulator can handle them.
8. **checking TRS**
   The term rewrite system checker is invoked.

3.2 PSF Libraries

The PSF Toolkit provides some libraries containing frequently used data types. Using libraries has several advantages. The main advantage is that specifications can be built using existing modules. Also, specifications get a certain uniformity if they are based on the same set of basic modules.

A PSF library consists of a set of ITIL-modules possibly depending on each other and on other libraries. The compiler resolves imports from libraries by means of a search path consisting of one or more directories.

In [38] a small standard library is used to support the communication protocols specified in it. A library with more data types is proposed in [68]. Both libraries are in the PSF Toolkit, but only to support older specifications, since there is a third library described in [40] that is a revised version of the second. This revised version has a much faster term rewriting system for some of the data types, and contains some additions.

3.3 Simulator

The simulator lets a user interactively simulate a process from a specification in TIL-code. We describe briefly how the simulator operates and some of the features.

3.3.1 Basic Simulation

Simulation starts with selecting a start process for which a process-tree is build. Each process in the tree keeps track of the code it has to execute. This code is split up in the current operator or atom to be executed (head) and the remaining code (tail).

From the process-tree a list of atoms that can be executed at the current state called the action-list is calculated, including possible communications between the atoms. Choosing an atom from the action-list results in the execution of this atom after which the atom is removed from the code to be executed in the process-tree, and the process-tree is updated.

3.3.2 Features

Basic simulation is sufficient for testing of small specifications, but for larger specifications more control over the simulation is needed. In the following we describe some features implemented in the simulator that make testing of large specifications easier.

Random

In random mode, the simulator runs continuously selecting atoms at random from the list of possible atoms to be executed.

Breakpoints

Breakpoints can be set on atoms in order to stop random simulation. The simulator can handle breakpoints in three different ways.

on execution (the default)
: Whenever an atom is executed on which a breakpoint is set, the random mode is turned off.

stop when one
: If one or more of the atoms in the action-list have a breakpoint set on them, the random mode is turned off.

stop when all
: When all of the atoms in the action-list have breakpoints set on them, the random mode is turned off. When random mode is on, atoms with breakpoints on them are not chosen. This makes it possible to synchronize the simulation of processes.

Tracing

Atoms can be selected to be traced. Upon execution of an atom that is selected for tracing, the atom will be displayed.

Process Status

The process-tree can be viewed in the following form. For each process the process id

(PID), the process id of the parent (PPID), the status of the process (STATUS), the flags of the process (FLAGS), the priority (PRIO), and the head that is currently simulated (HEAD), are given.
If a process has children, STATUS indicates this with an S (sleeping) and the number of children. FLAGS can contain the following flags:

D indicates that the process resulted in a deadlock.

I the process is idle, due to an *interrupt*, *disrupt*, *, or # operator.

P the process can act as a port (a *sum* operator used to receive a value from another process).

E indicates that the atom in the head is encapsulated.

H indicates that the atom in the head is hidden.

C a communication is possible with another atom.

The process status also gives a list of possible communications. For each of the communications the process id of the two processes involved (PID), the flags of the communication (FLAGS), the priority (PRIO), and the resulting communication (COMM) are given. FLAGS can contain an E (encapsulated) or H (hidden).

For example, if we simulate the process definition $P = encaps(H, (a \parallel b) \parallel (c \parallel d))$ with $H = \{a, b\}$ and communication $a \mid b = cab$ we get the following process status.

```
PID   PPID STATUS FLAGS PRIO HEAD
  0          S  1              <+>
  1      0   S  2              <||>
  2      1   S  2              <||>
  3      1   S  2              <||>
  4      2          E C     0  a
  5      2          E C     0  b
  6      3                  0  c
  7      3                  0  d
PID    PID FLAGS PRIO COMM
  4      5          0  cab
```

The alternative in the head of process 0 is caused by the possibility in PSF (and TIL) to define more than one process definition with the same left hand side, in which case they have to be interpreted as alternatives.

History

The simulator is equipped with a history mechanism that makes it possible to undo and redo actions. In addition to single steps through history, it is also possible to mark a state by giving it a name and later go back to a marked state by selecting a state from a list of names.

3.3.3 Simulator Pre-Processor

The simulator can only expand *sum* and *merge* expressions over sets consisting solely of an enumeration of constant elements. In the other cases, the simulator is not capable of

calculating the elements of the sort or set, of which the number of elements can even be infinite. For this purpose, a pre-processor has been built that tries to convert sorts and sets to sets consisting of only an enumeration.

A sort can be converted to an enumerated set when all function definitions with this sort as return-type have no arguments (i.e. are constants). If a function takes arguments then the sort can only be converted if the sorts of the arguments can be converted. In that case, the set contains elements build up from this function with all the possible combinations of values for the arguments.

A set can be converted to an enumerated set when it depends on enumerated sets and on sorts which can be converted to enumerated sets. Furthermore, a set can be converted when it depends on sets and sorts that can be converted to enumerated sets. For the intersection set operator (.) it is only necessary that one of the operands can be converted to an enumerated set, since values can be tested on being an element of a set. For the difference set operator (\) only the left operand needs to be convertible.

For the sorts that cannot be converted a finite projection of the initial algebra can be calculated, by enumerating normal forms. Calculation of an initial algebra is done in segments. The first segment consists of constant functions. Following segments are computed by filling the arguments of the non-constant functions with values from earlier segments. Results are normalized and added to the new segment. The calculation goes on until no elements are formed in the new segment, or until a given maximum number of elements of a sort is exceeded.

3.3.4 Sum-ports

Often, the *sum* is used to receive a value from another process. In some of these cases, a term for the variable of the *sum* can be computed. Consider the process definitions for P and Q, and the set definition for H

```
P = send(b)
Q = sum(x in X, receive(x) . action(x))
H = { send(x), receive(x) | x in X }
```

From examining the expression $encaps(H, P \parallel Q)$, it can be seen that we only need to expand the *sum* for the value b of variable x.

The simulator is capable of computing the right values if the following conditions are fulfilled.

- The first atom of the *sum* has to be easy to find, i.e. the expression of the *sum* consists only of either an atom or a sequence of which the left-most operand is an atom. Or the expression of the *sum* is a *sum* or a sequence of which the left-most operand is a *sum*.
  In case it is a *sum*, this condition must be fulfilled for this *sum* also.
- The atom must have the variables of the *sums* as its arguments, and the variables may not appear in other arguments of the atom.

- The atom has to be encapsulated. For that, the encapsulation and hide sets (that are involved) have to consist only of an enumeration of atoms with only variables (placeholders) as arguments.
- The communication partner must have the same variables (placeholders) of the atom's arguments, which are variables of the *sums*, somewhere in one of the arguments in the specified communication.

3.4 Term Rewriter

For testing the equations in a specification, the PSF Toolkit has been provided with a term-rewriter called 'trs'. This term-rewriter rewrites the terms given by the user according to the equations found in the TIL-code using the rightmost-innermost strategy. The kernel of the term-rewriter is also part of the kernel of the simulator. Upon request, a trace of the rewrite steps is given. Below we show an interactive session with the term-rewriter.

```
% trs Booleans.til
--> or(true,true)
or(true, true)  = true
--> eq(or(true,false),not(true))
eq(or(true, false), not(true))  = false
--> >trace on
Trace on
--> eq(or(true,false),not(true))
eq(or(true, false), not(true))  =
  not(true)
  -> false
  or(true, false)
  -> true
  eq(true, false)
  -> false
false
-->
%
```

The first equation found with matching left hand side is used for rewriting a term. This makes term rewriting dependent on the order of the equations given to the term-rewriter. A *confluent* specification is not order dependent, however it is a common error to incorporate order dependencies in specifications. For that reason the term-rewriter has an option for handling the equations in reversed order.

Chapter 4

ToolBus

The ToolBus coordination architecture [6] is a software application architecture developed at CWI (Amsterdam) and the University of Amsterdam. It utilizes a scripting language based on process algebra to describe the communication between software tools. A ToolBus script describes a number of processes that can communicate with each other and with various tools existing outside the ToolBus. The role of the ToolBus when executing the script is to coordinate the various tools in order to perform some complex task. A language-dependent adapter that translates between the internal ToolBus data format and the data format used by the individual tools makes it possible to write every tool in the language best suited for the task(s) it has to perform.

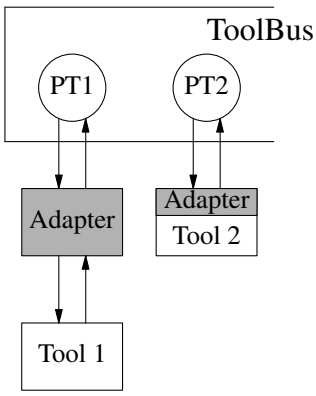


Figure 4-1. Model of tool and ToolBus interconnection

We give a minimal description of the ToolBus, just enough for our purposes. We refer to the user guide distributed with the ToolBus software package for a complete description. In

Figure 4-1 two possible ways of connecting tools to the ToolBus are displayed. One way is to use a separate adapter and the other to have a built-in adapter. Communications between ToolBus processes can be done using the primitives `snd-msg` and `rec-msg`. A ToolBus process can communicate with a tool using the primitives `snd-do` and `snd-eval`. With the latter the tool has to send back a value which the ToolBus process can receive with the primitive `rec-value`. A tool can send an event to a ToolBus process that is to be received with the primitive `rec-event`, to be acknowledged by the ToolBus process using the primitive `snd-ack-event`. An overview of possible communications inside the ToolBus and with the tools is given in Table 4-1, here `<function>` represents the function to be called by the adapter of the tool.

Table 4-1. ToolBus communications

ToolBus process	**ToolBus process**
`snd-msg(`*Term*`, ...)`	`rec-msg(`*Term*`, ...)`
ToolBus process	**Tool**

`snd-do(`*ToolID*`, <function>(`*arg*`, ...))`
`snd-eval(`*ToolID*`, <function>(`*arg*`, ...))`
`snd-ack-event(`*ToolID*`, `*Term*`)`
	`<function>`
	`<function>`
	`<rec-ack-event>`

Tool	**ToolBus process**
`snd-value(`*Term*`)` `snd-event(`*Term*`, ...)`	`rec-value(`*ToolID*`, `*Term*`)` `rec-event(`*ToolID*`, `*Term*`, ...)`

4.1 Example

As an example of the use of the ToolBus, we specify an application carried out in the form as shown in Figure 4-1. In this example, Tool1 can either send a 'message' to Tool2 and then wait for an acknowledgement from Tool2, or it can send a 'quit' after which the application will shutdown.

The implementation consists of three Tcl/Tk[1] [50] programs (Tool1, its adapter, and Tool2), and a ToolBus script. A screendump of this application at work together with the viewer of the ToolBus is shown in Figure 4-2. With the viewer it is possible to step through the execution of the ToolBus script and view the variables of the individual processes inside the ToolBus. The ToolBus script is shown below. The `execute` actions in the ToolBus script correspond to the starting of the adapter for Tool1 and the starting of Tool2 in parallel with the processes `PT1` and `PT2` respectively. The variables `T1` and `T2` are used as identifiers for Tool1 and Tool2, and the terms `t1` and `t2` serve as identifiers for the ToolBus processes `PT1` and `PT2`.

```
process PT1 is
let
   T1: tool1adapter
in
   execute(tool1adapter, T1?) .
   (
```

1. Tcl/Tk is combination of the tool command language Tcl and the Tk GUI toolkit extension package.

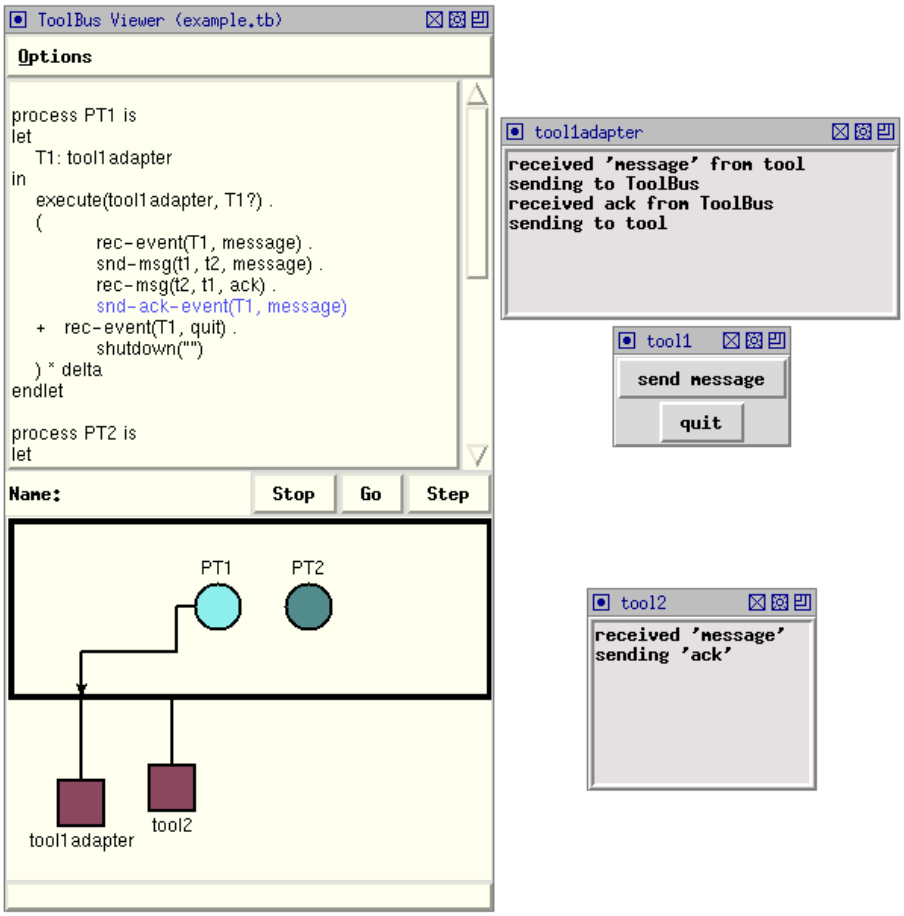


Figure 4-2. Screendump of the example as ToolBus application with viewer

```
      rec-event(T1, message) .
      snd-msg(t1, t2, message) .
      rec-msg(t2, t1, ack) .
      snd-ack-event(T1, message)
   +  rec-event(T1, quit) .
      shutdown("")
   ) * delta
endlet

process PT2 is
let
   T2: tool2
in
   execute(tool2, T2?) .
   (
      rec-msg(t1, t2, message) .
      snd-eval(T2, eval(message)) .
      rec-value(T2, value(ack)) .
```

```
        snd-msg(t2, t1, ack)
    ) * delta
endlet

tool tool1adapter is {
    command = "wish-adapter -script tool1adapter.tcl" }
tool tool2 is { command = "wish-adapter -script tool2.tcl" }

toolbus(PT1, PT2)
```

Following the description of the ToolBus processes is the description of how to execute the tools by the execute actions. The last line of the ToolBus script starts the processes `PT1` and `PT2` in parallel.

Part II

Animation of Process Algebra Specifications

Chapter 5

Simulation and Animation

When simulating a process algebra specification, one easily loses track of the current state of the parallel processes in the specification. A visualization of the state can be very helpful, especially for larger specifications. We can even go further. By also visualizing the transitions between the states we get an animation of the simulations of our specification.

What do we expect from such an animation? First of all, what our simulator already does, show which actions are performed. We also like to see which processes can perform an action and how their states are influenced by execution of an action. But above all, we would like to see a picture in which objects are shown that represent the processes and their connecting communication channels, and in which the formerly mentioned actions are visualized.

In the following section we present a platform for the coupling of animation to the simulator of the PSF Toolkit and describe the implementation of this platform. The structure of an animation is presented in section 5.2. In the sections thereafter we show how to build animations on top of the platform.

5.1 Simulation-Animation Platform

The simulation-animation platform consists of the simulator from the PSF Toolkit coupled to an animator. The animator executes an animation provided by the user. Such an animation can be build using commands from a library of animation functions. An animation consists of a number of commands that build up a picture and for each atom that can be executed by the simulator a set of commands that perform the animation for that atom. There are two modes of control, passive animation and active animation.

With passive animation, the simulator is in control. It sends the atoms it executes to the animator. Upon the receipt of an atom from the simulator, the animator interprets the atom and executes the set of commands associated with this atom. With active animation, the

animator is in control. Selection of atoms to be executed by the simulator is done through the interface of the animation. For this, the simulator has to send a list of atoms that can be executed in a certain state to the animator. The animation must contain for each atom that can be executed a set of commands that add the atom to a list for one of the items (representing one of the processes of the simulated specification) in the animation. A user can select an atom by pointing at an item on which a list of atoms pops up belonging to this item. On selecting one of the atoms from a list, the animator sends the atom to the simulator and executes the set of commands for animation of this atom. In this way, a user can see which atoms can be executed in a certain state and to what item they belong.

The animator is implemented in Tcl/Tk and serves as a framework for the animations. An animation is actually a part of the animator instead of input for the animator. The reason for implementing the animator as a framework is that a user can now add Tcl/Tk commands to the animation. So, if the library of animation functions is not sufficient, a user can always add the needed functionality to the animation.

5.2 Structure of an Animation

An animation is a piece of Tcl/Tk code executed as part of the animator. It should at least consist of a part that builds a picture for the animation and an animation function. For active animation also a choose function is needed. Below we present the structure we use for our animations.

```
initialization

proc ANIM_action {atom} {
    if {match} {
        animation
    } elseif {match} {
        animation
    ...
    }
}

proc ANIM_choose {atom} {
    if {match} {
        add-list
    } elseif {match} {
        add-list
    ...
    }
}
```

The *initialization* consist of a serie of commands from the animation library for building the picture of the animation. The `ANIM_action` function is called by the animator with the executed action sent by the simulator as argument. In the if-elseif construction a *match* is a match of a regular expression with the action to select an *animation* consisting of a serie of commands from the animation library that perform the action. The `ANIM_choose` can be omitted from the animation, but then there is no active animation. The function is called by the animator for each action in the list the animator receives from the simulator. This function is similar to the `ANIM_action` function, but instead of an *animation* commands must be given for adding the action to a list for a particular item from which an action can

be chosen that has to be executed.

5.3 Building an Animation

We describe how to make an animation and the different forms of control, using the Alternating Bit Protocol as an example. A specification of this protocol can be found in Appendix A.1. Most of the lines of code we present here start with line-numbers for reference purposes, they are not part of the code. For an animation we have to initialize the windows first. The command

```
1 Anim::Windows 440 220 61 10
```

gives us the windows shown in Figure 5-1.

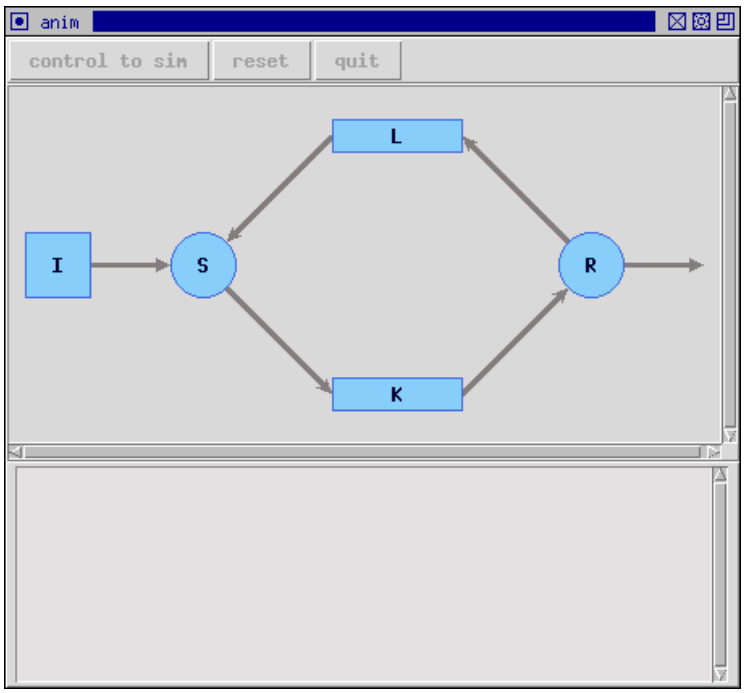


Figure 5-1. Screendump of animation window

The command is built up from the function `Windows` from the animation package `Anim` followed with some arguments indicating the width and height for the windows.

The Figure shows three buttons, which are disabled at the moment. Below that a canvas (with width `440` and height `220` in pixels) where the actual animation takes place, and below that a text-window (with width `61` and height `10` in characters) with additional scrollbar. In the text-window, the atoms that are executed by the simulator are displayed (the same as in the TRACE-window of the simulator when tracing is on).

The picture in the canvas is made with the following commands.

```
 2 Anim::CreateItem recti rect 30 110 20 20 "I"
 3 Anim::CreateItem ovals oval 120 110 20 20 "S"
 4 Anim::CreateItem ovalr oval 360 110 20 20 "R"
 5 Anim::CreateItem rectl rect 240 30 40 10 "L"
 6 Anim::CreateItem rectk rect 240 190 40 10 "K"
 7 Anim::CreateLine toS pos 50 110 item ovals chop -arrow last
 8 Anim::CreateLine fromR item ovalr chop pos 430 110 -arrow last
 9 Anim::CreateLine StoK item ovals se item rectk w -arrow last
10 Anim::CreateLine KtoR item rectk e item ovalr sw -arrow last
11 Anim::CreateLine RtoL item ovalr nw item rectl e -arrow last
12 Anim::CreateLine LtoS item rectl w item ovals ne -arrow last
```

The command in line 2 creates a rectangle (indicated by the second argument `rect`) at the position `30`,`110` (calculated from the top left corner) with width and height both `20`. The actual width and height are twice these sizes. The sizes given here indicate the distance from the position `30`,`110` to the border of the rectangle. (It is done this way to eliminate rounding of numbers in calculations.) The first argument is the name of the rectangle, so that it can be referenced later, and the last argument gives the text to be displayed in the item.

The command in line 7 creates a line with name `toS` from position `50`,`110` (`pos 50 110`) to the border of the item with name `ovals` (`item ovals chop`). And at the end of the line, an arrow is drawn (`-arrow last`).

To display text at some positions later on, we do the following.

```
13 Anim::TextposLine toS toS s
14 Anim::TextposLine fromR fromR s
15 Anim::TextposLine StoK StoK ne
16 Anim::TextposItem atK rectk s n
17 Anim::TextposLine KtoR KtoR nw
18 Anim::TextposLine RtoL RtoL sw
19 Anim::TextposItem atL rectl n s
20 Anim::TextposLine LtoS LtoS se
```

The command in line 13 defines a position for text with the name `toS` (the first argument) at line `toS` (the second argument) and with anchor `s` (south), which means that the south of the text will be placed just above the line. The command in line 16 defines a position with name `atK` at the south of item `rectk` with anchor `n` (north).

5.3.1 Passive Animation

Now we describe the interpretations for the atoms in the trace of the simulator. We do this by defining the function `ANIM_action` as follows.

```
21 proc ANIM_action {atom} {
22    if {[regexp {^input\('(.*)\)$} $atom match arg1]} {
23       Anim::Clear recti
24       Anim::Clear ovals
25       Anim::CreateText toS "$arg1"
26       Anim::ActivateLine toS
27       Anim::AddClear ovals {line toS} {text toS}
28    } elseif {[regexp {^skip frame-comm\(frame\((.*), '(.*)\)\)$} \
29          $atom match arg1 arg2]} {
30       Anim::Clear ovals
31       Anim::CreateText StoK "$arg2 ($arg1)"
32       Anim::ActivateLine StoK
```

```
33        Anim::AddClear rectk {line StoK} {text StoK}
34     } elseif {[regexp {^skip<(0|1)>$} $atom match]} {
35        Anim::Clear rectk
36        Anim::CreateText atK "$match"
37        Anim::AddClear rectk {text atK}
38     } elseif {[regexp \
39           {^skip frame-or-error\(frame\((.*), '(.*)\)\)$} $atom \
40           match arg1 arg2]} {
41        Anim::Clear rectk
42        Anim::CreateText KtoR "$arg2 ($arg1)"
43        Anim::ActivateLine KtoR
44        Anim::AddClear ovalr {line KtoR} {text KtoR}
45     } elseif {[regexp {^skip frame-or-error\(frame-error\)$} $atom \
46           match]} {
47        Anim::Clear rectk
48        Anim::CreateText KtoR "error"
49        Anim::ActivateLine KtoR
50        Anim::AddClear ovalr {line KtoR} {text KtoR}
51     } elseif {[regexp {^output\('(.*)\)$} $atom match arg1]} {
52        Anim::Clear ovalr
53        Anim::CreateText fromR "$arg1"
54        Anim::ActivateLine fromR
55        Anim::AddClear ovalr {line fromR} {text fromR}
56     } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $atom match \
57           arg1]} {
58        Anim::Clear ovalr
59        Anim::CreateText RtoL "ack($arg1)"
60        Anim::ActivateLine RtoL
61        Anim::AddClear rectl {line RtoL} {text RtoL}
62     } elseif {[regexp {^skip<(2|3)>$} $atom match]} {
63        Anim::Clear rectl
64        Anim::CreateText atL "$match"
65        Anim::AddClear rectl {text atL}
66     } elseif {[regexp {^skip ack-or-error\(ack\((.*)\)\)$} $atom \
67           match arg1]} {
68        Anim::Clear rectl
69        Anim::CreateText LtoS "ack($arg1)"
70        Anim::ActivateLine LtoS
71        Anim::AddClear recti {line LtoS} {text LtoS}
72     } elseif {[regexp {^skip ack-or-error\(ack-error\)$} $atom \
73           match]} {
74        Anim::Clear rectl
75        Anim::CreateText LtoS "error"
76        Anim::ActivateLine LtoS
77        Anim::AddClear ovals {line LtoS} {text LtoS}
78     }
79  }
```

We take line 22 as an example of how an atom can be matched. First note that in Tcl the value of a variable with the name *var* is substituted for $*var*. The condition of the `if`-command is enclosed in braces (`{ }`). In Tcl, square brackets (`[ ]`) indicate that the text in between has to be evaluated as a command. Here, this is the matching of a regular expression with the value in the variable `atom`. In the regular expression `^input\('(.*)\)$` the `^` and `$` match with the begin and end of the action in *atom*, so that we match all of *atom* and not just a part of it. The `\(` and `\)` match with a `(` and a `)` respectively. We use `.*` to match with anything and we put it in between `( )` to save the part it matched (this becomes available in the variable with name *arg1*). The other characters match with themselves. The variable with name *match* will contain everything

that has been matched. So in case the atom is `input('a)` the regular expression will match and variable *arg1* gets the value `a`.

The order of the matching of regular expressions can be relevant. For instance, the regular expression `^snd\((.*)\)$` matches the action `snd(f, g)`. In such cases the matching for the action with more arguments has to take place before the matching for the action with fewer arguments. Thus, the matching with regular expression `^snd\((.*), (.*)\)$` has to be placed before the matching with regular expression `^snd\((.*)\)$` in the if-elseif construction.

In line 25 we create a text (the value of `arg1`) on the position `toS` created earlier with the use of `Anim::TextposLine`. The line `toS` is activated in line 26 (on color displays it gets a different color and on monochrome displays it becomes solid).

In line 27 we add the line `toS` and the text `toS` to the clear-list of `ovals`. With the next match of an atom (line 28) we give the order to clear this list for `ovals` (line 30). Instead of line 27 and 30 we also could have done

```
Anim::DeactivateLine toS
Anim::DeleteText toS
```

directly after line 29. The use of clear-lists has the advantage that is not necessary to know what has to be cleared (deactivated and deleted) on the next action in which an item is involved (here `ovals`).

Now let us look at the result of this. After the simulation of the atoms

input('a)
skip frame-comm(frame(0, 'a))

we get the picture in Figure 5-2. In the animation we only display 'a(0)' as communication between `S` and `K` instead of 'skip frame-comm(frame(0, 'a))' (see line 31).

5.3.2 Active Animation

It is also possible to let the animation control the simulation. For this, we have to define a function `ANIM_choose`. When this function is found in the animation, the control is given to the animation automatically and the buttons **`control to sim`**, **`reset`**, and **`quit`** are enabled. The first one gives control to the simulator, that gives us passive animation. The simulator then has a button **`control to anim`** enabled to give control back to the animation. The button **`reset`** performs a re-initialisation of the animation and sends a reset action to the simulator, and the button **`quit`** sends a quit action to the simulator.

```
80 proc ANIM_choose {atom} {
81    if {[regexp {^input\('(.*)\)$} $atom match arg1]} {
82       Anim::AddList recti $match
83    } elseif {[regexp {^skip frame-comm\(frame\((.*), '(.*)\)\)$} \
84          $atom match arg1 arg2]} {
85       Anim::AddList ovals $match
86    } elseif {[regexp {^skip<(0|1)>$} $atom match]} {
87       Anim::AddList rectk $match
88    } elseif {[regexp
```

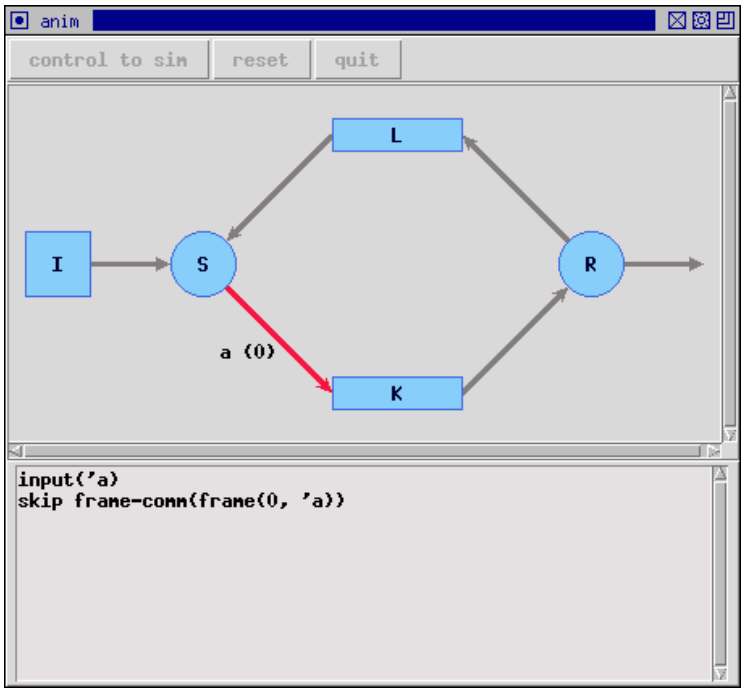


Figure 5-2. Alternating bit protocol: passive animation

```
 89          {^skip frame-or-error\(frame\((.*), '(.*)\)\)$} $atom \
 90          match arg1 arg2]} {
 91        Anim::AddList rectk $match
 92      } elseif {[regexp {^skip frame-or-error\(frame-error\)$} $atom \
 93          match]} {
 94        Anim::AddList rectk $match
 95      } elseif {[regexp {^output\('(.*)\)$} $atom match arg1]} {
 96        Anim::AddList ovalr $match
 97      } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $atom match \
 98          arg1]} {
 99        Anim::AddList ovalr $match
100      } elseif {[regexp {^skip<(2|3)>$} $atom match]} {
101        Anim::AddList rectl $match
102      } elseif {[regexp {^skip ack-or-error\(ack\((.*)\)\)$} $atom \
103          match arg1]} {
104        Anim::AddList rectl $match
105      } elseif {[regexp {^skip ack-or-error\(ack-error\)$} $atom \
106          match]} {
107        Anim::AddList rectl $match
108      }
109  }
```

For each atom in the choose-list of the simulator the above function is called. Each item in the animation has its own choose-list. When there are atoms added to a list with the use of `Anim::AddList`, the item becomes activated (on color displays it gets a different color and on monochrome displays it becomes stippled). When an activated item is clicked upon with the mouse, a list pops up from which an atom can be selected for execution. Leaving

the list with the mouse makes the list disappear. So the lists can be examined without making a selection. A snapshot of active animation is shown in Figure 5-3.

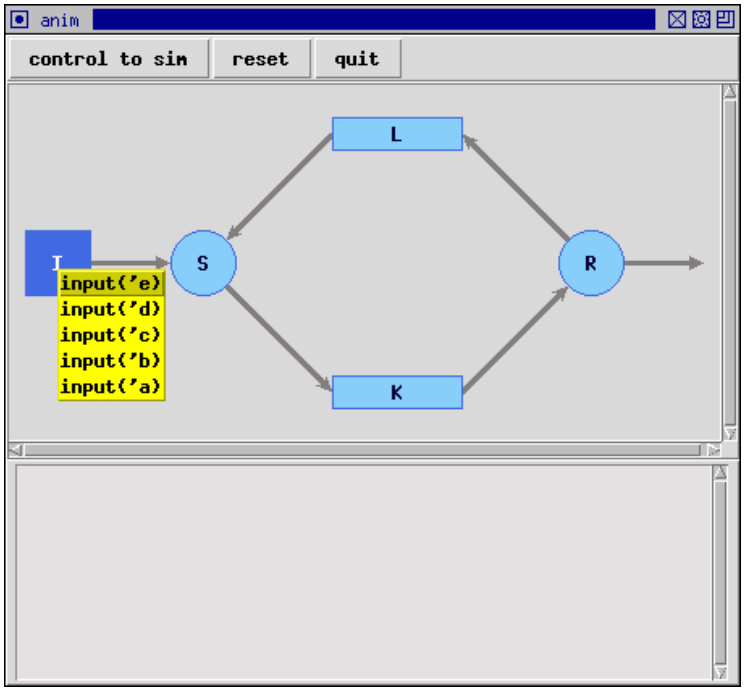


Figure 5-3. Alternating bit protocol: active animation

5.4 Visual Attractive Animation

The animation functions shown so far, are satisfactory for displaying processes and their communications. However, more can be done to make the animations more attractive, such as moving items, queues, display counters on an information panel, etc.

Here, a few features are shown of which the ones mentioned above are the most important. For this, we use the specification of a small factory consisting of input, output, some stations and conveyor belts. It produces the products A and B which take slightly different routes through the factory. A specification of this factory can be found in Appendix A.2.

We first give the commands for the picture in the canvas of the animation.

```
1 Anim::Windows 340 200 30 10
2 Anim::CreateItem inp rect 30 30 15 15 "In"
3 Anim::CreateItem s1 rect 30 100 15 15 "1"
4 Anim::CreateItem s2 rect 100 100 15 15 "2"
5 Anim::CreateItem s3 rect 170 100 15 15 "3"
6 Anim::CreateItem s4 rect 240 100 15 15 "4"
7 Anim::CreateItem s5 rect 240 170 15 15 "5"
8 Anim::CreateItem s6 rect 310 170 15 15 "6"
9 Anim::CreateItem out rect 310 100 15 15 "Out"
```

```
10 Anim::CreateLine ins1 item inp s item s1 n -arrow last
11 Anim::TextposLine ins1 ins1 e
12 Anim::CreateLine outs6 item s6 n item out s -arrow last
13 Anim::TextposLine outs6 outs6 w
14 Anim::CreateLine s1s2 item s1 e item s2 w -width 15
15 Anim::CreateLine s2s3 item s2 e item s3 w -width 15
16 Anim::CreateLine s3s4 item s3 e item s4 w -width 15
17 Anim::CreateLine s3s5 item s3 s pos [Anim::Dim s3 x] \
18    [Anim::Dim s5 y] item s5 w -width 15
19 Anim::CreateLine s4s5 item s4 s item s5 n -width 15
20 Anim::CreateLine s5s6 item s5 e item s6 w -width 15
```

This gives us the picture in Figure 5-4.

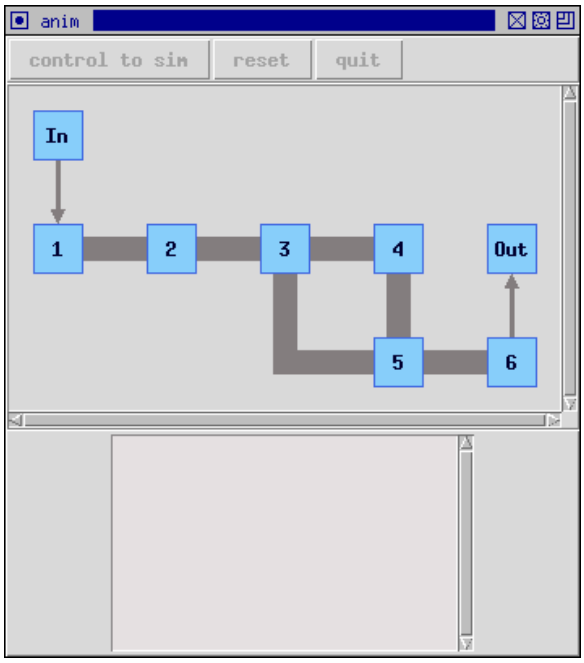


Figure 5-4. Factory

In line 17, we see the use of function `Anim::Dim`. It is used the get a dimension from its first argument (here, the x-coordinate of item `s3` and the y-coordinate of item `s5`). The square brackets around it are to let Tcl/Tk know it has to call the function. It is also possible to do more calculations, for example with the use of the Tcl/Tk function `expr` like this

```
[expr [Anim::Dim s3 x] * 2 + 5]
```

which takes the x-coordinate of `s3`, multiplies it by 2 and adds 5 to it.

5.4.1 Moving Items

Instead of showing that a product is moved from one station to another by means of an

arrow and some text, we actually want to see it moving over the conveyor belt. We define the function `ANIM_action` as follows.

```
21 proc ANIM_action {atom} {
22    if {[regexp {^input\((.*)\)$} $atom match arg1]} {
23       Anim::CreateText ins1 "$arg1"
24       Anim::ActivateLine ins1
25    } elseif {[regexp {^comm-input\((.*)\)$} $atom match arg1]} {
26       Anim::DeleteText ins1
27       Anim::DeactivateLine ins1
28       Anim::CreateItem AT1 rect [Anim::Dim s1 x] [Anim::Dim s1 y] \
29          7 7 "$arg1" -free -color 1
30    } elseif {[regexp {^comm-belt\(3, 4, .*\)$} $atom match arg1]} {
31       Anim::Move AT3 rightto [Anim::Dim s4 x] -newid AT4
32    } elseif {[regexp {^comm-belt\(3, 5, .*\)$} $atom match arg1]} {
33       Anim::Move AT3 downto [Anim::Dim s5 y] rightto \
34          [Anim::Dim s5 x] -newid AT5
35    } elseif {[regexp {^comm-belt\(4, 5, .*\)$} $atom match arg1]} {
36       Anim::Move AT4 downto [Anim::Dim s5 y] -newid AT5
37    } elseif {[regexp {^comm-belt\((.*), (.*), .*\)$} $atom match \
38          arg1 arg2]} {
39       Anim::Move AT$arg1 rightto [Anim::Dim s$arg2 x] \
40          -newid AT$arg2
41    } elseif {[regexp {^comm-output\((.*)\)$} $atom match arg1]} {
42       Anim::DestroyItem AT6
43       Anim::CreateText outs6 "$arg1"
44       Anim::ActivateLine outs6
45    } elseif {[regexp {^output\((.*)\)$} $atom match arg1]} {
46       Anim::DeleteText outs6
47       Anim::DeactivateLine outs6
48    }
49 }
```

Line 31 shows how we move a product from station 3 to station 4. With the option `-newid` we give it a new name. In this way, we do not have to keep track of which item is at what position (the name of the item indicates its location).

In lines 28 and 29, items are created with the options `-free` and `-color`. The option `-free` indicates that this item has to be freed (destroyed) on a reset. The option `-color` *x* indicates that the color for the item must come from colorset *x*, where *x* can be either 0 or 1, or a colorset created with the function `Anim::Colorset`. A snapshot of this passive animation is shown in Figure 5-5.

The function for active animation is given below.

```
50 proc ANIM_choose {atom} {
51    if {[regexp {^input\((.*)\)$} $atom match arg1]} {
52       Anim::AddList inp $match
53    } elseif {[regexp {^comm-input\((.*)\)$} $atom match arg1]} {
54       Anim::AddList s1 $match
55    } elseif {[regexp {^comm-belt\((.*), (.*), .*\)$} $atom match \
56          arg1 arg2]} {
57       Anim::AddList AT$arg1 $match
58    } elseif {[regexp {^comm-output\((.*)\)$} $atom match arg1]} {
59       Anim::AddList s6 $match
60    } elseif {[regexp {^output\((.*)\)$} $atom match arg1]} {
61       Anim::AddList out $match
62    }
63 }
```

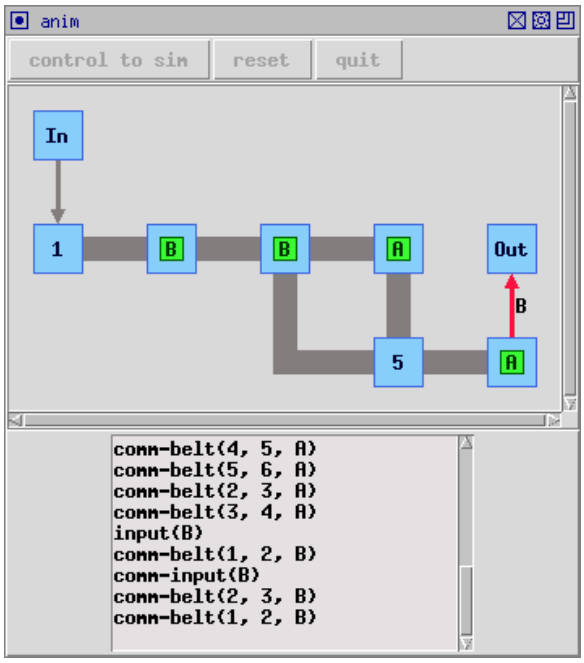


Figure 5-5. Factory: passive animation

5.4.2 Queues

Now, we extend our specification of the factory with input- and output-queues. In the animation, we replace line 2 with

```
Anim::CreateQueue qin 25 30 13 1 -anchor w
```

and line 9 with

```
Anim::CreateQueue qout 310 115 1 7 -orient vertical -anchor s
```

This gives us a horizontal input-queue of `13` characters long and `1` character high, at position `25,30`. By using the option `-orient vertical` a vertical output-queue is created.

This is enough for passive animation. However, for active animation we need an item on both sides of the queue in order to control the input and output of the queue. We now replace line 2 with

```
Anim::CreateItem qin-out rect 22 30 7 15 ""
Anim::CreateQueue qin [Anim::Dim qin-out e,x] 30 10 1 -anchor w
Anim::CreateItem qin-in rect [expr [Anim::DimQ qin e,x] + 7] 30 7 \
   15 "In"
```

and line 9 with

```
Anim::CreateItem qout-in rect [Anim::Dim s6 x] 107 12 8 ""
Anim::CreateQueue qout [Anim::Dim qout-in x] \
```

```
      [Anim::Dim qout-in n,y] 1 5 -orient vertical -anchor s
   Anim::CreateItem qout-out rect [Anim::DimQ qout x] \
      [expr [Anim::DimQ qout n,y] - 8] 12 8 "Out"
```

The code for passive and active animation is given below

```
 64 proc ANIM_action {atom} {
 65    if {[regexp {^q-input\((.*)\)$} $atom match arg1]} {
 66       Anim::AddQueue qin $arg1
 67    } elseif {[regexp {^comm-q-input\((.*)\)$} $atom match arg1]} {
 68       Anim::SubQueue qin
 69       Anim::CreateText ins1 $arg1
 70       Anim::ActivateLine ins1
 71    } elseif {[regexp {^comm-input\((.*)\)$} $atom match arg1]} {
 72       Anim::DeleteText ins1
 73       Anim::DeactivateLine ins1
 74       Anim::CreateItem AT1 rect [Anim::Dim s1 x] [Anim::Dim s1 y] \
 75       7 7 "$arg1" -free -color 1
 76    } elseif {[regexp {^comm-belt\(3, 4, .*\)$} $atom match arg1]} {
 77       Anim::Move AT3 rightto [Anim::Dim s4 x] -newid AT4
 78    } elseif {[regexp {^comm-belt\(3, 5, .*\)$} $atom match arg1]} {
 79       Anim::Move AT3 downto [Anim::Dim s5 y] rightto \
 80          [Anim::Dim s5 x] -newid AT5
 81    } elseif {[regexp {^comm-belt\(4, 5, .*\)$} $atom match arg1]} {
 82       Anim::Move AT4 downto [Anim::Dim s5 y] -newid AT5
 83    } elseif {[regexp {^comm-belt\((.*), (.*), .*\)$} $atom match \
 84          arg1 arg2]} {
 85       Anim::Move AT$arg1 rightto [Anim::Dim s$arg2 x] \
 86          -newid AT$arg2
 87    } elseif {[regexp {^comm-output\((.*)\)$} $atom match arg1]} {
 88       Anim::DestroyItem AT6
 89       Anim::CreateText outs6 "$arg1"
 90       Anim::ActivateLine outs6
 91    } elseif {[regexp {^comm-q-output\((.*)\)$} $atom match arg1]} {
 92       Anim::DeleteText outs6
 93       Anim::DeactivateLine outs6
 94       Anim::AddQueue qout $arg1
 95    } elseif {[regexp {^q-output\((.*)\)$} $atom match arg1]} {
 96       Anim::SubQueue qout
 97    }
 98 }
 99 proc ANIM_choose {atom} {
100    if {[regexp {^q-input\((.*)\)$} $atom match arg1]} {
101       Anim::AddList qin-in $match
102    } elseif {[regexp {^comm-q-input\((.*)\)$} $atom match arg1]} {
103       Anim::AddList qin-out $match
104    } elseif {[regexp {^comm-input\((.*)\)$} $atom match arg1]} {
105       Anim::AddList s1 $match
106    } elseif {[regexp {^comm-belt\((.*), (.*), .*\)$} $atom match \
107          arg1 arg2]} {
108       Anim::AddList AT$arg1 $match
109    } elseif {[regexp {^comm-output\((.*)\)$} $atom match arg1]} {
110       Anim::AddList s6 $match
111    } elseif {[regexp {^comm-q-output\((.*)\)$} $atom match arg1]} {
112       Anim::AddList qout-in $match
113    } elseif {[regexp {^q-output\((.*)\)$} $atom match arg1]} {
114       Anim::AddList qout-out $match
115    }
116 }
```

A snapshot of this is shown in Figure 5-6.

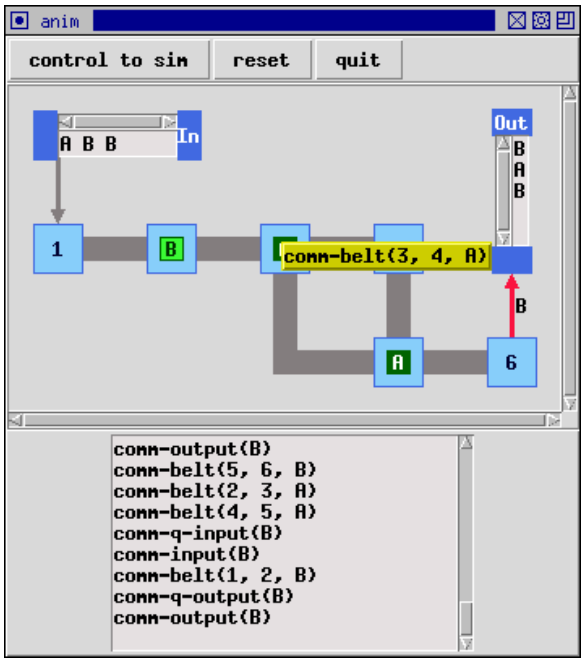


Figure 5-6. Factory with queues: active animation

5.4.3 Information Panel

In order to get an even better view, support for accounting is added. If we want to display the lengths of the queues and the amount of input and output of the factory, we can add the following code.

```
117 Anim::CreateBox info queues -side top -ipadx 1 -ipady 1 -expand \
118    -bw 2 -relief ridge
119 Anim::CreateBox queues queueinput -side left
120 Anim::CreateLabel queueinput inputtext "queue In" -width 9 \
121    -anchor w
122 Anim::CreateLabel queueinput inputvar q-input -var -bw 2 \
123    -relief sunken -width 2
124 Anim::CreateBox queues queueoutput -side left
125 Anim::CreateLabel queueoutput outputtext "queue Out" -width 9 \
126    -anchor w
127 Anim::CreateLabel queueoutput outputvar q-output -var -bw 2 \
128    -relief sunken -width 2
129 Anim::InitVar q-input 0
130 Anim::InitVar q-output 0
131 Anim::CreateBox info table -side top -bw 2 -relief ridge
132 Anim::CreateBox table header -side left
133 Anim::CreateLabel header col0 "" -width 6
134 Anim::CreateLabel header col1 "A" -width 2
135 Anim::CreateLabel header col2 "B" -width 2
136 Anim::CreateBox table row1 -side left
137 Anim::CreateLabel row1 input input -width 6 -anchor w
138 Anim::CreateLabel row1 inpA input(A) -var -width 2 -bw 2 \
```

```
139    -relief sunken
140 Anim::CreateLabel row1 inpB input(B) -var -width 2 -bw 2 \
141    -relief sunken
142 Anim::CreateBox table row2 -side left
143 Anim::CreateLabel row2 output output -width 6 -anchor w
144 Anim::CreateLabel row2 outpA output(A) -var -width 2 -bw 2 \
145    -relief sunken
146 Anim::CreateLabel row2 outpB output(B) -var -width 2 -bw 2 \
147    -relief sunken
148 Anim::InitArray input [list A 0 B 0]
149 Anim::InitArray output [list A 0 B 0]
```

At line 117, a box is created with the name `queues` and parent `info`. Box `info` is predefined and is normally empty. In that box we create the boxes `queueinput` and `queueoutput`. In box `queueinput` we create two labels, one which contains text and one which will contain the last value assigned to variable `q-input` (this is indicated with the option `-var`). Variables must be initialized with the use of function `Anim::InitVar,` in order to initialize them again after a reset.

In box `info` also a box `table` is made. In this box we display the arrays `input` and `output`, which must be initialized with function `Anim::InitArray`.

Now, in the function `ANIM_action` one can assign values to these variables with either the **set** or the **incr** command of Tcl. We insert after line 65 the commands

```
incr q-input
incr input($arg1)
```

after line 68

```
incr q-input -1
```

after line 94

```
incr q-output
```

and after line 96

```
incr q-output -1
incr output($arg1)
```

Unfortunately, in Tcl these variables must be declared to be global in the function `ANIM_action`. We do this by inserting

```
global q-input q-output input output
```

after line 64. A snapshot of the animation with information panel is shown in Figure 5-7.

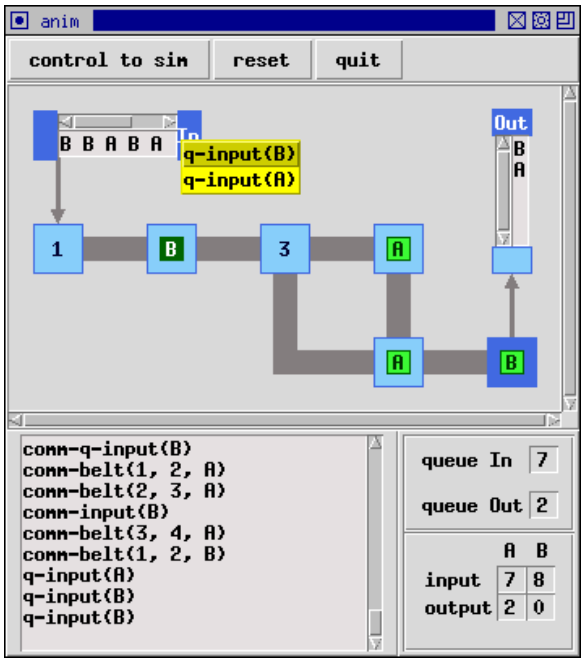


Figure 5-7. Factory with info-panel: active animation

Chapter 6

Generation of Animations

In the previous chapter a platform is presented for simulation and animation of process algebra specifications. These animations have to be created by hand. So whenever the specification changes, the animation has to be adapted. This makes it difficult to use it for testing, especially for larger specifications.

We try to overcome this problem by generating animations from the specifications. In using static analysis of the specification for the generation of animation we have to deal with open terms, i.e. terms that contain variables, causing problems when term matching is involved. Dynamic analysis of the specification can only solve these problems in some cases because in general all possible executions have to be considered to be sure that all characteristics of the specification are encountered. Here, we use static analyse to find out to what extent we can generate animations and how we have to adapt the specifications in order to get better results.

We have divided the problem of generating an animation from a specification into several steps. First, we have to analyse the specification with as result a process graph and a list consisting of the atoms that are part of the processes in the graph and of the communications that can occur between the processes.
Secondly, we have to convert the process graph into a picture. We use the program *dot*,[2] which calculates coordinates for nodes and edges of a graph. We generate an animation from the output of *dot* by a Perl [67] script.
Thirdly, we generate an action-function and a choose-function from the list of atoms and communications.

We explain the implementation of the various parts in general and use a specification of the Alternating Bit Protocol in PSF as an example. The specification of this protocol can be found in Appendix A.1.

2. Dot is part of the software package *Graphviz* from AT&T Bell Laboratories.

6.1 Generation of a Process Graph

We describe here the steps that we make in order to generate a graph from a specification. Several steps could have been incorporated, but we have chosen to keep our code as simple as possible.

Build Process Tree

For each definition of a process we build a process tree in which the nodes represent the operators and processes, and the edges represent a list of atoms. These lists of atoms eliminate the sequential operator (.).

Expand Tree

We take the process tree for the top process and expand it, by replacing the processes with their process tree. This is done recursively, but a process is only expanded once in a tree since we have all possible actions that can occur already in this tree. Except for the subtrees of a parallel operator (||), in which a process may be expanded in each subtree, so that possible communications can be found.

Mark up Tree

Put ID on Processes

Give the top process of the tree and of the subtrees of a node that represents a parallel operator, an ID. Mark all atoms with the process-ID of the subtree it belongs to.

Find Sum Atoms

We mark all atoms that can act as a sum-port (see section 3.3.4). These are the atoms first in the list of atoms belonging to the edge from the node for the sum operator to its subtree, and that have the variable of this sum operator in one of their arguments.

This information is later used in deciding the type of the communications.

Encapsulate and Hide Atoms

We also mark the atoms that will be encapsulated or hidden. This information will be used later in calculation of the communications. (We are matching open terms, so this can result in not detecting an atom as a member of a set.)

Find Communications

We go down the tree to the leaf nodes. From there we go up and list the atoms we encounter. When we meet an encapsulation operator, we delete the atoms from our list that are encapsulated by this operator. When we meet a hide operator, we mark the atoms that are hidden by this operator. When we meet a parallel operator, we calculate the possible communications between the atoms from the list for each subtree, and assign this list of communications to this node. We also decide on the direction of the communications. When the left communication partner can act as a sum-port we indicate this with '<-', for the right communication partner we use '->', for both '<->', and '-' for none.

Back at the top, we have collected a list of all atoms that can be performed.

Encapsulate and Hide Communications

We mark the communications that will be encapsulated or hidden.

Collect the communications

We go down in the tree and on our way up we list the communications. When we meet an encapsulation operator, we remove the communications that are encapsulated by this operator from the list.
At the top, we have collected a list of all possible communications.

Properties of the Processes

By inspecting the list of atoms and communications, we can see which of the processes are used. There is no need to put processes in the graph that are not used. However, for debugging purposes this is made optional.

We consider processes which contain atoms that are part of sum-constructions and that are not hidden, input-processes. We also consider processes which contain atoms that are not hidden, output-processes. We want to mark them as such, so that we can try to put the input-processes at the top and the output-processes at the bottom in our animation.
From the list of atoms we can decide which are the input-processes and output-processes.

Print Graph

We start with creating a node called 'Input' to which we can connect the input-processes.
Then we traverse our graph and create a node for every process that has got an ID and that is used. When we encounter a node that represents an encapsulation, we start a subgraph. If the node has a list of communications, we create edges between the processes that take part in a communication in this list. These edges are directed according to the communication. If a side of a communication is a sum-construction, it gets an arrow. Care is taken to not create multiple edges between two processes that have the same direction.
We create a node called 'Output' to which we can connect the output-processes, and we create the edges between the input and output nodes.

```
digraph ABP {
    node [color=lightblue]
    node [style=filled]
    subgraph clusterinput { I [label="Input", color=green]; }
    subgraph cluster {
        subgraph cluster1 {
            { rank=min; n4 [label="Sender"]; }
            { rank=max; n5 [label="Receiver"]; }
            n6 [label="K"];
            n6 → n5 [dir=forward];
            n5 → n6 [dir=none];
            n4 → n6 [dir=forward];
            n7 [label="L"];
            n5 → n7 [dir=forward];
            n4 → n7 [dir=none];
        }
    }
    subgraph clusteroutput { O [label="Output", color=green]; }
```

```
    I → n4 [dir=forward, label=""];
    n5 → O [dir=forward, label=""];
}
```

Print Communication List

For each communication in our list, we print 'skip' if it is marked as hidden, the communication itself followed by the IDs of the processes which cause this communication with a direction (either '-', '->', '<-', or '<->') in between.

```
skip frame-or-error(frame(!b!, !d!)) 6  -> 5
skip frame-or-error(frame-error) 5  -  6
skip frame-comm(frame(!b!, !d!)) 4  -> 6
skip ack-comm(ack(!b!)) 5  -> 7
skip ack-comm(ack(!b!)) 5  -> 7
skip ack-or-error(ack(!b!)) 4  -  7
skip ack-or-error(ack-error) 4  -  7
```

Note that we put variable names inside '!', so that we can recognize them as variables later on.

Print Atom List

For each atom in our list, we print 'skip' if it is marked as hidden followed by the atom itself, and if it not marked as hidden, then we print the atom followed by 'I ->' and the ID of the process it belongs to, if it is an input-process and the ID of the process and '-> O', if it is an output-process.

```
input(!d!) I  -> 4
output(!d!) 5  -> O
skip<0> 6
skip<1> 6
skip<2> 7
skip<3> 7
```

6.2 Generation of an Animation

If we apply the program *dot* on the generated graph that is shown above, we get the following output (line-numbers are not part of the output).

```
 1 digraph ABP {
 2    node [label = "\N",
 3       color = lightblue,
 4       style = filled ];
 5    graph [lp= "81,0"];
 6    graph [bb= "0,0,162,342"];
 7    subgraph clusterinput {
 8       graph [lp= ""];
 9       graph [bb= "45,288,117,342"];
10       I [label=Input, color=green, pos="81,315", width="0.75",
11          height="0.50"];
12    }
13    subgraph cluster {
14       graph [lp= ""];
15       graph [bb= "0,63,162,279"];
16       subgraph cluster1 {
17          graph [bb= "9,72,153,270"];
18          {
19             graph [rank= min];
20             graph [bb= ""];
```

```
21            n4 [label=Sender, pos="81,243", width="0.81",
22               height="0.50"];
23         }
24         {
25            graph [rank= max];
26            graph [bb= ""];
27            n5 [label=Receiver, pos="53,99", width="0.97",
28               height="0.50"];
29         }
30         n6 [label=K, pos="45,171", width="0.75", height="0.50"];
31         n7 [label=L, pos="117,171", width="0.75", height="0.50"];
32         n6 → n5 [dir=forward,
33            pos="e,45,117 41,153 41,145 42,135 43,127"];
34         n5 → n6 [dir=none, pos="53,154 55,143 57,128 57,117"];
35         n4 → n6 [dir=forward,
36            pos="e,54,188 72,226 68,217 63,207 58,197"];
37         n5 → n7 [dir=forward,
38            pos="s,103,155 99,150 89,139 77,125 68,115"];
39         n4 → n7 [dir=none, pos="90,226 95,214 103,200 108,188"];
40      }
41   }
42   subgraph clusteroutput {
43      graph [lp= ""];
44      graph [bb= "14,0,92,54"];
45      O [label=Output, color=green, pos="53,27", width="0.83",
46         height="0.50"];
47   }
48   I → n4 [dir=forward,
49      pos="e,81,261 81,297 81,289 81,280 81,271"];
50   n5 → O [dir=forward, pos="e,53,45 53,81 53,73 53,64 53,55"];
51 }
```

The positions are in default units, 1/72 of an inch, and widths and heights are in inches. These have to be converted to pixels, which usually are 75 pixels per inch.

From the bounding-box in line 6, we derive the size we have to use for the window that will contain the picture. That gives us the following line.

```
Anim::Windows 188 376 -text 60 10
```

The last part gives us a text-window of width 60 and height 10 for printing the actions that are performed in the animation.

For the bounding-box in line 16 we draw a box in our picture.

```
Anim::CreateLine box1 pos 19 291 pos 169 291 pos 169 85 pos 19 85 \
   pos 19 291 -width 1
```

We do this only for bounding-boxes belonging to a subgraph with the name *cluster* followed by a number. These represent the encapsulations in the specification.

We create the nodes and define for each node a text-position at which the atoms belonging to this node will be printed.

```
Anim::CreateItem I oval 94 38 28 18 "Input" -anchor s -color 1
Anim::TextposItem textI I ce n
Anim::CreateItem n4 oval 94 113 30 18 "Sender" -anchor s -color 0
Anim::TextposItem textn4 n4 ce n
Anim::CreateItem n5 oval 65 263 36 18 "Receiver" -anchor s \
   -color 0
```

```
Anim::TextposItem textn5 n5 ce n
Anim::CreateItem n6 oval 56 188 28 18 "K" -anchor s -color 0
Anim::TextposItem textn6 n6 ce n
Anim::CreateItem n7 oval 131 188 28 18 "L" -anchor s -color 0
Anim::TextposItem textn7 n7 ce n
Anim::CreateItem O oval 65 338 31 18 "Output" -anchor s -color 1
Anim::TextposItem textO O ce n
```

We also create the edges and define text-positions for them, at which the communications will be printed.

```
Anim::CreateLine linen6ton5 pos 52 206 pos 52 215 pos 53 225 \
   pos 54 233 pos 56 244 -arrow last -smooth
Anim::Textpos textn6ton5 53 225 ce
Anim::CreateLine linen5ton6 pos 65 205 pos 67 217 pos 69 232 \
   pos 69 244 -arrow none -smooth
Anim::Textpos textn5ton6 68 224 ce
Anim::CreateLine linen4ton6 pos 85 130 pos 80 140 pos 75 150 \
   pos 70 161 pos 66 170 -arrow last -smooth
Anim::Textpos textn4ton6 75 150 ce
Anim::CreateLine linen5ton7 pos 117 204 pos 113 210 pos 102 221 \
   pos 90 236 pos 80 246 -arrow first -smooth
Anim::Textpos textn5ton7 102 221 ce
Anim::CreateLine linen4ton7 pos 103 130 pos 108 143 pos 117 157 \
   pos 122 170 -arrow none -smooth
Anim::Textpos textn4ton7 112 150 ce
Anim::CreateLine lineIton4 pos 94 56 pos 94 65 pos 94 74 pos 94 83 \
   pos 94 94 -arrow last -smooth
Anim::Textpos textIton4 94 74 ce
Anim::CreateLine linen5toO pos 65 281 pos 65 290 pos 65 299 \
   pos 65 308 pos 65 319 -arrow last -smooth
Anim::Textpos textn5toO 65 299 ce
```

This results in the picture given in Figure 6-1.

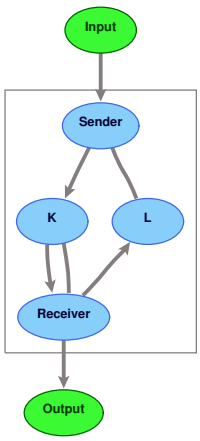


Figure 6-1. Alternating Bit Protocol

Note the two lines between node *K* and node *Receiver*. We could not determine the direction of the communication of one of them. Also, the arrow between the nodes should represent two different communications, but we found only one. Let's take a look at the

process-definitions for *Receiver*.

```
Receiver = Receive-Frame(0)
Receive-Frame(b) = (
            sum(d in DATA,
               receive-frame-or-error(frame(flip(b),d)))
         +  receive-frame-or-error(frame-error)
         ) . Send-Ack(flip(b))
      +  sum(d in DATA, receive-frame-or-error(frame(b,d)) .
            Send-Message(b,d)
         )
Send-Ack(b) = send-ack(ack(b)) . Receive-Frame(flip(b))
Send-Message(b,d) = output(d) . Send-Ack(b)
```

The three candidates to communicate here are:

```
receive-frame-or-error(frame(flip(b),d))
receive-frame-or-error(frame-error)
receive-frame-or-error(frame(b,d))
```

For the first one, we cannot find a communication. The supposed communication partner is `send-frame-or-error(frame(b,d))` from *K*. Static analysis is not sufficient in this case, because we need to rewrite the terms with all possible values for the variables to decide if a communication is possible between the two. In the general case this is not possible at all, since there may be an infinite number of possible values for a variable. To inform the user, we give a warning whenever an atom is encapsulated for which we could not find a possible communication.
To solve this, we give another definition for the process *Receive-Frame*.

```
Receive-Frame(b) =
      sum(f in FRAME,
         receive-frame-or-error(f) . (
            [flip(frame-bit(f)) = b] → Send-Ack(flip(b))
         +  [f = frame-error] → Send-Ack(flip(b))
         +  [frame-bit(f) = b] → Send-Message(b, frame-data(f))
         )
      )
```

We introduced here the functions *frame-bit* and *frame-data*, which extract the concerning fields from the frame. This does not only solve the problem, it also makes the definition much clearer.

The same applies for the communication between the nodes *L* and *Sender*, so we redefine the process *Receive-Ack* in the same manner.

```
Receive-Ack(b,d) =
      sum(a in ACK,
         receive-ack-or-error(a) . (
            [flip(ack-bit(a)) = b] → Send-Frame(b, d)
         +  [a = ack-error] → Send-Frame(b, d)
         +  [ack-bit(a) = b] → Receive-Message(flip(b))
         )
      )
```

Now we can determine the direction of the communication between *L* and *Sender*. This results in the picture in Figure 6-2.

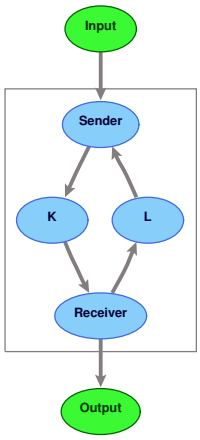


Figure 6-2. Alternating Bit Protocol (adjusted)

6.2.1 Generating the Action Function

From the list of communications and the list of atoms we derive the function which does the animation for these actions.

```
proc ANIM_action {line} {
   if {[regexp {^skip frame-or-error\(frame\((.*), (.*)\)\)$} \
         $line match]} {
      Anim::Clear n6
      Anim::CreateText textn6ton5 "$match"
      Anim::ActivateLine linen6ton5
      Anim::AddClear n5 {line linen6ton5} {text textn6ton5}
   } elseif {[regexp {^skip frame-or-error\(frame-error\)$} $line \
         match]} {
      Anim::Clear n6
      Anim::Clear n5
      Anim::CreateText textn5ton6 "$match"
      Anim::ActivateLine linen5ton6
      Anim::AddClear n5 {line linen5ton6} {text textn5ton6}
      Anim::AddClear n6 {line linen5ton6} {text textn5ton6}
   } elseif {[regexp {^skip frame-comm\(frame\((.*), (.*)\)\)$} \
         $line match]} {
      Anim::Clear n4
      Anim::CreateText textn4ton6 "$match"
      Anim::ActivateLine linen4ton6
      Anim::AddClear n6 {line linen4ton6} {text textn4ton6}
   } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
      Anim::Clear n5
      Anim::CreateText textn5ton7 "$match"
      Anim::ActivateLine linen5ton7
      Anim::AddClear n7 {line linen5ton7} {text textn5ton7}
   } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
      Anim::Clear n5
      Anim::CreateText textn5ton7 "$match"
      Anim::ActivateLine linen5ton7
      Anim::AddClear n7 {line linen5ton7} {text textn5ton7}
   } elseif {[regexp {^skip ack-or-error\(ack\((.*)\)\)$} $line \
```

```
            match]} {
         Anim::Clear n7
         Anim::Clear n4
         Anim::CreateText textn4ton7 "$match"
         Anim::ActivateLine linen4ton7
         Anim::AddClear n4 {line linen4ton7} {text textn4ton7}
         Anim::AddClear n7 {line linen4ton7} {text textn4ton7}
      } elseif {[regexp {^skip ack-or-error\(ack-error\)$} $line \
            match]} {
         Anim::Clear n7
         Anim::Clear n4
         Anim::CreateText textn4ton7 "$match"
         Anim::ActivateLine linen4ton7
         Anim::AddClear n4 {line linen4ton7} {text textn4ton7}
         Anim::AddClear n7 {line linen4ton7} {text textn4ton7}
      } elseif {[regexp {^input\((.*)\)$} $line match]} {
         Anim::Clear I
         Anim::CreateText textIton4 "$match"
         Anim::ActivateLine lineIton4
         Anim::AddClear n4 {line lineIton4} {text textIton4}
      } elseif {[regexp {^output\((.*)\)$} $line match]} {
         Anim::Clear n5
         Anim::CreateText textn5toO "$match"
         Anim::ActivateLine linen5toO
         Anim::AddClear n5 {line linen5toO} {text textn5toO}
      } elseif {[regexp {^skip<0>$} $line match]} {
         Anim::Clear n6
         Anim::CreateText textn6 "$match"
         Anim::AddClear n6 {text textn6}
      } elseif {[regexp {^skip<1>$} $line match]} {
         Anim::Clear n6
         Anim::CreateText textn6 "$match"
         Anim::AddClear n6 {text textn6}
      } elseif {[regexp {^skip<2>$} $line match]} {
         Anim::Clear n7
         Anim::CreateText textn7 "$match"
         Anim::AddClear n7 {text textn7}
      } elseif {[regexp {^skip<3>$} $line match]} {
         Anim::Clear n7
         Anim::CreateText textn7 "$match"
         Anim::AddClear n7 {text textn7}
      }
   }
```

6.2.2 Generating the Choose Function

From the list of communications and the list of atoms we also derive the function for the construction of the choose-lists for active animation. This looks the same as the action-function except for the parts inside the if-else construction.

```
   proc ANIM_choose {line} {
      if {[regexp {^skip frame-or-error\(frame\((.*), (.*)\)\)$} \
            $line match]} {
         Anim::AddList n6 $match
      } elseif {[regexp {^skip frame-or-error\(frame-error\)$} $line \
            match]} {
         Anim::AddList n6 $match
         Anim::AddList n5 $match
      } elseif {[regexp {^skip frame-comm\(frame\((.*), (.*)\)\)$} \
            $line match]} {
```

```
        Anim::AddList n4 $match
    } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
        Anim::AddList n5 $match
    } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
        Anim::AddList n5 $match
    } elseif {[regexp {^skip ack-or-error\(ack\((.*)\)\)$} $line \
            match]} {
        Anim::AddList n7 $match
        Anim::AddList n4 $match
    } elseif {[regexp {^skip ack-or-error\(ack-error\)$} $line \
            match]} {
        Anim::AddList n7 $match
        Anim::AddList n4 $match
    } elseif {[regexp {^input\((.*)\)$} $line match]} {
        Anim::AddList I $match
    } elseif {[regexp {^output\((.*)\)$} $line match]} {
        Anim::AddList n5 $match
    } elseif {[regexp {^skip<0>$} $line match]} {
        Anim::AddList n6 $match
    } elseif {[regexp {^skip<1>$} $line match]} {
        Anim::AddList n6 $match
    } elseif {[regexp {^skip<2>$} $line match]} {
        Anim::AddList n7 $match
    } elseif {[regexp {^skip<3>$} $line match]} {
        Anim::AddList n7 $match
    }
}
```

6.3 Complexities and Features

In the generation of an animation for a specification some complex issues can arise. We describe these issues and how to cope with them. Some solutions demand a certain specification style and others are implemented as features of the animation generation tool.

6.3.1 Merge

In order to show how we deal with the generalized merge, we consider a specification of a small factory consisting of six stations connected by conveyer belts, with an input and an output. It produces two products which take slightly different routes through the factory. The complete specification can be found in Appendix A.2. Here we show the process definitions for the stations.

```
Stations = merge(s in STATION-set, Station(s))
Station(s) =
        [eq-stat(s, 1) = true] → (
            sum(p in PRODUCT,
                read-input(p) . to-belt(s, next(s, p), p)
            ) . Station(s)
        )
      + [eq-stat(s, 6) = true] → (
            sum(p in PRODUCT,
                from-belt(s, p) . send-output(p)
            ) . Station(s)
        )
      + [and(not(eq-stat(s, 1)), not(eq-stat(s, 6))) = true] → (
            sum(p in PRODUCT,
                from-belt(s, p) .
```

```
            to-belt(s, next(s, p), p)
        ) . Station(s)
    )
```

If we simply expand the merge as many times as there are elements in the set *STATION-set*, we end up with six stations that can all communicate with each other. But we want only the communications that really represent a conveyer belt. We could do a better job if the conditional expressions do not contain a variable, so we can evaluate them and disregard the following process expression on a negative result.

So, we have to expand the merge for each element of the set with this element filled in for the variable of the sum operator, and replace every occurrence of a variable with its value, whether it is a variable of a sum operator, or a variable we obtained a value for from matching a process with the left hand side of a process definition.

We give here the equations for the function *next* that decides what the next station is.

```
[3] next(1, p) = 2
[4] next(2, p) = 3
[5] next(3, A) = 4
[6] next(3, B) = 5
[7] next(4, p) = 5
[8] next(5, p) = 6
```

We see that a rewriting of the function *next* with only a value given for the station, gives us the new station, except for station 3 since it depends on the product. We can alter the last part of the process definition like this.

```
+  [and(not(eq-stat(s, 1)), not(eq-stat(s, 6))) = true] → (
      sum(p in PRODUCT,
         from-belt(s, p) . (
            [p = A] → to-belt(s, next(s, A), p)
         +  [p = B] → to-belt(s, next(s, B), p)
         )
      ) . Station(s)
   )
```

This gives us the picture in Figure 6-3.

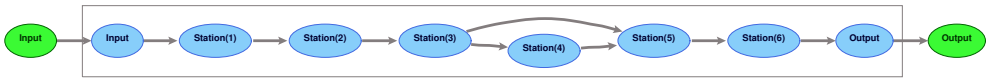


Figure 6-3. Factory

We used here an option that gives an orientation from left to right, instead of the default top to bottom. Note that there are two nodes named 'Input' and two nodes named 'Ouput'. The nodes inside the box represents the processes Input and Output from the specification, and the others are added to connect the input-processes and output-processes as described on page 49.

6.3.2 Combination of Processes

Consider now a generalized form of the factory in which all stations are connected with each other by conveyer belts. We use a scheduler to control this factory in such a way that it

acts the same as the factory in the previous factory. The specification can be found in Appendix A.3.

Lets take a look at the specification of the scheduler.

```
Scheduler =
      sum(p in PRODUCT,
         rec-start(p) .
         (
            SubScheduler(1, p)
         ||  Scheduler
         )
      )
SubScheduler(s, p) =
         [not(eq-stat(s, 6)) = true] → (
            rec-request(s, p) .
            Next(s, p, next(s, p))
         )
      +  [s = 6] →
            rec-end
Next(s, p, n) =
      send-next(s, n) .
      SubScheduler(n, p)
```

We see here that for each product a subscheduler is created. If we generate an animation for this specification it gives us the picture in Figure 6-4.

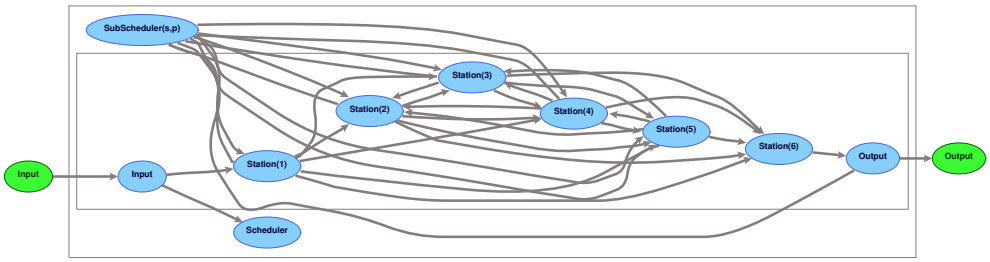


Figure 6-4. Scheduled factory

The processes *Scheduler* and *SubScheduler* in this picture do not reflect the specification. We should create and destroy *SubScheduler* processes dynamically, but that is not possible (at the moment). But since there is no communication possible between the *Scheduler* and *SubScheduler*, or between two *SubScheduler*s, we can consider them as one process. This results in the picture shown in Figure 6-5. Whether this behaviour is always wanted, we do not know, so we made this combination of processes optional.

6.3.3 Heuristics

Sometimes the direction of a communication cannot be decided upon, for instance when both communication partners are not sum-ports. However, from the names of the communication partners one could speculate the direction. If one of the partners has the name *send* and the other the name *receive* then one can conclude on the direction of the communication, taken the names are not poorly chosen. We have implemented an optional decision on the direction using heuristics when it is not possible otherwise. The possible

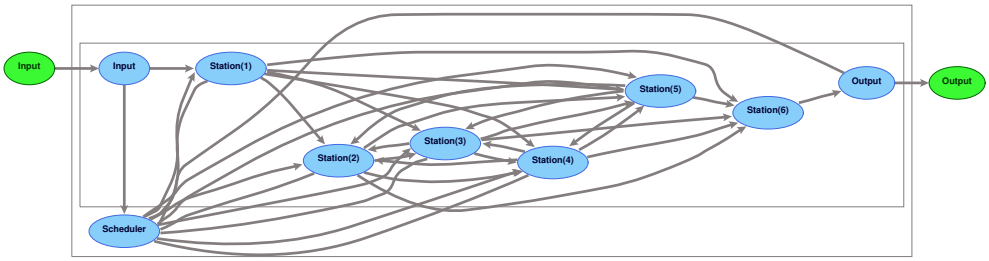


Figure 6-5. Scheduled factory with combined processes

combinations of names of the communication partners are given in Table 6-1. The '*' represent at least one character.

Table 6-1. Communication heuristics

send	receive
snd	rec
snd-*	rec-*
*-snd	*-rec
-snd-	*-rec-*
snd	rec-*
snd	*-rec
snd-*	rec
*-snd	rec
-snd-	*-rec
*-snd	*-rec-*

It is most likely that other combinations are desired for a particular specification, but the ones listed here are sufficient for our specifications. It is our intention to implement an option with which a user can specify a particular combination of names that should act as sending and receiving actions.

6.4 Remarks

Although it seems that we can generate animations for all specifications with only a few adjustments, we should keep in mind that expanding processes is done through open term matching with the left hand side of process definition. This can result in a mismatch since the process to expand can have an argument that should be rewritten in order to match but contains a variable which prevents a rewrite.
Also, deciding if an atom is an element of a hide or an encapsulation set is open term matching and thus can result in a mismatch for the same reason.
So we must try to circumvent these situations. We can use conditional expressions for this, but they make the specifications larger.

The direction of a communication is now based on the presence of a sum-construction at

the sides of the communication. In some cases, we could try to do a better job by examining the context of both sides of the communication.
We should also note that a sum-construction is not always meant to be a port. It could for instance also be used to connect to a random process.

Despite the above, generating an animation is very useful in testing and understanding specifications. One of its main advantages is that a generated animation reflects the specification, in contrast with other techniques such as visualization through transition systems, so that events can easily be traced back to their origin in the specification.

Chapter 7

Related Work

There are many examples of animation of process algebra specifications that are built in an ad hoc manner. These examples usually concern a particular specification and mostly are visualizations of execution traces. Moreover, they usually do not have any interaction through the visualization like our animations have. Animation of process algebra specifications that use some kind of model can roughly be divided in *state-based* and *data-flow-based* approaches. In this chapter we list some of the approaches and compare these with our approach.

7.1 State-Based

Often specifications are converted to another formalism for visualization or animation. For instance, LOTOS [29] and μCRL [26] connect to CADP (Caesar/Aldebaran Development Package) [23], which is based on finite-state machines. An example is shown in Figure 7-1.

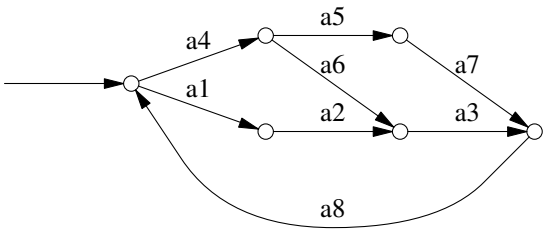


Figure 7-1. Example of a state machine

State machines can easily be animated. For large specifications, the generation of state machines can take a lot of time and space, and can lead to an explosion of the number of states far above the limits of the computer in use. A similar transformation can be applied to PSF specifications and animation of the state machines can be based on the animation library presented in Chapter 5. The disadvantage of this is that in most cases a particular

state in a state machine cannot be coupled back to a process or processes in the specification, which makes it difficult to use in testing and education.

7.2 Data-Flow-Based

In many animations of communication protocols Message Sequence Charts (MSCs) [30] are used for visualization of the behaviour of these protocols. In [59] LOTOS execution traces are transformed into MSCs and in [66] interworkings, a technique similar to MSCs, are generated from PSF process traces. In a MSC a system component is represented by a vertical line called an instance with time running from top to bottom along this instance. Exchange of messages is described by arrows between the instances. An example is given in Figure 7-2.

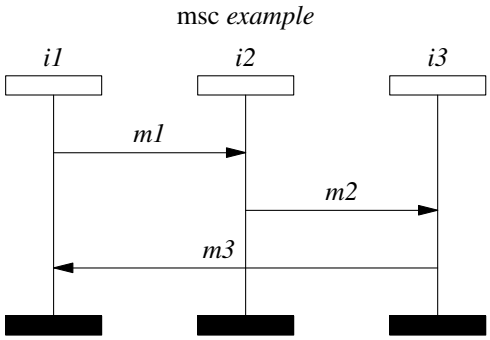


Figure 7-2. Example of a Message Sequence Chart

One advantage of our animations is that they give a two dimensional view of processes and their connections with other processes, instead of a one dimensional view of the processes with a time scale. This gives a better view with a larger number of processes. An advantage of MSCs is that they show a history of actions.

7.3 Framework

In some cases the animations are still built in an ad hoc manner, but now a framework is used. A framework is meant to create animations in an easy and consistent manner. An example of such a framework is Jasper [61], where the behaviour of a protocol must be defined by a set of Java classes and the view of the protocol is given in the form of a Time Sequence Diagram. Another example is ANGOR [14], an open animation environment for linking execution actions to graphical elements in a Java framework. A number of graphical elements is predefined, and others can be added.

Our work can be considered a framework giving standard features, but it can also be extended with other features for a particular animation. It also allows for easy generation of consistent animations.

7.4 Animation Generation

Most animations in the related work mentioned in this chapter need to be adapted by hand or take a lot of time to be generated on a change in the specification. We can generate the animations from the specifications in a fast and consistent manner, making it ideal in development and debugging of specifications.

Part III

Software Engineering with PSF

Chapter 8

Re-engineering the PSF Compiler

In software engineering and re-engineering it is common practice to decompose systems into components that communicate with each other. An advantage of this decomposition is that maintenance can be done on smaller components that are easier to comprehend. Another advantage is that components can be reused in other applications or can be replaced with different implementations offering the same functionality. To allow a number of components to communicate with each other a so-called coordination architecture is required. In connection with PSF we will make use of the ToolBus described in Chapter 4. The ToolBus utilizes a scripting language to describe processes inside the ToolBus that can communicate with each other and with various tools existing outside the ToolBus. For larger systems, such scripts can become rather complex and for that reason quite difficult to test and debug. Specification of a script in PSF enables one to apply the analysis tools available for PSF to the specification of the script. Moreover, if one or more tools have been specified in PSF the script may also be analysed in combination with PSF specifications of components of the whole system.

The purpose of this chapter is to get experience with decomposing a software system using process algebra. We do this by making a specification of the behaviour of the implementation of the system and transforming the specification into a specification of this system as a ToolBus application that functions as the design of the decomposed system. From this design we build a new implementation of the system as a ToolBus application. Furthermore, we obtain a specification of the architecture of the system by abstracting from implementation details and applying algebraic laws. This re-engineering process is depicted in Figure 8-1. The re-engineering process shows the way to an engineering process which starts with a specification of the architecture and through development of a ToolBus application specification leads to an implementation.

As a case study, we re-engineer the PSF compiler. At the start of the re-engineering process this compiler consists of several components run by a driver, which makes it a suitable candidate for ToolBus based coordination. We describe the specification of the compiler in section 8.1. In section 8.2 we develop a PSF library of ToolBus internals. We give an

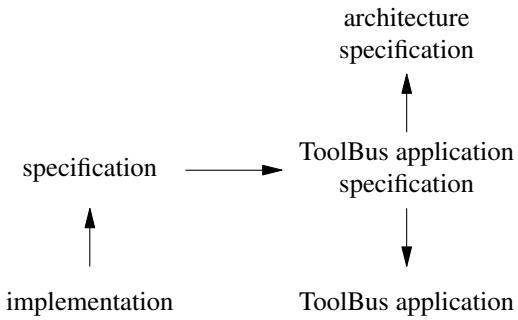


Figure 8-1. Re-engineering process

example specification to show how to use this library in section 8.3, and turn this specification into a ToolBus application. In section 8.4 we provide a specification of the compiler as a ToolBus application, using the PSF ToolBus library, from which we implement the compiler as a real ToolBus application. In section 8.5 a specification of the architecture for this (re-engineered) compiler is extracted from its specification. In section 8.6 we build a parallel version of the compiler based on the architecture of the re-engineered compiler while reusing specifications and implementations for components of the compiler as a ToolBus application.

8.1 Specification of the Compiler

As starting point for the re-engineering of the PSF compiler we use a specification of the behaviour of the compiler. A description of the PSF compiler can be found in section 3.1. We repeat here the phases of the compiler driver.

1. **collecting modules**
   The modules are collected from the files given to the compiler, and missing imported modules are searched for in the libraries.
2. **sorting modules**
   The modules are sorted according to their import relation.
3. **splitting files**
   Files scanned in phase 1 that contain more than one module are split into files containing one module each.
4. **parsing** (from PSF to MTIL)
   All modules that are out of date, that is the destination file does not exist, or the source file (with extension .psf) is newer than the destination file (with extension .mtil), are parsed.

5. **normalizing** (from MTIL to ITIL)
   All modules that are out of date, that is the destination file does not exist, or the source file (with extension .mtil) is newer than the destination file (with extension .itil) or one of its imported modules (ITIL) is newer, are normalized.
6. **flattening** (from ITIL to TIL)
   The main module is translated from ITIL to TIL.
7. **converting sorts to sets**
   The simulator preprocessor is invoked for converting sorts to sets so that the simulator can deal with them.
8. **checking TRS**
   The term rewrite system checker is invoked.

We will not display the specification of the compiler here, but we show the generated animation of the compiler in Figure 8-2. The animation clearly shows the different phases of the compiler that communicate with the compiler driver. The processes PsfMtil, MtilItil, ItilTil, SimPP, and TrsCheck are implemented as calls to separate programs. These are used as components and an abstraction is made from their internal workings in the context of this specification.

To give some insight in the complexity of the specification, the import structure of the modules of the specification is shown in Figure 8-3. The module Booleans originates from a standard data library of PSF.

8.2 PSF ToolBus Library

This section presents a specification of a library of interfaces for PSF which can be used as a basis for the specification of ToolBus applications. This specification does not cover all the facilities of the ToolBus, but just what is necessary for the project at hand.

8.2.1 Data

First, a sort is defined for the data terms (`Tterm`) used in the tools. An abstraction is made from the actual data used by the tools.

```
data module ToolTypes
begin
    exports
    begin
        sorts
            Tterm
    end
end ToolTypes
```

Next, the sorts are introduced for the data terms (`TBterm`) and identifiers (`TBid`) which will be used inside the ToolBus as well as for communication with the ToolBus.

```
data module ToolBusTypes
begin
    exports
    begin
```

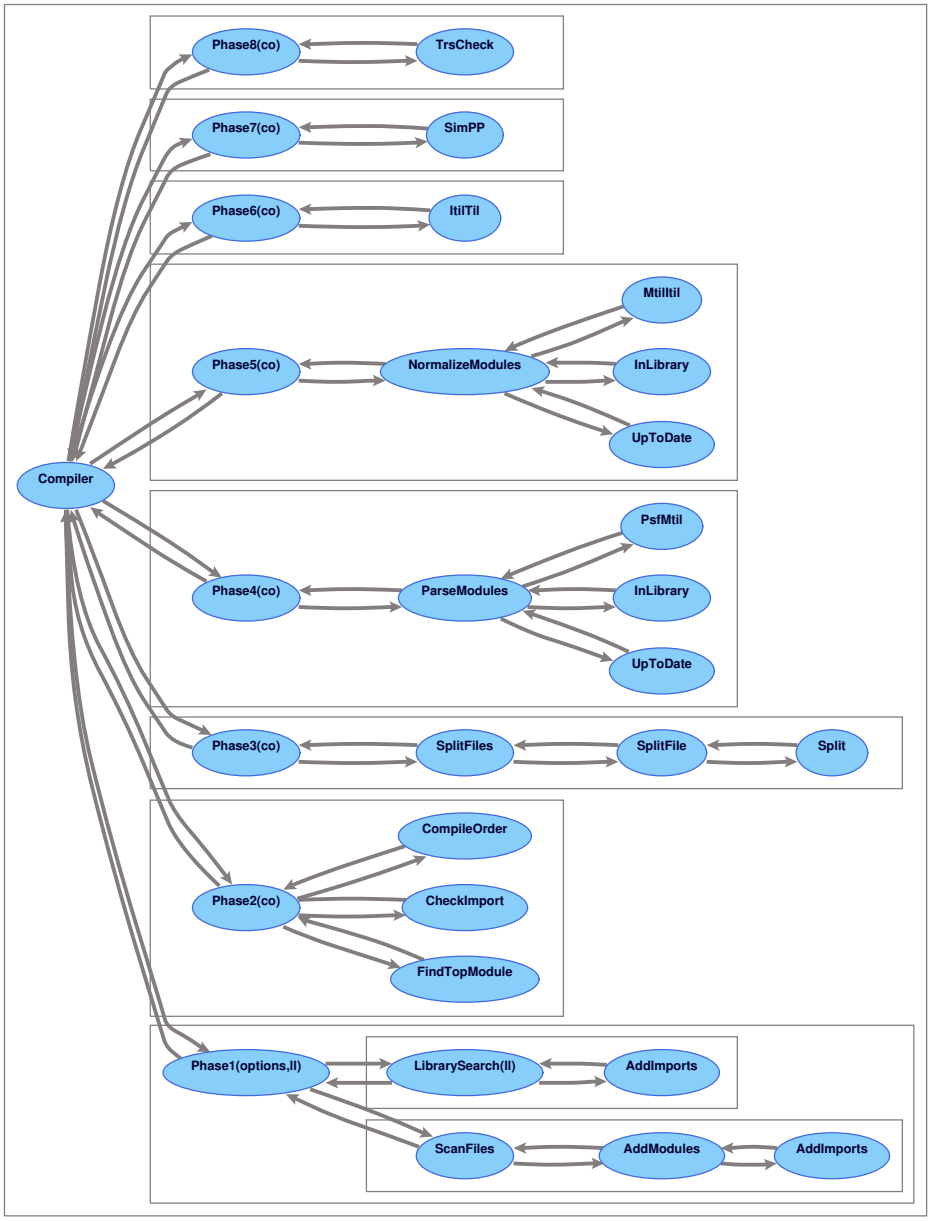


Figure 8-2. Generated animation of the compiler

```
        sorts
            TBterm,
            TBid
    end
end ToolBusTypes
```

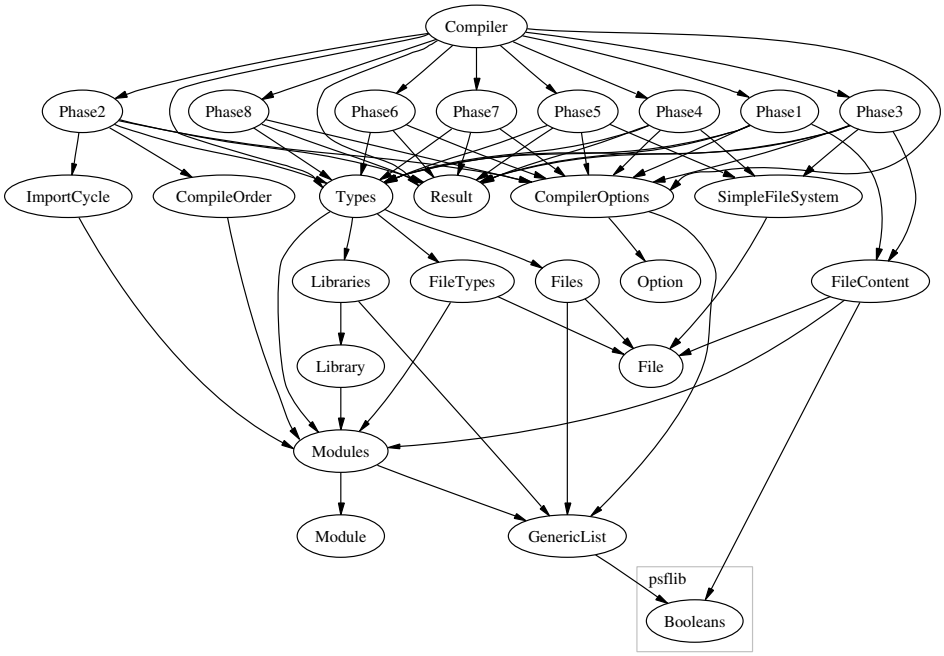


Figure 8-3. Import graph of the specification of the compiler

The module ToolFunctions provides names for conversions between data terms used outside and inside the ToolBus.

```
data module ToolFunctions
begin
    exports
    begin
        functions
            tbterm : Tterm → TBterm
            tterm : TBterm → Tterm
    end
    imports
        ToolTypes,
        ToolBusTypes
    variables
        t : → Tterm
    equations
    [’] tterm(tbterm(t)) = t
end ToolFunctions
```

The ToolBus has access to several functions operating on different types. Here only the operators for tests on equality and inequality of terms, will be needed. These are introduced in the module ToolBusFunctions.

```
data module ToolBusFunctions
begin
    exports
    begin
```

```
        functions
            equal : TBterm # TBterm → BOOLEAN
    end
    imports
        ToolBusTypes,
        Booleans
    variables
        tb1 : → TBterm
        tb2 : → TBterm
    equations
    [’] equal(tb1, tb1) = true
    [’] not(equal(tb1, tb2)) = true
end ToolBusFunctions
```

Here, we applied a little trick for specifying inequality, based on the rewriting strategy (innermost) in use. When the terms `a` and `b` are equal, `equal(a,b)` gives `true` and `not(equal(a,b))` thus gives `false`. When the terms are not equal, there is no matching equation for `equal(a,b)`, and therefore `not(equal(a,b))` gives `true`.

8.2.2 Connecting Tools to the ToolBus

We repeat Figure 8-4 from Chapter 4 showing two possible ways of connecting tools to the ToolBus. One way is to use a separate adapter and the other is to have a built-in adapter.

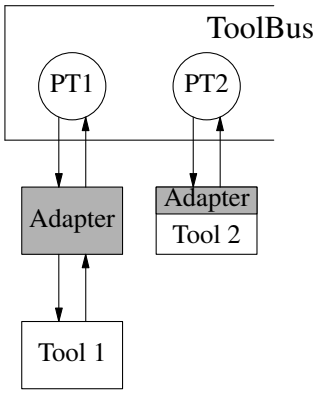


Figure 8-4. Model of tool and ToolBus interconnection

We define the primitives for communication between a tool and its adapter.

```
process module ToolAdapterPrimitives
begin
    exports
    begin
        atoms
            tooladapter-rec : Tterm
            tooladapter-snd : Tterm
    end
    imports
        ToolTypes
```

```
end ToolAdapterPrimitives
```

The primitives for communication between a tool and the ToolBus are fixed by the ToolBus design. At this stage these need to be formally defined in PSF, however. These primitives can be used for communication between an adapter and the ToolBus as well, since the adapter logically takes the place of the tool it is supposed to connect to the ToolBus.

```
process module ToolToolBusPrimitives
begin
    exports
    begin
        atoms
            tooltb-snd : TBterm
            tooltb-rec : TBterm
            tooltb-snd-event : TBterm
            tooltb-rec-ack-event : TBterm
    end
    imports
        ToolBusTypes
end ToolToolBusPrimitives
```

Inside a ToolBus script a number of primitives can be used. These primitives consist of the actions for communication between ToolBus processes and their synchronous communication action, the actions used to communicate with the tools, and the action required to shutdown the ToolBus.

```
process module ToolBusPrimitives
begin
    exports
    begin
        atoms
            tb-snd-msg : TBterm # TBterm
            tb-rec-msg : TBterm # TBterm
            tb-comm-msg : TBterm # TBterm
            tb-snd-msg : TBterm # TBterm # TBterm
            tb-rec-msg : TBterm # TBterm # TBterm
            tb-comm-msg : TBterm # TBterm # TBterm
            tb-snd-eval : TBid # TBterm
            tb-rec-value : TBid # TBterm
            tb-snd-do : TBid # TBterm
            tb-rec-event : TBid # TBterm
            tb-snd-ack-event : TBid # TBterm
            tb-shutdown
    end
    imports
        ToolBusTypes
    communications
        tb-snd-msg(tb1, tb2) | tb-rec-msg(tb1, tb2) =
            tb-comm-msg(tb1, tb2) for tb1 in TBterm, tb2 in TBterm
        tb-snd-msg(tb1, tb2, tb3) | tb-rec-msg(tb1, tb2, tb3) =
            tb-comm-msg(tb1, tb2, tb3)
            for tb1 in TBterm, tb2 in TBterm, tb3 in TBterm
end ToolBusPrimitives
```

In Table 8-1 we give an overview of the corresponding ToolBus primitives and the primitives used in the PSF ToolBus library with communications for the primitives of the latter. Arguments of the primitives are left out for clearness. The `<function>` is not really a primitive, but the invocation of the function found in the second argument from either a `snd-eval` or a `snd-do` that is received by the adapter of a tool. The receive

action of the adapter is the real primitive with which `tooltb-rec` corresponds.

Table 8-1. ToolBus and PSF ToolBus Library primitives

ToolBus	PSF ToolBus Library	
snd-msg rec-msg	tb-snd-msg tb-rec-msg	tb-comm-msg
snd-eval <function> snd-do <function> snd-value rec-value	tb-snd-eval tooltb-rec tb-snd-do tooltb-rec tooltb-snd tb-rec-value	tooltb-rec-eval tooltb-rec-do tooltb-snd-value
snd-event rec-event snd-ack-event <rec-ack-event>	tooltb-snd-event tb-rec-event tb-snd-ack-event tooltb-rec-ack-event	tooltb-snd-event tooltb-rec-ack-event

The ToolBus provides primitives allowing an arbitrary number of terms as parameters for communication between processes in the ToolBus. Here, the specification only covers the case of two and three term arguments for the primitives, because versions with more arguments are usually not needed. In order to do better, lists of terms have to be introduced that can be used in place of a single term. Specification of such lists is certainly possible in PSF but an unnecessary complication at this stage. The two-term version can be used with the first term as a 'to' or 'from' identifier and the second as a data argument. The three-term version can be used with the first term as 'from', the second as 'to', and the third as the actual data argument. If more arguments have to be passed, they can always be grouped into a single argument.

The module NewTool is a generic module with parameter Tool for connecting a tool to the ToolBus.

```
process module NewTool
begin
  parameters
    Tool
    begin
      processes
        Tool
    end Tool
  exports
  begin
    atoms
      tooltb-snd-value : TBid # TBterm
      tooltb-rec-eval : TBid # TBterm
      tooltb-rec-do : TBid # TBterm
      tooltb-snd-event : TBid # TBterm
      tooltb-rec-ack-event : TBid # TBterm
    processes
      TBProcess
    sets
      of atoms
```

```
            TBProcess = {
               tb-rec-value(tid, tb), tooltb-snd(tb),
               tb-snd-eval(tid, tb), tb-snd-do(tid, tb),
               tooltb-rec(tb), tb-rec-event(tid, tb),
               tooltb-snd-event(tb), tb-snd-ack-event(tid, tb),
               tooltb-rec-ack-event(tb)
               | tid in TBid, tb in TBterm
            }
   end
   imports
      ToolToolBusPrimitives,
      ToolBusPrimitives
   communications
      tooltb-snd(tb) | tb-rec-value(tid, tb) =
         tooltb-snd-value(tid, tb) for t in TBterm, tid in TBid
      tooltb-rec(tb) | tb-snd-eval(tid, tb) =
         tooltb-rec-eval(tid, tb) for t in TBterm, tid in TBid
      tooltb-rec(tb) | tb-snd-do(tid, tb) =
         tooltb-rec-do(tid, tb) for t in TBterm, tid in TBid
      tooltb-snd-event(tb) | tb-rec-event(tid, tb) =
         tooltb-snd-event(tid, tb) for t in TBterm, tid in TBid
      tooltb-rec-ack-event(tb) | tb-snd-ack-event(tid, tb) =
         tooltb-rec-ack-event(tid, tb) for tb in TBterm, tid in TBid
   definitions
      TBProcess = encaps(TBProcess, Tool)
end NewTool
```

The process Tool accomplishes the connection between a process inside the ToolBus and a tool outside the ToolBus. The process TBProcess encapsulates the process Tool in order to enforce communications and thereby to prevent communications with other tools or processes. Note that TBProcess is used as the name of the main process and as the name of the encapsulation set. By doing so, they can both be renamed with a single renaming. This renaming is necessary if more than one tool is connected to the ToolBus (which is of course the essence of the ToolBus).

The module NewToolAdapter is a generic module with parameters Tool and Adapter for connecting a tool and its adapter.

```
process module NewToolAdapter
begin
   parameters
      Tool
      begin
         atoms
            tool-snd : Tterm
            tool-rec : Tterm
         processes
            Tool
      end Tool,
      Adapter
      begin
         processes
            Adapter
      end Adapter
   exports
   begin
      atoms
         tooladapter-comm : Tterm
         adaptertool-comm : Tterm
```

```
        processes
            ToolAdapter
        sets
            of atoms
                ToolAdapter = {
                    tool-snd(t), tooladapter-rec(t),
                    tool-rec(t), tooladapter-snd(t)
                    | t in Tterm
                }
    end
    imports
        ToolAdapterPrimitives,
        ToolBusTypes
    communications
        tool-snd(t) | tooladapter-rec(t) = tooladapter-comm(t)
            for t in Tterm
        tool-rec(t) | tooladapter-snd(t) = adaptertool-comm(t)
            for t in Tterm
    definitions
        ToolAdapter = encaps(ToolAdapter, Adapter || Tool)
end NewToolAdapter
```

The process ToolAdapter puts an Adapter and a Tool in parallel and enforces communication between them by an encapsulation. In this case the main process and the encapsulation set have the same name, so that only one renaming is needed.

8.2.3 ToolBus Instantiation

The module NewToolBus is a generic module with parameter Application for instantiation of the ToolBus with an application.

```
process module NewToolBus
begin
    parameters
        Application
        begin
            processes
                Application
        end Application
    exports
    begin
        processes
            ToolBus
    end
    imports
        ToolBusPrimitives
    atoms
        application-shutdown
        tbc-shutdown
        tbc-app-shutdown
        TB-shutdown
        TB-app-shutdown
    processes
        ToolBus-Control
        Shutdown
    sets
        of atoms
            H = {
                tb-snd-msg(tb1, tb2), tb-rec-msg(tb1, tb2),
```

```
            tb-snd-msg(tb1, tb2, tb3), tb-rec-msg(tb1, tb2, tb3)
            | tb1 in TBterm, tb2 in TBterm, tb3 in TBterm
         }
         TB-H = {
            tb-shutdown, tbc-shutdown,
            tbc-app-shutdown, application-shutdown
         }
         P = { TB-shutdown, TB-app-shutdown }
   communications
      tb-shutdown | tbc-shutdown = TB-shutdown
      tbc-app-shutdown | application-shutdown = TB-app-shutdown
   definitions
      ToolBus =
         encaps(TB-H,
            prio(P > atoms,
               ToolBus-Control
            || disrupt(
                  encaps(H, Application),
                  Shutdown
               )
            )
         )
      ToolBus-Control = tbc-shutdown . tbc-app-shutdown
      Shutdown = application-shutdown
end NewToolBus
```

A toolbus application can be described basically with `ToolBus = encaps(H, Application).` The remaining code is needed to force a shutdown of all processes that otherwise would be left either running or in a state of deadlock after a ToolBus shutdown by the application. When an application needs to shutdown it performs an action `tb-shutdown` which will communicate with the action `tbc-shutdown` of the `ToolBus-Control` process, which then performs a `tbc-app-shutdown` that will communicate with `application-shutdown` of the `Shutdown` process enforcing a disrupt of the `Application` process.

In Figure 8-5 an overview is given of the import relations of the modules in the PSF ToolBus library. The module Booleans stems from a standard data library of PSF.

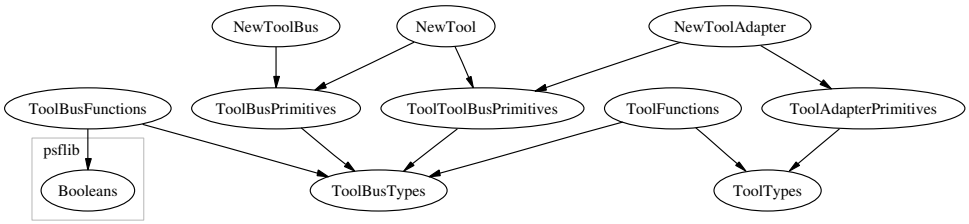


Figure 8-5. Import graph of the ToolBus library

8.3 Example

As an example of the use of the PSF ToolBus library, we specify an application carried out in the form as shown in Figure 8-4, which is also used as example in describing the

ToolBus. In this example, Tool1 can either send a 'message' to Tool2 and then wait for an acknowledgement from Tool2, or it can send a 'quit' after which the application will shutdown.

8.3.1 Specification of the Tools

The first module defines the data that will be used.

```
data module Data
begin
   exports
   begin
      functions
         message : → Tterm
         ack : → Tterm
         quit : → Tterm
   end
   imports
      ToolTypes
end Data
```

A specification of Tool1 and its adapter is then obtained.

```
process module Tool1
begin
   exports
   begin
      atoms
         snd : Tterm
         rec : Tterm
      processes
         Tool1
   end
   imports
      Data
   definitions
      Tool1 =
            snd(message) .
            rec(ack) .
            Tool1
         + snd(quit)
end Tool1
```

```
process module AdapterTool1
begin
   exports
   begin
      processes
         AdapterTool1
   end
   imports
      Data,
      ToolFunctions,
      ToolAdapterPrimitives,
      ToolToolBusPrimitives
   definitions
      AdapterTool1 =
            tooladapter-rec(message) .
            tooltb-snd-event(tbterm(message)) .
            tooltb-rec-ack-event(tbterm(message)) .
```

```
            tooladapter-snd(ack) .
            AdapterTool1
         + tooladapter-rec(quit) .
            tooltb-snd-event(tbterm(quit))
end AdapterTool1
```

Tool1 and its adapter are combined by importing NewToolAdapter and binding the parameters.

```
process module Tool1Adapter
begin
   imports
      NewToolAdapter {
         Tool bound by [
            tool-snd → snd,
            tool-rec → rec,
            Tool → Tool1
         ] to Tool1
         Adapter bound by [
            Adapter → AdapterTool1
         ] to AdapterTool1
         renamed by [
            ToolAdapter → Tool1Adapter
         ]
      }
end Tool1Adapter
```

We specify Tool2.

```
process module Tool2
begin
   exports
   begin
      processes
         Tool2
   end
   imports
      Data,
      ToolFunctions,
      ToolToolBusPrimitives
   definitions
      Tool2 =
         tooltb-rec(tbterm(message)) .
         tooltb-snd(tbterm(ack)) .
         Tool2
end Tool2
```

8.3.2 Specification of the ToolBus Processes

Some identifiers are defined in order to distinguish the messages sent between ToolBus processes themselves and between ToolBus processes and their accompanying tools. The lowercase identifiers (of type TBterm) are used with the actions `tb-snd-msg` and `tb-rec-msg`. The first argument of a message will always be the origin of the message, and the second argument will serve as its destination. Uppercase identifiers (of type TBid) are used as tool identifiers. Strictly speaking these are not necessary, since there can't be any communication with any other tool because of encapsulation. By using them, however, the actions for communication with a tool will have more similarity to the ones used in the ToolBus.

```
data module ID
begin
   exports
   begin
      functions
         T1 : → TBid
         t1 : → TBterm
         T2 : → TBid
         t2 : → TBterm
   end
   imports
      ToolBusTypes
end ID
```

For both tools a ToolBus process is defined. The specifications for these processes describe the protocol for communication between the tools.

```
process module PTool1
begin
   exports
   begin
      processes
         PTool1
   end
   imports
      Tool1Adapter,
      ID,
      ToolBusPrimitives,
      ToolBusFunctions
   processes
      PT1
   definitions
      PTool1 = Tool1Adapter ‖ PT1
      PT1 =
            tb-rec-event(T1, tbterm(message)) .
            tb-snd-msg(t1, t2, tbterm(message)) .
            tb-rec-msg(t2, t1, tbterm(ack)) .
            tb-snd-ack-event(T1, tbterm(message)) .
            PT1
         +  tb-rec-event(T1, tbterm(quit)) .
            snd-tb-shutdown
end PTool1
```

```
process module PTool2
begin
   exports
   begin
      processes
         PTool2
   end
   imports
      Tool2,
      ID,
      ToolBusPrimitives
   processes
      PT2
   definitions
      PTool2 = Tool2 ‖ PT2
      PT2 =
         tb-rec-msg(t1, t2, tbterm(message)) .
         tb-snd-eval(T2, tbterm(message)) .
```

```
            tb-rec-value(T2, tbterm(ack)) .
            tb-snd-msg(t2, t1, tbterm(ack)) .
            PT2
end PTool2
```

8.3.3 Specification of the ToolBus Application

The ToolBus processes are connected with the tools and together they constitute the process System that merges the resulting two processes.

```
process module Tools
begin
   exports
   begin
      processes
         System
   end
   imports
      NewTool {
         Tool bound by [
            Tool → PTool1
         ] to PTool1
         renamed by [
            TBProcess → XPTool1
         ]
      },
      NewTool {
         Tool bound by [
            Tool → PTool2
         ] to PTool2
         renamed by [
            TBProcess → XPTool2
         ]
      },
      ID,
      ToolBusFunctions
   definitions
      System = XPTool1 || XPTool2
end Tools
```

At this stage renamings are necessary to be able to distinguish the two processes TBProcess (and sets).

The process System is now transformed into a ToolBus application.

```
process module App
begin
   imports
      NewToolBus {
         Application bound by [
            Application → System
         ] to Tools
      }
end App
```

The main process of this application is ToolBus. A generated animation is shown in Figure 8-6, in which AdapterTool1 just sent a message it had received from Tool1, to ToolBus process PT1.

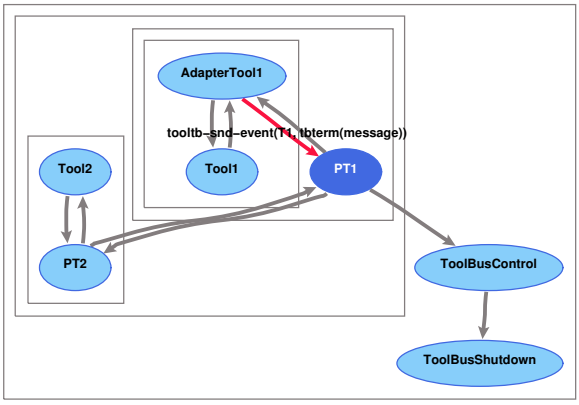


Figure 8-6. Animation of the ToolBus specification example

8.3.4 Example as ToolBus Application

The application we have specified above has been implemented as an application consisting of three Tcl/Tk programs (see the ToolBus example in section 4.1). We repeat the ToolBus script below. The processes `PT1` and `PT2` closely resemble the processes `PTool1` and `PTool2` in our PSF specification. The `execute` actions in the ToolBus script correspond to starting the adapter for Tool1 and starting Tool2 in parallel with the processes `PT1` and `PT2` respectively.

```
process PT1 is
let
   T1: tool1adapter
in
   execute(tool1adapter, T1?) .
   (
      rec-event(T1, message) .
      snd-msg(t1, t2, message) .
      rec-msg(t2, t1, ack) .
      snd-ack-event(T1, message)
   +  rec-event(T1, quit) .
      shutdown("")
   ) * delta
endlet

process PT2 is
let
   T2: tool2
in
   execute(tool2, T2?) .
   (
      rec-msg(t1, t2, message) .
      snd-eval(T2, eval(message)) .
      rec-value(T2, value(ack)) .
      snd-msg(t2, t1, ack)
   ) * delta
endlet
```

```
tool tool1adapter is {
   command = "wish-adapter -script tool1adapter.tcl" }
tool tool2 is { command = "wish-adapter -script tool2.tcl" }

toolbus(PT1, PT2)
```

The actions `snd-eval` and `rec-value` differentiate from their equivalents in the PSF specification. The term `eval(message)` instead of just `message` is needed because the interpreter of evaluation requests that a tool receives from the ToolBus, calls a function with the name it finds as function in this term. We could have used any name instead of `eval` provided that Tool2 has got a function with that name.

The same scheme is needed by the ToolBus for `rec-value`, but for a different reason. Users have problems understanding that in something like `rec-value(T, ack) + rec-value(T, X?)`, although the value returned is `ack`, the ToolBus can choose for the `rec-value(T, X?)`. To prevent such errors by users it is required to give at least the name of a function in the argument.

The processes in the ToolBus script use iteration and the processes in the PSF specification recursion. In PSF it is also possible to use iteration in this case, since the processes have no arguments to hold the current state. On the other hand, in PSF it is not possible to define variables for storing a global state, so when it is necessary to hold the current state, this must be done through the arguments of a process and be formalized via recursion.

The last line of the ToolBus script starts the processes `PT1` and `PT2` in parallel. Its equivalent in the PSF specification is the process `System`.

8.4 The Compiler as ToolBus Application

Instead of calling the parser (process PsfMtil) and normalizer (process MtilItil) directly, they should be called via the ToolBus. This can be accomplished by specifying an adapter for the compiler, and a ToolBus script consisting of the ToolBus processes for the compiler, parser and normalizer. The resulting animation is shown in Figure 8-7.

The specification of the ToolBus processes is shown below. The specification for the tools and adapter used in here is similar to the specifications for the tools and adapter used in the example on page 79. The terms `tool-mtil` and `tool-itil` originate from the specification of the compiler and have to be converted to a `tterm` in order to be used in an adapter, and the resulting terms have to be converted to a `tbterm` in order to be used in a ToolBus process.

```
process module PPSF
begin
   exports
   begin
      processes
         PPSF
   end
   imports
      PSFAdapter,
      ToolFunctions,
      ToolBusPrimitives,
      ToolBusFunctions,
```

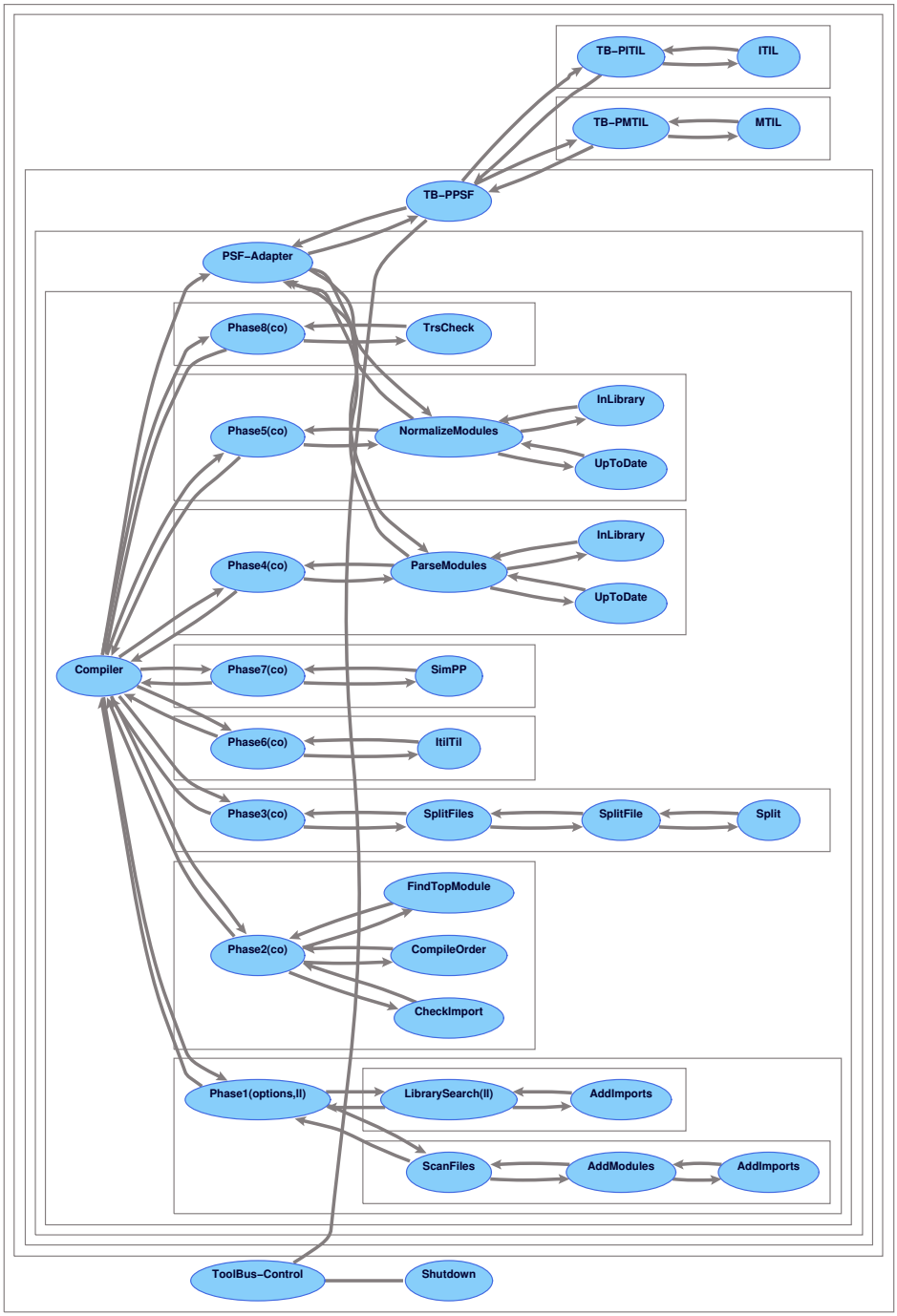


Figure 8-7. Generated animation of the compiler as ToolBus application

```
      ToolBus-ID
   definitions
      PPSF =
            PSFAdapter
         ||
            (
               (
                  sum(args in TBterm,
                     tb-rec-event(PSF, tbterm(tterm(tool-mtil)),
                        args) .
                     tb-snd-ack-event(PSF,
                        tbterm(tterm(tool-mtil))) .
                     tb-snd-msg(psf, mtil, args)
                  ) .
                  sum(result in TBterm,
                     tb-rec-msg(mtil, psf, result) .
                     tb-snd-do(PSF, result)
                  )
               +  sum(args in TBterm,
                     tb-rec-event(PSF, tbterm(tterm(tool-itil)),
                        args) .
                     tb-snd-ack-event(PSF,
                        tbterm(tterm(tool-itil))) .
                     tb-snd-msg(psf, itil, args)
                  ) .
                  sum(result in TBterm,
                     tb-rec-msg(itil, psf, result) .
                     tb-snd-do(PSF, result)
                  )
               +  tb-rec-event(PSF, tbterm(quit)) .
                  tb-shutdown
               ) * delta
            )
end PPSF

process module PMTIL
begin
   exports
   begin
      processes
         PMTIL
   end
   imports
      MTIL,
      ToolBusPrimitives,
      ToolBus-ID
   definitions
      PMTIL =
            MTIL
         ||
            (
               sum(args in TBterm,
                  tb-rec-msg(psf, mtil, args) .
                  tb-snd-eval(MTIL, args) .
                  sum(result in TBterm,
                     tb-rec-value(MTIL, result) .
                     tb-snd-msg(mtil, psf, result)
                  )
               ) * delta
            )
end PMTIL
```

```
process module PITIL
begin
    exports
    begin
        processes
            PITIL
    end
    imports
        ITIL,
        ToolBusPrimitives,
        ToolBus-ID
    definitions
        PITIL =
              ITIL
            ||
              (
                 sum(args in TBterm,
                    tb-rec-msg(psf, itil, args) .
                    tb-snd-eval(ITIL, args) .
                    sum(result in TBterm,
                       tb-rec-value(ITIL, result) .
                       tb-snd-msg(itil, psf, result)
                    )
                 ) * delta
              )
end PITIL
```

The import graph of the specification of the compiler as ToolBus application is shown in Figure 8-8.

8.4.1 Implementation of the Compiler as ToolBus Application

The original implementation of the compiler has been provided with an interface that communicates with the adapter. The adapter is written in Perl [67] as an extension of the Perl-adapter provided with the ToolBus. The ToolBus script is derived from the specification of the ToolBus processes. The parser and normalizer are wrapped with Perl scripts that take care of fetching the exit status of the two tools and sending this information back as a result. The actual application is a Perl script that provides an environment with all the right settings and invokes the ToolBus, according to the arguments given on the command line.

Although it is not of primary interest at this stage, a comparison of the performance for the compiler that uses the ToolBus (tbpsf) and the original compiler (psf) is given. The tests consist of a complete compilation of the specification of the compiler as a ToolBus application (consisting of 49 modules) and an update in which only several modules have to be (re)compiled. The tests have been performed on two different machines, one with only one cpu (M1), and one with four cpu's (M4). The timings[3] shown in Table 8-2 are averages over several runs. It clearly shows that the use of the ToolBus imposes a lot of overhead, largely due to context switching. Because of the four cpu's, the configuration M4 needs fewer context switches, resulting in less overhead.

3. The configurations were running in normal operation mode, which means that timings are influenced by other processes and load on the file server, and for that reason are very rough.

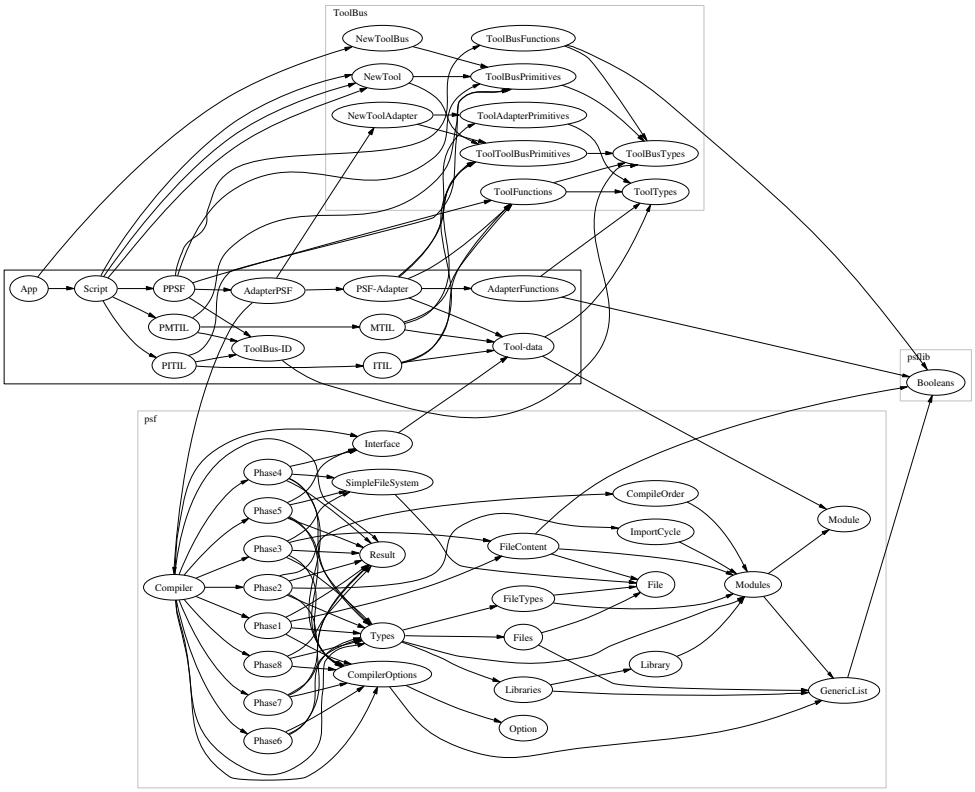


Figure 8-8. Import graph of the specification of the compiler as ToolBus application

Table 8-2. Performance of the compilers

	M1		M4	
	complete	update	complete	update
psf	5.5s	3.0s	5.5s	2.8s
tbpsf	17.2s	5.8s	7.5s	3.3s

8.5 Software Architecture

A software design consist of several levels, each lower one refining the design on the higher level. The highest level is often referred to as the architecture, the organization of the system as a collection of interacting components. In conventional software engineering processes, the architecture is usually described rather informally by means of a boxes-and-lines diagram. Following a lot of research going on in this area architectural descriptions are becoming more formal, especially due to the introduction of architectural description languages (ADLs). A specification in an ADL can be refined (in several steps) to a design from which an implementation of the system can be built. Here, the reverse has to be done.

Given a specification of a design in PSF one tries to extract the underlying architecture by means of an appropriate abstraction. The specification of the architecture will still be in PSF, however in such a way that one can generate an animation. This corresponds to the boxes-and-lines diagram but it is fully specified.

In the following sections we describe the possibilities for abstraction, and apply these to extract the architecture of the compiler.

8.5.1 Abstraction

In [52], action refinement is used as a technique for mapping abstract actions onto concrete processes, called virtual implementation, which is more fully described in [53]. For extracting the architecture from a specification we use the reverse of action refinement: action abstraction. One can do this by hiding internal actions of a component, and applying process algebra laws to combine consecutive internal actions into a single (internal) action. But also in this transformation step one has to abstract from implementation decisions that do not belong at the resulting higher level of abstraction. Often this can be done by only looking at the external behaviour of a component, its interface.

With parameterized actions, data terms are available which can also be refined. At a certain abstract level one does not care how data is implemented as long as the data is of a particular type. For instance in a message passing system one can deal with any message as just a message without knowing its content. Then for the specification at an abstract level one can use the zero-adic (constant) function *message* for the parameter of an action. In the specification at a lower level of abstraction this constant can be refined to a more complex term. Data abstraction is the reverse of this, we then replace complex terms with zero-adic functions. With this kind of abstraction, a receiving action of such a term can now use this zero-adic function instead of a variable coming from a summation construction.

8.5.2 Architecture of the Compiler

In the specification of the compiler the order of compilation steps is laid down. First all modules are parsed and then all modules are normalized. This is an implementation decision. A module can be normalized as soon as it has been parsed and all the modules it imports have been normalized. To abstract from this decision we specify the compiler with the following process.

```
PSF' =
   skip .
   (
      (
         skip .
         snd(do(tterm(tool-mtil), tterm(args))) .
         rec(result)
      +  skip .
         snd(do(tterm(tool-itil), tterm(args))) .
         rec(result)
      ) * snd(quit)
   )
```

Here, we use the abstract data terms 'args' and 'result'. This process describes the external

behaviour of the compiler. The skip actions are abstractions of internal actions.

The adapter for the compiler is defined as follows, where also the abstract form of the data terms is used.

```
PSF-Adapter =
   (
      tooladapter-rec(do(tterm(tool-mtil), tterm(args))) .
      tooltb-snd-event(tbterm(tterm(tool-mtil)),
         tbterm(tterm(args))) .
      tooltb-rec-ack-event(tbterm(tterm(tool-mtil))) .
      tooltb-rec(tbterm(tterm(result)) .
      tooladapter(tterm(result))
   +  tooladapter-rec(do(tterm(tool-itil), tterm(args))) .
      tooltb-snd-event(tbterm(tterm(tool-itil)),
         tbterm(tterm(args))) .
      tooltb-rec-ack-event(tbterm(tterm(tool-itil))) .
      tooltb-rec(tbterm(tterm(result)) .
      tooladapter(tterm(result))
   ) *
   tooladapter-rec(quit) .
   tooltb-snd-event(tbterm(quit))
```

The parallel composition of PSF' and PSF-Adapter combined with encapsulation of the communication actions is equivalent to the following process.

```
AdapterPSF' =
   skip .
   (
      (
         skip .
         tooladapter-comm(do(tterm(tool-mtil), tterm(args))) .
         tooltb-snd-event(PSF, tbterm(tterm(tool-mtil)), args)
         tooltb-rec-ack-event(tbterm(tterm(tool-mtil))) .
         tooltb-rec(result) .
         adaptertool-comm(tterm(result))
      +  skip .
         tooladapter-comm(do(tterm(tool-itil), tterm(args))) .
         tooltb-snd-event(PSF, tbterm(tterm(tool-mtil)), args)
         tooltb-rec-ack-event(tbterm(tterm(tool-mtil))) .
         tooltb-rec(result) .
         adaptertool-comm(tterm(result))
      ) *
      tooladapter-comm(quit) .
      tooltb-snd-event(tbterm(quit))
   )
```

We hide all internal actions of this process and replace the data terms with a more abstract form.

```
AdapterPSF'' =
   skip .
   (
      (
         skip .
         skip .
         tooltb-snd-event(PSF, tool-mtil, args)
         tooltb-rec-ack-event(tool-mtil) .
         tooltb-rec(result) .
         skip
      +  skip .
```

```
            skip .
            tooltb-snd-event(PSF, tool-mtil, args)
            tooltb-rec-ack-event(tool-mtil) .
            tooltb-rec(result) .
            skip
         ) *
         skip .
         tooltb-snd-event(quit)
      )
```

The ToolBus process PPSF (see page 83) with the data terms can be written in an abstract form as follows.

```
PPSF' =
         AdapterPSF''
      ||  (
            (
               tb-rec-event(PSF, tool-mtil, args) .
               tb-snd-ack-event(PSF, tool-mtil) .
               tb-snd-msg(psf, mtil, args) .
               tb-rec-msg(mtil, psf, result) .
               tb-snd-do(PSF, result)
            +  tb-rec-event(PSF, tool-itil, args) .
               tb-snd-ack-event(PSF, tool-itil) .
               tb-snd-msg(psf, itil, args) .
               tb-rec-msg(itil, psf, result) .
               tb-snd-do(PSF, result)
            ) *
            tb-rec-event(PSF, quit) .
            tb-shutdown
         )
```

After encapsulation of the communication actions between the tool and its ToolBus process this is equivalent to the following.

```
PPSF'' =
   skip .
   (
      (
         skip .
         skip .
         tb-comm-event(PSF, tool-mtil, args) .
         tb-comm-ack-event(PSF, tool-mtil) .
         tb-snd-msg(psf, mtil, args) .
         tb-rec-msg(mtil, psf, result) .
         tb-comm-do(PSF, result) .
         skip
      +  skip .
         skip .
         tb-comm-event(PSF, tool-itil, args) .
         tb-comm-ack-event(PSF, tool-itil) .
         tb-snd-msg(psf, itil, args) .
         tb-rec-msg(itil, psf, result) .
         tb-comm-do(PSF, result) .
         skip
      ) *
      skip .
      tb-comm-event(PSF, quit) .
      tb-shutdown
   )
```

Hiding all communications between the tool and the ToolBus process the following result is obtained.

```
PPSF''' =
   skip .
   (
      (
         skip .
         skip .
         skip .
         skip .
         tb-snd-msg(psf, mtil, args) .
         tb-rec-msg(mtil, psf, result) .
         skip .
         skip
      +  skip .
         skip .
         skip .
         skip .
         tb-snd-msg(psf, itil, args) .
         tb-rec-msg(itil, psf, result) .
         skip .
         skip
      ) *
      skip .
      skip .
      tb-shutdown
   )
```

Applying the τ-law $x.\tau = x$ of our process algebra yields

```
PPSF'''' =
   skip .
   (
      (
         skip .
         tb-snd-msg(psf, mtil, args) .
         tb-rec-msg(mtil, psf, result)
      +  skip .
         tb-snd-msg(psf, itil, args) .
         tb-rec-msg(itil, psf, result)
      ) *
      skip .
      tb-shutdown
   )
```

The same is done for the processes PMTIL and PITIL.

```
PMTIL'''' =
   (
      tb-rec-msg(psf, mtil, args) .
      tb-snd-msg(mtil, psf, result)
   ) * delta
PITIL'''' =
   (
      tb-rec-msg(psf, itil, args) .
      tb-snd-msg(itil, psf, result)
   ) * delta
```

The parallel composition of the above three processes describes the intended architecture. An animation of this architecture is shown in Figure 8-9. With some renaming the

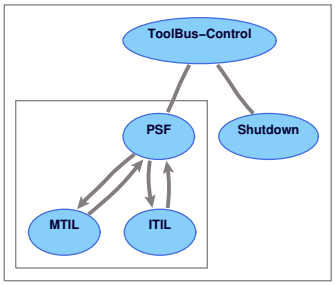


Figure 8-9. Animation of the architecture

processes can be written in a more suitable form.

```
Compiler = PSF || MTIL || ITIL
PSF =
   skip .
   (
      (
         skip .
         snd(psf, mtil, args) .
         rec(mtil, psf, result)
      +  skip .
         snd(psf, itil, args) .
         rec(itil, psf, result)
      ) *
      skip .
      shutdown
   )
MTIL =
   (
      rec(psf, mtil, args) .
      snd(mtil, psf, result)
   ) * delta
ITIL =
   (
      rec(psf, itil, args) .
      snd(itil, psf, result)
   ) * delta
```

The above PSF text provides a specification of the compiler architecture. The architecture does not enforce any restrictions on the type of connections used to *glue* the various components together. Both the original compiler as well as the re-engineered version compiler that makes use of the ToolBus are implementations of this architecture.

8.6 Parallel Compiler

The parsing and normalization of modules allows for parallelization. Parsing of modules and the normalization of other modules which already have been parsed and for which all the modules that they import have already been normalized, can be done in parallel. We build a parallel compiler and reuse as much as possible from the specifications and implementation of the re-engineered compiler.

8.6.1 Architecture

Instead of issuing commands for parsing and normalization of modules, the parallel compiler should compose an information structure that tells which modules have to be parsed and/or normalized and on which modules they depend that also have to be parsed and/or normalized. The compiler has to send this structure to a scheduler which decides when modules are to be parsed or normalized.

We give here the specification of the architecture for the parallel compiler.

```
Compiler = PSF || Scheduler || MTIL || ITIL
PSF =
   skip .
   (
      skip .
      tb-snd-msg(psf, scheduler, compile-info) .
      tb-rec-msg(scheduler, psf, result) .
      tb-shutdown
   +  skip .
      tb-shutdown
   )
Scheduler =
   tb-rec-msg(psf, scheduler, compile-info) .
   (
      (
         skip .
         tb-snd-msg(scheduler, mtil, args)
      +  tb-rec-msg(mtil, scheduler, result)
      +  skip .
         tb-snd-msg(scheduler, itil, args)
      +  tb-rec-msg(itil, scheduler, result)
      ) * tb-snd-msg(scheduler, psf, result)
   )
MTIL =
   (
      tb-rec-msg(scheduler, mtil, args) .
      tb-snd-msg(mtil, scheduler, result)
   ) * delta
ITIL =
   (
      tb-rec-msg(scheduler, itil, args) .
      tb-snd-msg(itil, scheduler, result)
   ) * delta
```

An animation of this architecture is shown in Figure 8-10. This specification features only one MTIL and one ITIL process, but this scheme allows for more MTIL and ITIL processes in parallel. Whichever process is free can pick up a request from the scheduler to parse or normalize a module.

Although the specification of the architecture contains separate processes for compiler and scheduler, it does not imply that these need to be implemented as separate tools. The scheduler can be incorporated in the compiler, as we show below.

The parallel composition of PSF and Scheduler, is equivalent to the following process.

```
   skip . (
      skip . tb-comm-msg(psf, scheduler, compile-info) .
      (
```

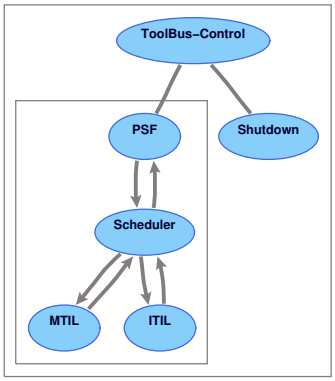


Figure 8-10. Animation of the architecture

```
        P * tb-comm-msg(scheduler, psf, result) . tb-shutdown
    )
+   Q
)
```

Here, `P` stands for the alternative composition of the send and receive actions in the Scheduler process, and `Q` stands for **skip .** `tb-shutdown`.

Hiding the communications between compiler and scheduler results in the following.

skip . (**skip . skip .** `(P * Q) + Q`)

Applying the τ-law $x.\tau = x$ gives

skip . (**skip .** `(P * Q) + Q`)

Applying the algebraic law for iteration $x * y = (x * y) + y$ gives

skip . (**skip .** `((P * Q) + Q) + Q`)

Applying the τ-law $v.(\tau.(x + y) + x) = v.(x + y)$ gives

skip . `((P * Q) + Q)`

Applying the algebraic law for iteration $x * y = (x * y) + y$ in reverse gives

skip . `(P * Q)`

Replacing `P` and `Q` gives us

```
skip .
(
      skip .
      tb-snd-msg(scheduler, mtil, args)
  +   tb-rec-msg(mtil, scheduler, result)
  +   skip .
      tb-snd-msg(scheduler, itil, args)
  +   tb-rec-msg(itil, scheduler, result)
  ) * skip .
  tb-shutdown
)
```

This looks the same as the compiler process in the architecture of tbpsf but then with the

sending and receiving actions in parallel with the scheduler process in this architecture.

8.6.2 Specification of the Parallel Compiler

As we already mentioned in the previous section, there are several options for the cooperation of the compiler and the scheduler. A possibility is to incorporate the scheduler in the compiler and let the scheduler part take care of the connections with the ToolBus. Here, however, we have chosen to implement the Scheduler as a separate process (tool) to be connected to the ToolBus which gets its information from the compiler over the ToolBus.

We reuse a large part of the specification of the compiler for the specification of the parallel compiler. The parsing and normalization phases are replaced by a phase that builds up a Compiler-Information structure. The ToolBus processes are adjusted and extended to reflect the processes in the specification of the architecture, and the specification of the scheduler is added. The animation of the parallel compiler is shown in Figure 8-11.

The import graph for the specification of the parallel compiler is shown in Figure 8-12. The module Naturals with all its imports[4] and the module Tables stem from the standard library of PSF. Naturals is used for counting the MTIL and ITIL processes that have to be started in the specification of the ToolBus script and Tables is used for the construction of the Compiler-Information structure.

8.6.3 Implementation of the Parallel Compiler

The implementation of the compiler has been extended with a phase for building the Compiler-Information structure which can be invoked instead of the parsing and normalizing phases, controlled by an option. The scheduler has been implemented in Perl. The actual application is a Perl script that provides an environment with the right settings and which will invoke the ToolBus according to the arguments given on the command line. This script also gives the possibility to start the parallel compiler with indicated numbers of parsing and normalization processes.

In Table 8-3 the performance of the parallel compiler is shown for several combinations of numbers of parsing and normalization processes, for the complete compilation of the specification of the compiler as a ToolBus application. We see that on configuration M1 the parallel compiler has a better performance than tbpsf, but it is not faster than psf. So the communication overhead connected with the ToolBus is too large to overcome on this configuration. Parallel compilation on configuration M4 is faster than psf, although not much, because the amount of work that can be done in parallel is limited by the imposed order of compilation of the modules due to their import relation.

---

4. The imported modules by the Naturals are shown as a single module in gray for clarity of the Figure.

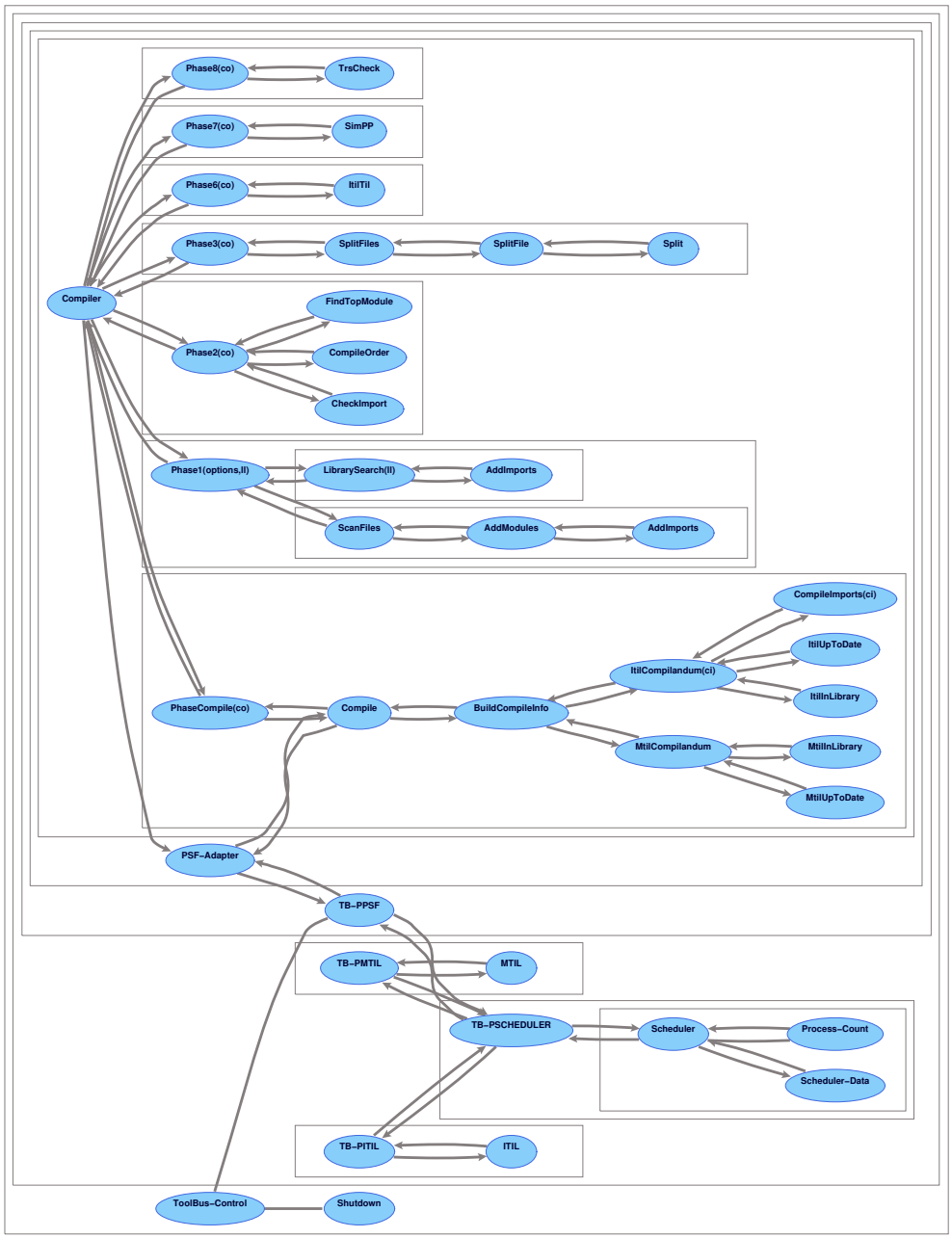


Figure 8-11. Generated animation of the parallel compiler as ToolBus application

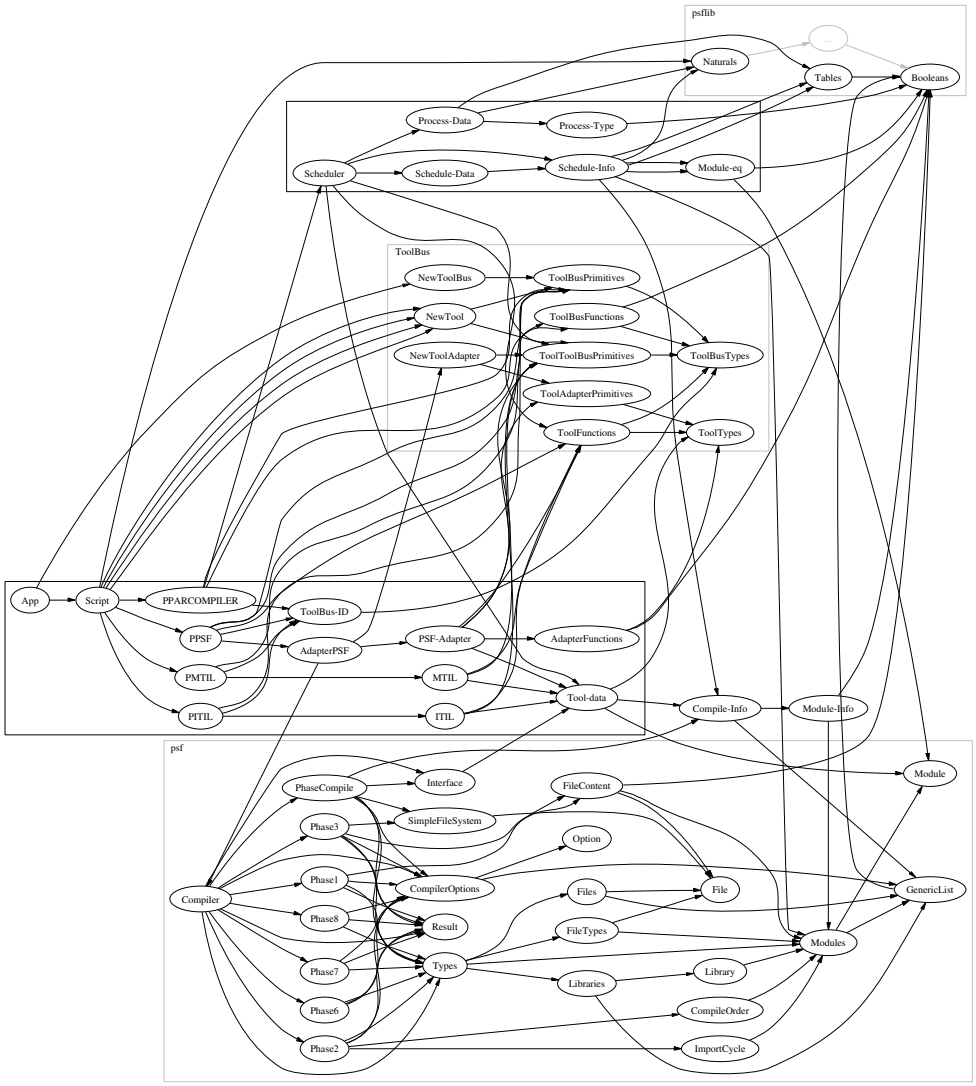


Figure 8-12. Import graph of the specification of the parallel compiler

Table 8-3. Performance of the parallel compiler

# processes		complete	
mtil	itil	M1	M4
1	1	12.8s	6.0s
1	2	11.7s	5.4s
1	3	11.6s	5.4s
2	1	12.9s	6.1s
2	2	12.1s	5.2s
2	3	11.9s	4.8s
2	4	11.6s	4.9s
3	2	11.7s	5.0s
3	3	11.6s	4.8s
3	4	11.6s	4.9s
4	4	11.6s	4.7s

Chapter 9

Software Architecture with PSF

In section 8.5 we introduced software architecture as the highest level of abstraction in software design. We extracted a specification of the architecture of the PSF compiler from the specification of the compiler by abstracting from implementation decisions. With some renamings this resulted in the architecture shown in Figure 8-9. Software architectures are usually described rather informally by means of boxes-and-lines diagrams. Following a lot of research in this area, architectural descriptions are becoming more formal, especially due to the introduction of architectural description languages (ADLs). A specification in an ADL can be refined (in several steps) to a design from which an implementation of the system can be built.

In this chapter we present a PSF library for specifying software architectures. This provides the formalization of boxes-and-lines diagrams used for denotation of and communication on software architectures. With the use of the PSF Toolkit it is possible to generate an animation from the specification which can be brought to life with the simulator of the Toolkit. Furthermore, we present vertical and horizontal implementation techniques to obtain a ToolBus application specification from an architecture specification.

In section 9.1 we describe the PSF Architecture Library and in section 9.2 we give an example of the use of this library. We introduce the vertical and horizontal implementation techniques in section 9.3, where we also demonstrate these techniques by applying them on our example.

9.1 PSF Architecture Library

First we define the types for the id's of the components, the connections between components, and the data.

```
data module ArchitectureTypes
begin
  exports
  begin
```

```
        sorts
            ID,
            CONNECTION,
            DATA
        functions
            _>>_ : ID # ID → CONNECTION
    end
end ArchitectureTypes
```

The sorts `ID` and `DATA` are abstract data types. Elements of these sorts have to be provided with the specification of an architecture. We could do without the function >> and use just two `ID` arguments instead of one `CONNECTION` argument, but now the connection clearly stands out from other terms and therefore makes the specifications easier to read.

We define the primitives for the communication between the components and a quitting action that communicates with the architecture environment.

```
process module ArchitecturePrimitives
begin
    exports
    begin
        atoms
            snd : CONNECTION # DATA
            rec : CONNECTION # DATA
            comm : CONNECTION # DATA

            snd-quit
    end
    imports
        ArchitectureTypes
    communications
        snd(c, s) | rec(c, s) = comm(c, s)
            for c in CONNECTION, s in DATA
end ArchitecturePrimitives
```

Note that we do not specify a particular kind of connection. In our belief the choice of the kind of connection should not be made on the architecture level, but on a lower level.

We now specify the architecture environment parameterized with the architecture specification.

```
process module Architecture
begin
    parameters
        System
        begin
            processes
                System
        end System
    exports
    begin
        processes
            Architecture
    end
    imports
        ArchitecturePrimitives
    atoms
        rec-quit
        quit
        snd-shutdown
```

```
        rec-shutdown
        shutdown
    processes
        ArchitectureControl
        ArchitectureShutdown
    sets
        of atoms
            H = {
                snd(c, s), rec(c, s) | c in CONNECTION, s in DATA
            }
            ArchitectureH = {
                snd-quit, rec-quit,
                snd-shutdown, rec-shutdown
            }
    communications
        snd-quit | rec-quit = quit
        snd-shutdown | rec-shutdown = shutdown
    definitions
        Architecture =
              encaps(ArchitectureH,
                  disrupt(
                      encaps(H, System),
                      ArchitectureShutdown
                  )
              ||  ArchitectureControl
              )
        ArchitectureControl =
              rec-quit .
              snd-shutdown
        ArchitectureShutdown = rec-shutdown
end Architecture
```

PSF does not have a single action to end all processes. Such an action is actually a communication with the environment in which the processes run and this environment has to end all processes. We have specified this behaviour with the processes ArchitectureControl as the environment, ArchitectureShutdown to disrupt the running of the processes, and splitting up the actions quit and shutdown in a send and receive part.

9.2 Example

For showing the use of the PSF Architecture library we consider the same example we used for the ToolBus and for the PSF ToolBus library (see sections 4.1 and 8.3).

We first specify a module for the data and id's we use.

```
data module Data
begin
    exports
    begin
        functions
            message : → DATA
            ack : → DATA
            quit : → DATA

            c1 : → ID
            c2 : → ID
    end
    imports
```

```
        ArchitectureTypes
end Data
```

We then specify the system of our application.

```
process module ApplicationSystem
begin
   exports
   begin
      processes
         ApplicationSystem
   end
   imports
      Data,
      ArchitecturePrimitives
   atoms
      send-message
      stop
   processes
      Component1
      Component2
   definitions
      Component1 =
            send-message .
            snd(c1 >> c2, message) .
            rec(c2 >> c1, ack) .
            Component1
         +  stop .
            snd-quit
      Component2 =
         rec(c1 >> c2, message) .
         snd(c2 >> c1, ack) .
         Component2
      ApplicationSystem = Component1 || Component2
end ApplicationSystem
```

The `snd-quit` in the process definition for Component1 communicates with the architecture environment followed by a disrupt to end all processes.

Next, we put the system in the architecture environment by means of binding the main process to the System parameter of the environment.

```
process module Application
begin
   imports
      Architecture {
         System bound by [
            System → ApplicationSystem
         ] to ApplicationSystem
         renamed by [
            Architecture → Application
         ]
      }
end Application
```

The generated animation of the architecture is shown in Figure 9-1. Here, Component1 has just sent a message to Component2, which is ready to send an acknowledgement back. Each box represents an encapsulation of the processes inside the box, and a darker ellipse is a process which is enabled to perform an action in the given state.

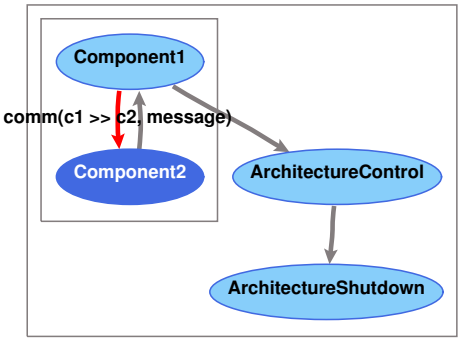


Figure 9-1. Animation of an example architecture

The module mechanism of PSF can be used for more complex components to hide the internal actions and sub-processes of a component. With the use of parameterization it is even possible to make several instances of a component.

9.3 From Architecture to ToolBus Application Design

It is only useful to invest a lot of effort in the architecture if we can relate it to a design on a lower level. In this section we describe the techniques we use to get from an architecture specification to a ToolBus application specification. These techniques are important in our software engineering process as they compress a large number of algebraic law applications into a few steps and so make the process feasible. We demonstrate these techniques with our toy example from section 8.3.

9.3.1 Horizontal Implementation

Given two processes S and I, I is an implementation of S if I is more deterministic than (or equivalent to) S. As the actions S and I perform belong to the same alphabet, S and I belong to the same level of abstraction. Such an implementation relation is called *horizontal*.

To achieve a horizontal implementation we use parallel composition, which can be used to constrain a process. Consider process $P = a \cdot P$, which can do action a at every moment. We put P in parallel with the process $Q = x \cdot b \cdot Q$, where x is a local action of Q, and we define the communication $a \mid b = c$. If we enforce the communication by encapsulation, process P can only do action a whenever process Q has first done action x. So process P is constrained by Q and $P \between Q$ is a horizontal implementation of P, provided Q only interacts with P through b. This form of controlling a process is also known as *superimposition* [11] or *superposition* [34].

9.3.2 Vertical Implementation

In [52], action refinement is used as a technique for mapping abstract actions onto concrete processes, called *vertical* implementation, which is described more extensively in [53].

With vertical implementation we want to relate processes that belong to different levels of abstraction, where the change of level usually comes with a change of alphabet. For such processes we like to develop *vertical* implementation relations that, given an abstract process S and a concrete process I, tell us if I is an implementation for the specification S. More specifically, we want to develop a mapping of abstract actions to sequences of one or more concrete actions so that S and I are *vertical bisimilar*.

We give a rationale of vertical implementation. Consider the processes $P = a \cdot b$ with a an internal action and $Q = c \cdot d \cdot e$ with internal actions c and d. If we refine abstract action a from process P to the sequence of concrete actions $c \cdot d$ and rename action b to e we obtain process Q. The processes P and Q are vertical bisimilar with respect to the mapping consisting of the above refinement and renaming.

We can explain the notion *vertical bisimilar* by the following. We hide the internal action a of process P by replacing it with the silent step τ to obtain $P = \tau \cdot b$. Applying the algebraic law $x \cdot \tau = x$ gives us $P = \tau \cdot \tau \cdot b$. If we now replace the first τ with c and the second with d, and rename b into e we obtain the process Q. With H as hide operator and R as renaming operator we can prove that $R_{\{b \to e\}}(H_{\{a\}}(P))$ and $H_{\{c,d\}}(Q)$ are rooted weak bisimilar (see Figure 9-2). So vertical bisimulation is built on rooted weak bisimulation as horizontal implementation relation.

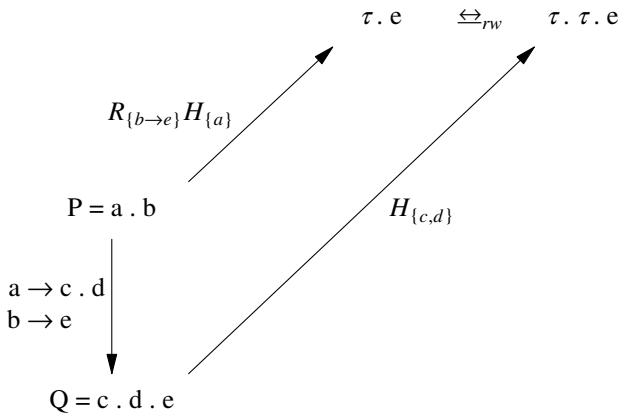


Figure 9-2. Implementation relations

9.3.3 Example

Take the process `Component1` from the architecture of our toy example.

```
Component1 =
      send-message .
      snd(c1 >> c2, message) .
      rec(c2 >> c1, ack) .
      Component1
   +  stop .
```

```
      snd-quit
```

We can make a virtual implementation by applying the following mapping.

```
send-message          → tb-rec-event(T1, tbterm(message))
snd(c1 >> c2, message)→ tb-snd-msg(t1, t2, tbterm(message))
rec(c2 >> c1, ack)    → tb-rec-msg(t2, t1, tbterm(ack)) .
                        tb-snd-ack-event(T1, tbterm(message))
stop                  → tb-rec-event(T1, tbterm(quit))
snd-quit              → snd-tb-shutdown
```

And renaming `Component1` into `PT1` gives the following result.

```
PT1 =
      tb-rec-event(T1, tbterm(message)) .
      tb-snd-msg(t1, t2, tbterm(message)) .
      tb-rec-msg(t2, t1, tbterm(ack)) .
      tb-snd-ack-event(T1, tbterm(message)) .
      PT1
   +  tb-rec-event(T1, tbterm(quit)) .
      snd-tb-shutdown
```

In a similar way an implementation (PT2) for `Component2` can be obtained.

We can show that `Component1` and `PT1` are vertical bisimilar. The mapping consist of the refinements

```
snd(c1 >> c2, message)→ tb-snd-msg(t1, t2, tbterm(message))
rec(c2 >> c1, ack)    → tb-rec-msg(t2, t1, tbterm(ack)) .
                        tb-snd-ack-event(T1, tbterm(message))
snd-quit              → snd-tb-shutdown
```

and the renamings of the local actions

```
send-message          → tb-rec-event(T1, tbterm(message))
stop                  → tb-rec-event(T1, tbterm(quit))
```

Applying the renamings on process `Component1` and hiding of the to be refined actions results in

```
Component1' =
      tb-rec-event(T1, tbterm(message)) . τ . τ . Component1'
   +  tb-rec-event(T1, tbterm(quit)) . τ
```

Hiding of the actions in the refinements in process `PT1` results in

```
PT1' =
      tb-rec-event(T1, tbterm(message)) . τ . τ . τ . PT1'
   +  tb-rec-event(T1, tbterm(quit)) . τ
```

It follows that `Component1'` $\leftrightarrows_{rw}$ `PT1'`.

We now make horizontal implementations for `PT1` and `PT2` by constraining them with `Tool1Adapter` and `Tool2` from section 8.3.1.

```
PTool1 = Tool1Adapter ‖ PT1
PTool2 = Tool1 ‖ PT1
```

Note that `Tool1Adapter` is itself a constraining of `AdapterTool1` with `Tool1`.

Chapter 10

A New PSF Simulator

In Chapter 8 we described the PSF ToolBus library for specifying ToolBus application specifications and in Chapter 9 we described the PSF Architecture library for specifying software architectures. In this chapter we develop a new implementation for the simulator of the PSF Toolkit using these PSF libraries together with the refinement techniques that were also introduced in Chapter 9. From the requirements for the new simulator we develop an architecture specification, starting with a specification for a simple simulator and extending it with the necessary features. We develop a ToolBus application specification from the architecture specification using the refinement techniques. The ToolBus application specification serves as a base for the implementation of the simulator as a ToolBus application. The development process is depicted in Figure 10-1.

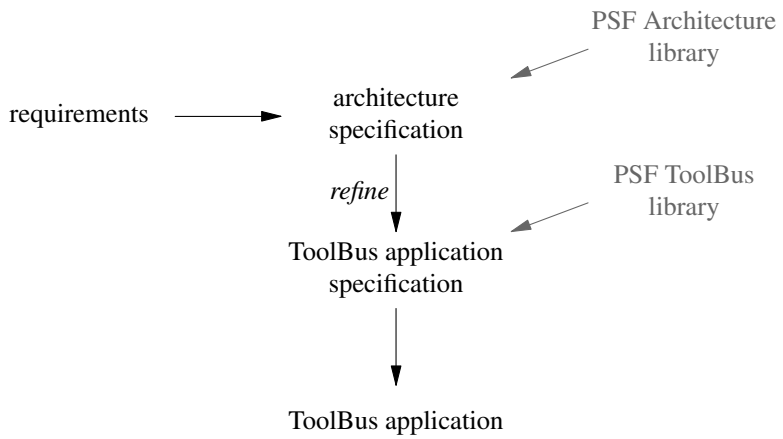


Figure 10-1. Development process for the simulator

The purpose of this case study is to obtain experience with the use of PSF in software engineering and to improve the implementation of the simulator.

In section 10.1 we describe the requirements for the new simulator. We develop an architecture specification for the new simulator in section 10.2. We refine the architecture specification into a ToolBus application specification in section 10.3 and in section 10.4 we describe the implementation of the new simulator. In section 10.5 we describe how to aggregate the graphical user interfaces for the different components of the new simulator into a single graphical user interface. In section 10.6 we extend the new simulator with a history mechanism showing the impact of a software evolution process iteration on the design process. We describe the coupling of animation to the new simulator in section 10.7. We discuss some of the functionality for the old simulator that is not implemented in the new simulator in section 10.8 and in section 10.9 we compare the old simulator with the new simulator.

10.1 Requirements

Although the existing version of the simulator from the PSF Toolkit is satisfactory, we think its implementation can be improved a lot. Its interface is outdated and the internal complexity can be lifted from the kernel of the simulator and pushed to separate components and their interaction. We give in this section the requirements for the new simulator without going into much detail, merely to give an idea of what the simulator should be capable of and what we expect from the new design.

10.1.1 Functional Requirements

The functional requirements we list here stem from the functionality of the old simulator. Some features have been left out because they are rarely used and can be established in a different way, such as reloading of specifications and argument selection of start processes.

The simulator should be able to simulate PSF specifications (or rather a compiled form of these) according to the semantics, and it must at least fulfill the following requirements.

- Selection of a process to start the simulation with from a list of possible processes.
- Simple interaction with the user for choosing an action to be executed from a list of possible executable actions at a certain state. Simple means that the actions are presented in a single unordered list.
- Show on request the status of processes currently being simulated in a way that their correlation is visible and how the list of possible actions is determined from them.
- Make it possible to trace certain actions as they are executed. These actions must be selected from all actions in an easy manner.
- Be able to run randomly and stop this whenever one or more breakpoints are encountered. That moment can be on execution of an action on which a breakpoint is set, when one or more actions with breakpoints set on them appear in the list of possible action, or when all actions in the list have breakpoints on them (synchronization). Selection of breakpoints should be made easy, preferably in a

similar way of selecting actions to be traced.

- A history mechanism that not only makes it possible to undo or redo a step, but also to go to a previously marked state.

10.1.2 Non-functional Requirements

The non-functional requirements we list here represent our wishes as opposed to the implementation of the old simulator.

- A modular design with easy to replace components. Especially, the simulator should have a separate kernel which can be used in other applications than the simulator.
- Can be used as a framework for simulating other languages similar to PSF, or variants of PSF.
- The user interface should be less dependent on the X Window System than the old simulator, and should be easy to adapt to changes in environment, application, or user demands.
- Easy coupling of the simulator with animation.

10.2 Architecture Specification

We specify the architecture in several steps, starting with the architecture of a simple simulator to which we add the features. The architecture specification as presented here is the result of common software development processes incorporated with an architecture phase.[5] In these processes there is feedback from successive phases, and so also the architecture phase gets this feedback.

10.2.1 A Simple Simulator

Our simple simulator consists of four system components.

kernel	does the actual simulation.
startprocess	takes care of choosing a process to start the simulating with.
actionchooser	takes care of choosing an action from a list of possible actions it receives from the kernel.
display	displays the information the other components wish to communicate to the user.

We first specify the id's for the four components and the data (in an abstract form) that are used in the communication between components in a separate module.

```
data module SimulatorData
begin
    exports
```

5. See [56] for an overview of software development processes.

```
    begin
        functions
            kernel : → ID
            startprocess : → ID
            actionchooser : → ID
            display : → ID

            start-process : → DATA
            action-choose-list : → DATA
            action : → DATA
            halt : → DATA
            reset : → DATA
    end
    imports
        ArchitectureTypes
end SimulatorData
```

The kernel can be in two states. One in which it actually simulates, and one in which it is waiting for communication with other components. This is specified using a boolean variable `wait`.

```
process module Kernel
begin
    exports
    begin
        processes
            Kernel
    end
    imports
        SimulatorData,
        ArchitecturePrimitives,
        Booleans
    atoms
        compute-choose-list
        compute-halt
    processes
        Kernel : BOOLEAN
    variables
        wait : → BOOLEAN
    definitions
        Kernel = Kernel(true)
        Kernel(wait) =
            (
              [wait = false] → (
                 compute-choose-list .
                 snd(kernel >> actionchooser, action-choose-list)
              +  compute-halt .
                 snd(kernel >> display, halt)
              ) .
              Kernel(true)
            + [wait = true] → (
                 rec(actionchooser >> kernel, action) .
                 Kernel(false)
              +  rec(startprocess >> kernel, start-process) .
                 snd(kernel >> display, start-process) .
                 snd(kernel >> actionchooser, reset) .
                 Kernel(false)
              )
            )
end Kernel
```

If the kernel is not in the wait state, there is a choice between two internal actions. The action `compute-choose-list` resembles the computation of a list of possible actions that can occur. This list is sent to the actionchooser. The other action `compute-halt` indicates that the kernel could not compute a list of possible actions, either because simulation ended, or a deadlock occurred. In the wait state the kernel can receive a `start-process` from the startprocess component, or it can receive an `action` from the actionchooser.

The startprocess component is very simple, it can only select a start process for simulation and send this process to the kernel.

```
process module StartProcess
begin
   exports
   begin
      processes
         StartProcess
   end
   imports
      ArchitecturePrimitives,
      SimulatorData
   atoms
      select-start-process
   definitions
      StartProcess =
         (
            select-start-process .
            snd(startprocess >> kernel, start-process)
         ) * delta
end StartProcess
```

The actionchooser can receive an `action-choose-list` or a `reset` from the kernel. When it receives an `action-choose-list` it can send an `action` to the kernel.

```
process module ActionChooser
begin
   exports
   begin
      processes
         ActionChooser
   end
   imports
      ArchitecturePrimitives,
      SimulatorData,
      Booleans
   atoms
      choose-action
   processes
      Choose : BOOLEAN
      Reset
   variables
      choose : → BOOLEAN
   definitions
      ActionChooser = Choose(false)
      Choose(choose) =
            rec(kernel >> actionchooser, action-choose-list) .
            Choose(true)
         + [choose = true] → (
               choose-action .
```

```
                (
                   snd(actionchooser >> kernel, action) .
                   Choose(false)
                +  Reset
                )
             )
          +  Reset
       Reset = rec(kernel >> actionchooser, reset) .
             Choose(false)
end ActionChooser
```

The possibility for a reset after an action has been chosen is necessary, otherwise a deadlock can occur when the kernel sends a reset caused by the receipt of a start-process.

The display can only receive data from other components. At this point in the design, it receives only from the kernel.

```
process module Display
begin
   exports
   begin
      processes
         Display
   end
   imports
      ArchitecturePrimitives,
      SimulatorData
   definitions
      Display =
         (
            rec(kernel >> display, halt)
         +  rec(kernel >> display, start-process)
         ) * delta
end Display
```

We combine the components to a system by merging the processes of the components.

```
process module SimulatorSystem
begin
   exports
   begin
      processes
         SimulatorSystem
   end
   imports
      Kernel,
      StartProcess,
      ActionChooser,
      Display
   definitions
      SimulatorSystem =
            Kernel
         ||  StartProcess
         ||  ActionChooser
         ||  Display
end SimulatorSystem
```

We complete the architecture of the simple simulator by putting the system in the architecture environment.

```
process module Simulator
```

```
begin
    imports
        Architecture {
            System bound by [
                System → SimulatorSystem
            ] to SimulatorSystem
            renamed by [
                Architecture → Simulator
            ]
        }
end Simulator
```

An animation of the architecture is shown in Figure 10-2.

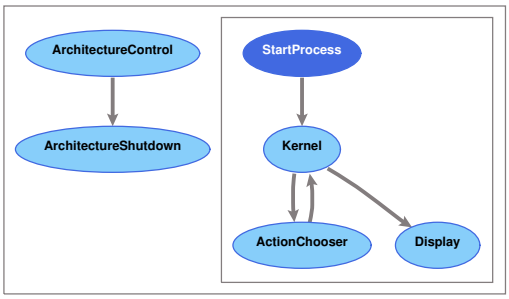


Figure 10-2. Architecture of a simple simulator

10.2.2 Functions

We extend the simple simulator with two functions that can be invoked by the user, `quit` and `process-status`.

To module SimulatorData we add the id `function` and data terms for the functions. And we add a module Function.

```
process module Function
begin
    exports
    begin
        processes
            Function
    end
    imports
        ArchitecturePrimitives,
        SimulatorData
    atoms
        push-quit
        push-process-status
    definitions
        Function =
            (
                push-quit .
                snd(function >> kernel, quit)
            +   push-process-status .
                snd(function >> kernel, process-status)
            ) * delta
```

end `Function`

To module Kernel we add the following alternatives to the wait state.

```
+  rec(function >> kernel, quit) .
   snd-quit
+  rec(function >> kernel, process-status) .
   snd(kernel >> display, process-status) .
   Kernel(wait)
```

After the kernel receives a `quit` it communicates with the architecture environment by means of a `snd-quit` on which the environment acts with a shutdown. And on receiving `process-status` it send the process status to the display (we use the same abstract data term here).

To the module Display we add an alternative for receiving a `process-status` message and in the module SimulatorSystem we merge the process Function with the other processes. The animation of the resulting architecture is shown in Figure 10-3.

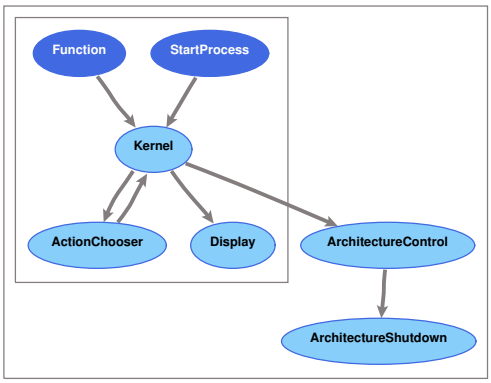


Figure 10-3. Architecture with functions

10.2.3 Tracing

We now add a component tracectrl that takes care of the tracing of actions (make them visible to the user) the moment they are executed. Whenever an action is chosen by the actionchooser it is send to tracectrl which decides, on indication by the user, whether it has to be traced, in which case a message is send to display. So it acts as a filter.

To module SimulatorData we add the id `tracectrl` and as data terms `trace-action` and `done`.

```
process module TraceCtrl
begin
    exports
    begin
        processes
            TraceCtrl
    end
    imports
```

```
        SimulatorData,
        ArchitecturePrimitives
    atoms
        trace
        no-trace
    definitions
        TraceCtrl =
            (
                rec(actionchooser >> tracectrl, action) .
                (
                    trace .
                    snd(tracectrl >> display, trace-action)
                +   no-trace
                ) .
                snd(tracectrl >> actionchooser, done)
            ) * delta
end TraceCtrl
```

The confirmation to the actionchooser is necessary, otherwise it is possible that the actionchooser continues and another message is sent to the display before a trace message is sent, and so a mix-up of the order of the messages on the display can occur.

We add the communication with tracectrl in the actionchooser directly after `action` is send to the kernel, as shown below with existing code in grey.

```
+  [choose = true] → (
       snd(actionchooser >> kernel, action) .
       snd(actionchooser >> tracectrl, action) .
       rec(tracectrl >> actionchooser, done) .
       Choose(false)
   )
```

To module Display we add an alternative for receiving a `trace-action` message and we add TraceCtrl to SimulatorSystem. The resulting architecture is shown in Figure 10-4.

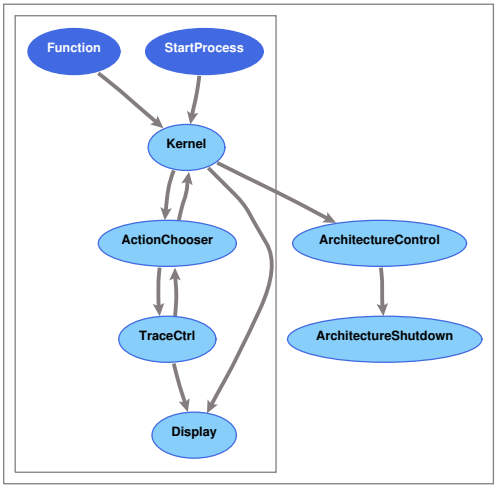


Figure 10-4. Architecture with tracing

10.2.4 Random

At this moment it is of no concern whether the user wants to let the actionchooser choose actions randomly, so this can be kept implicit with the actionchooser. But when we introduce breakpoints in order to stop the simulator from running randomly at certain moments, we need to know whether the simulator is running randomly explicitly. So we add a control state to the Choose process of the actionchooser and the possibility to switch random on and off.

```
process module ActionChooser
begin
   exports
   begin
      processes
         ActionChooser
   end
   imports
      ArchitecturePrimitives,
      SimulatorData,
      Booleans
   atoms
      choose-action
      random-on
      random-off
   processes
      Choose : BOOLEAN # BOOLEAN
      Reset : BOOLEAN
   variables
      random : → BOOLEAN
      choose : → BOOLEAN
   definitions
      ActionChooser = Choose(false, false)
      Choose(random, choose) =
            rec(kernel >> actionchooser, action-choose-list) .
            Choose(random, true)
         + [choose = true] → (
               choose-action .
               (
                  snd(actionchooser >> kernel, action) .
                  snd(actionchooser >> tracectrl, action) .
                  rec(tracectrl >> actionchooser, done) .
                  Choose(random, false)
               + Reset(random)
               )
            )
         + Reset(random)
         + [random = true] → (
               random-off .
               Choose(false, choose)
            )
         + [random = false] → (
               random-on .
               Choose(true, choose)
            )
      Reset(random) =
            rec(kernel >> actionchooser, reset) .
            Choose(random, false)
end ActionChooser
```

10.2.5 Breakpoints

In order to stop the simulator from running randomly at certain moments we add breakpoints. There are two type of breakpoints. One is when an action (indicated by the user) gets executed, and the other is when the list of possible actions contains one or more actions on which the user has set a breakpoint.

To module SimulatorData we add the id `breakctrl` and as data terms `break-action`, `break` end `no-break`.

```
process module BreakCtrl
begin
   exports
   begin
      processes
         BreakCtrl
   end
   imports
      SimulatorData,
      ArchitecturePrimitives
   atoms
      break
      no-break
      break-list
      no-break-list
   definitions
      BreakCtrl =
         (
            rec(actionchooser >> breakctrl, action) .
            (
               break .
               snd(breakctrl >> display, break-action) .
               snd(breakctrl >> actionchooser, break)
            +  no-break .
               snd(breakctrl >> actionchooser, no-break)
            )
         +  rec(actionchooser >> breakctrl, action-choose-list) .
            (
               no-break-list .
               snd(breakctrl >> actionchooser, action-choose-list)
            +  break-list .
               snd(breakctrl >> display, break) .
               snd(breakctrl >> actionchooser, break)
            )
         ) * delta
end BreakCtrl
```

In module ActionChooser we replace

```
rec(kernel >> actionchooser, action-choose-list) .
Choose(random, true)
```

with

```
rec(kernel >> actionchooser, action-choose-list) .
(
   [random = true] → (
      snd(actionchooser >> breakctrl,
         action-choose-list) .
      (
```

```
                        rec(breakctrl >> actionchooser, break) .
                        force-random-off .
                        present-list .
                        Choose(false, true)
                     +  rec(breakctrl >> actionchooser,
                           action-choose-list) .
                        present-list .
                        Choose(true, true)
                     )
                  )
               +  [random = false] → (
                     present-list .
                     Choose(false, true)
                  )
               )
```

and

```
snd(actionchooser >> tracectrl, action) .
rec(tracectrl >> actionchooser, done) .
Choose(random, false)
```

with

```
(
   [random = true] → (
      snd(actionchooser >> breakctrl, action) .
      (
         rec(breakctrl >> actionchooser, break) .
         force-random-off .
         Choose(false, false)
      +  rec(breakctrl >> actionchooser, no-break) .
         snd(actionchooser >> tracectrl, action) .
         rec(tracectrl >> actionchooser, done) .
         Choose(true, false)
      )
   )
+  [random = false] → (
      snd(actionchooser >> tracectrl, action) .
      rec(tracectrl >> actionchooser, done) .
      Choose(false, false)
   )
)
```

We also add the introduced actions `forced-random-off` and `present-list` to the atoms section of the module ActionChooser. The action `force-random-off` is necessary because it clearly differs from `random-off` which is invoked by the user. The action `present-list` has a more complex explanation. In the old situation this action could be combined with the receiving of the `action-choose-list`, we now have to do this later in the process because we first have to check on possible breakpoints in the case of random simulation.

To module Display we add alternatives for receiving a `break-action` and a `break` message and we add BreakCtrl to SimulatorSystem. The resulting architecture is shown in Figure 10-5.

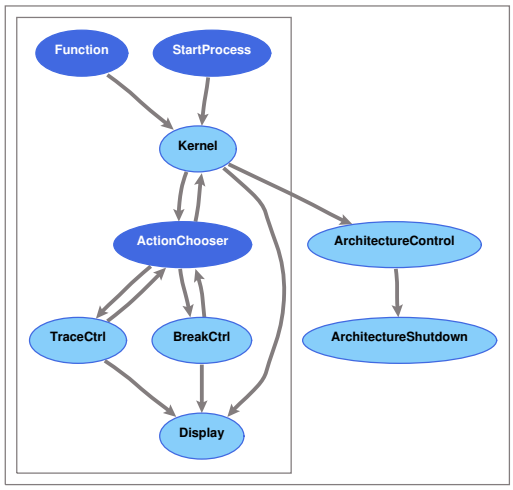


Figure 10-5. Architecture with breakpoints

10.3 System Specification

We take the specification of the architecture of the simulator and turn it into a specification of a ToolBus application with the use of the techniques described in the previous chapter.

10.3.1 Refining

Here we show the mapping for the vertical implementation of the architecture specification. We start with some default mappings that only apply when there are no other mappings to apply.

```
snd($1 >> $2, $3)   → tb-snd-msg($1, $2, tbterm($3))
rec($1 >> $2, $3)   → tb-rec-msg($1, $2, tbterm($3))
```

The `$n` on the left hand side represent matched terms that have to be filled in on the right hand side. Below the mappings per module are given.

Kernel

```
compute-choose-list   →
              tb-snd-eval(KERNEL, tbterm(compute-choose-list))
action-choose-list →
              tb-rec-value(KERNEL, tbterm(action-choose-list))
halt       → tb-rec-value(KERNEL, tbterm(halt))
rec(actionchooser >> kernel, action)→
              tb-rec-msg(actionchooser, kernel, tbterm(action)) .
              tb-snd-do(KERNEL, tbterm(action))
rec(function >> kernel, quit)  →
              tb-rec-msg(function, kernel, tbterm(quit)) .
              tb-snd-do(KERNEL, tbterm(quit))
snd-quit   → snd-tb-shutdown
rec(function >> kernel, process-status)→
```

```
                tb-rec-msg(function, kernel,
                   tbterm(process-status)) .
                tb-snd-eval(KERNEL, tbterm(process-status)) .
                tb-rec-value(KERNEL, tbterm(process-status))
rec(startprocess >> kernel, start-process)→
                tb-rec-msg(startprocess, kernel,
                   tbterm(start-process)) .
                tb-snd-do(KERNEL, tbterm(start-process))
```

StartProcess

```
select-start-process →
                tb-rec-event(STARTPROCESS, tbterm(start-process)) .
                tb-snd-ack-event(STARTPROCESS,
                   tbterm(start-process))
```

ActionChooser

```
force-random-off   →
                tb-snd-do(ACTIONCHOOSER, tbterm(random-off))
present-list  →
                tb-snd-do(ACTIONCHOOSER,
                   tbterm(action-choose-list))
choose-action→
                tb-rec-event(ACTIONCHOOSER, tbterm(action)) .
                tb-snd-ack-event(ACTIONCHOOSER, tbterm(action))
rec(kernel >> actionchooser, reset)→
                tb-rec-msg(kernel, actionchooser, tbterm(reset)) .
                tb-snd-do(ACTIONCHOOSER, tbterm(reset))
random-off → tb-rec-event(ACTIONCHOOSER, tbterm(random-off)) .
                tb-snd-ack-event(ACTIONCHOOSER, tbterm(random-off))
random-on  → tb-rec-event(ACTIONCHOOSER, tbterm(random-on)) .
                tb-snd-ack-event(ACTIONCHOOSER, tbterm(random-on))
```

Function

```
push-quit  → tb-rec-event(FUNCTION, tbterm(quit)) .
                tb-snd-ack-event(FUNCTION, tbterm(quit))
push-process-status   →
                tb-rec-event(FUNCTION, tbterm(process-status)) .
                tb-snd-ack-event(FUNCTION, tbterm(process-status))
```

TraceCtrl

```
rec(actionchooser >> tracectrl, action)→
                tb-rec-msg(actionchooser, tracectrl,
                   tbterm(action)) .
                tb-snd-eval(TRACECTRL, tbterm(action))
trace      → tb-rec-value(TRACECTRL, tbterm(trace))
no-trace   → tb-rec-value(TRACECTRL, tbterm(no-trace))
```

BreakCtrl

```
rec(actionchooser >> breakctrl, action)→
                tb-rec-msg(actionchooser, breakctrl,
                   tbterm(action)) .
                tb-snd-eval(BREAKCTRL, tbterm(action))
break      → tb-rec-value(BREAKCTRL, tbterm(break))
```

```
no-break   → tb-rec-value(BREAKCTRL, tbterm(no-break))
rec(actionchooser >> breakctrl, action-choose-list)→
             tb-rec-msg(actionchooser, breakctrl,
                tbterm(action-choose-list)) .
             tb-snd-eval(BREAKCTRL, tbterm(action-choose-list))
break-list → tb-rec-value(BREAKCTRL, tbterm(break))
no-break-list →
             tb-rec-value(BREAKCTRL, tbterm(action-choose-list))
```

Display

```
rec($1 >> display, $2)→
             tb-rec-msg($1, display, tbterm($2)) .
             tb-snd-do(DISPLAY, tbterm($2))
```

We rename all component modules and their main processes by putting a `P` in front of the name, indicating a Process in the ToolBus, to distinguish them from the tools for which we use a `T` in front of the name and possible adapters for which we use an `A`.

10.3.2 Constraining

We constrain the ToolBus processes obtained in the previous section with the specification of the tools. We confine ourselves to the constraining of the process `PKernel`, since the constraining of the other processes is rather straightforward and later we shall refine the Kernel even further. We show the module for the Kernel below. Here the main process `PT-Kernel` is the parallel composition of `PKernel` with the constraining process `TKernel`.

```
process module PKernel
begin
   exports
   begin
      processes
         PT-Kernel
   end
   imports
      SimulatorData,
      ToolBusPrimitives,
      ToolFunctions,
      TKernel,
      Booleans
   processes
      PKernel
      Kernel : BOOLEAN
   variables
      wait : → BOOLEAN
   definitions
      PT-Kernel = PKernel ‖ TKernel
      PKernel = Kernel(true)
      Kernel(wait) =
         (
           [wait = false] → (
              tb-snd-eval(KERNEL, tbterm(compute-choose-list)) .
              (
                 tb-rec-value(KERNEL,
                    tbterm(action-choose-list)) .
                 tb-snd-msg(kernel, actionchooser,
```

```
                        tbterm(action-choose-list))
                  +  tb-rec-value(KERNEL, tbterm(halt)) .
                     tb-snd-msg(kernel, display, tbterm(halt))
                  )
               ) .
               Kernel(true)
            +  [wait = true] → (
                  tb-rec-msg(actionchooser, kernel, tbterm(action)) .
                  tb-snd-do(KERNEL, tbterm(action)) .
                  Kernel(false)
               +  tb-rec-msg(function, kernel, tbterm(quit)) .
                  tb-snd-do(KERNEL, tbterm(quit)) .
                  snd-tb-shutdown
               +  tb-rec-msg(function, kernel,
                     tbterm(process-status)) .
                  tb-snd-eval(KERNEL, tbterm(process-status)) .
                  tb-rec-value(KERNEL, tbterm(process-status)) .
                  tb-snd-msg(kernel, display,
                     tbterm(process-status)) .
                  Kernel(true)
               +  tb-rec-msg(startprocess, kernel,
                     tbterm(start-process)) .
                  tb-snd-do(KERNEL, tbterm(start-process)) .
                  tb-snd-msg(kernel, display,
                     tbterm(start-process)) .
                  tb-snd-msg(kernel, actionchooser, tbterm(reset)) .
                  Kernel(false)
               )
            )
end PKernel
```

Where the tool `TKernel` is specified as follows.

```
process module TKernel
begin
   exports
   begin
      processes
         TKernel
   end
   imports
      SimulatorData,
      ToolToolBusPrimitives,
      ToolFunctions
   atoms
      action-choose-list
      halt
   definitions
      TKernel =
            tooltb-rec(tbterm(compute-choose-list)) .
            (
               action-choose-list .
               tooltb-snd(tbterm(action-choose-list))
            +  halt .
               tooltb-snd(tbterm(halt))
            ) . TKernel
         +  tooltb-rec(tbterm(action)) .
            TKernel
         +  tooltb-rec(tbterm(process-status)) .
            tooltb-snd(tbterm(process-status)) .
            TKernel
```

```
        + tooltb-rec(tbterm(start-process)) .
          TKernel
        + tooltb-rec(tbterm(quit))
end TKernel
```

10.3.3 The ToolBus Application

We show how the processes for the tools are imported and put in parallel in the module SimulatorSystem.

```
process module SimulatorSystem
begin
  exports
  begin
    processes
      SimulatorSystem
  end
  imports
        .
        .
    NewTool {
      Tool bound by [
        Tool → PT-Kernel
      ] to PKernel
      renamed by [
        TBProcess → Kernel
      ]
    },
        .
        .
  definitions
    SimulatorSystem =
        Kernel
      ‖ StartProcess
      ‖ ActionChooser
      ‖ Function
      ‖ TraceCtrl
      ‖ BreakCtrl
      ‖ Display
end SimulatorSystem
```

And finally we put this in the ToolBus environment.

```
process module Simulator
begin
  imports
    NewToolBus {
      Application bound by [
        Application → SimulatorSystem
      ] to SimulatorSystem
      renamed by [
        ToolBus → Simulator
      ]
    }
end Simulator
```

10.3.4 Further Specification of the Kernel Tool

We want to split the Kernel tool into a separate adapter and tool, so that a final implementation of the kernel can be used in other applications. We do this again by

applying the refining and constraining techniques. We take the specification of the Kernel tool as given in section 10.3.2 and apply the following mapping, where the first rule is a default mapping.

```
tooltb-rec(tbterm($1))  →
              tooltb-rec(tbterm($1)) .
              tooladapter-snd($1)
action-choose-list →
              tooladapter-rec(action-choose-list)
halt       → tooladapter-rec(halt)
tooltb-rec(tbterm(process-status))→
              tooltb-rec(tbterm(process-status)) .
              tooladapter-snd(process-status) .
              tooladapter-rec(process-status)
```

By renaming TKernel into AKernel we obtain the adapter of the Kernel as shown below.

```
process module AKernel
begin
   exports
   begin
      processes
         AKernel
   end
   imports
      SimulatorData,
      ToolAdapterPrimitives,
      ToolToolBusPrimitives,
      ToolFunctions
   definitions
      AKernel =
            tooltb-rec(tbterm(compute-choose-list)) .
            tooladapter-snd(compute-choose-list) .
            (
               tooladapter-rec(action-choose-list) .
               tooltb-snd(tbterm(action-choose-list))
            +  tooladapter-rec(halt) .
               tooltb-snd(tbterm(halt))
            ) . AKernel
         +  tooltb-rec(tbterm(action)) .
            tooladapter-snd(action) .
            AKernel
         +  tooltb-rec(tbterm(process-status)) .
            tooladapter-snd(process-status) .
            tooladapter-rec(process-status) .
            tooltb-snd(tbterm(process-status)) .
            AKernel
         +  tooltb-rec(tbterm(start-process)) .
            tooladapter-snd(start-process) .
            AKernel
         +  tooltb-rec(tbterm(quit)) .
            tooladapter-snd(quit)
end AKernel
```

Now we specify the new Kernel tool.

```
process module TKernel
begin
   exports
   begin
      atoms
```

```
            snd : Tterm
            rec : Tterm
         processes
            TKernel
      end
      imports
         SimulatorData
      definitions
         TKernel =
               rec(compute-choose-list) .
               (
                  snd(action-choose-list)
               +  snd(halt)
               ) . TKernel
            +  rec(action) .
               TKernel
            +  rec(process-status) .
               snd(process-status) .
               TKernel
            +  rec(start-process) .
               TKernel
            +  rec(quit)
end TKernel
```

We constrain the adapter with the tool as follows.

```
process module TA-Kernel
begin
   imports
      NewToolAdapter {
         Tool bound by [
            tool-snd → snd,
            tool-rec → rec,
            Tool → TKernel
         ] to TKernel
         Adapter bound by [
            Adapter → AKernel
         ] to AKernel
         renamed by [
            ToolAdapter → TA-Kernel
         ]
      }
end TA-Kernel
```

And we change in the module PKernel the constraining by TKernel into TA-Kernel. A generated animation of the complete specification of the simulator as ToolBus application is shown in Figure 10-6[6].

10.4 Implementation

The specification of the tools in the ToolBus application specification of the simulator is detailed enough to proceed with the implementation of the simulator. Although the specification of the kernel is far too simple for such a complex tool, it is satisfactory here because we use the old simulator as base for the new implementation.

---

6. Generated with a left to right orientation instead of top to bottom

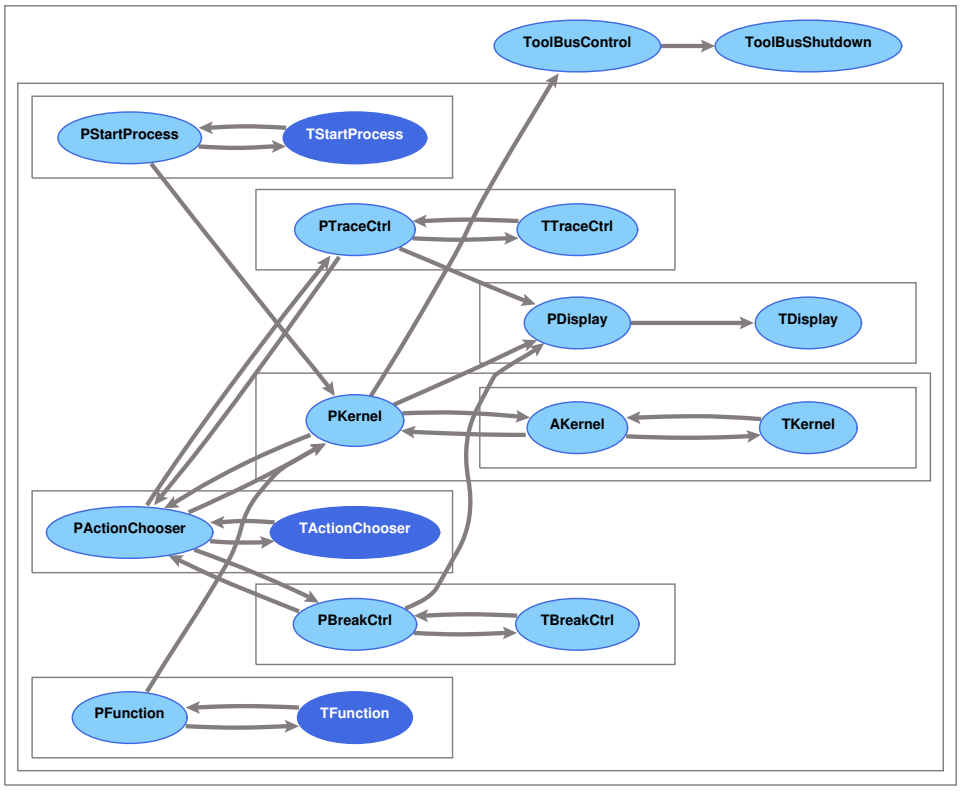


Figure 10-6. System design of the simulator

10.4.1 Kernel

Using the code of the old simulator as base we obtain an implementation of the kernel by doing the following

- remove the graphical user interface
- take out the embedded state machine
- add a component interface for communication with the outside world.

Of course the above three items are strongly related. An event originates from the gui and handling this event can cause a change of state in the state machine.

In the implementation of the kernel an event is received through the component interface. This event is handled and if necessary a reply is sent back through the component interface. The component interface actually is an extension of the interface used in the coupling of the simulator with the animation. The function of the state machine is lifted from the kernel and is now served by the ToolBus.

The adapter of the kernel is implemented in Perl [67] on top of the general Perl adapter

provided with the ToolBus. Perl is chosen because of its powerful regular expression matching and environment interaction.

10.4.2 Other Tools

The other tools are small and simple, and easy to implement. We therefore do not give a further description of their implementation. We have chosen to implement them in Tcl/Tk, mainly because of the ease to build a gui within this language, and its widespread availability.

10.4.3 ToolBus Script

The ToolBus script for controlling the separate tools of the simulator can be derived from the ToolBus processes in the specification of the simulator as ToolBus application. This transformation is done by hand mainly because in the specification recursion is used to hold the state of a process and in a ToolBus script this has to be done with iteration and state variables.

10.4.4 Simulator

To control the execution of the ToolBus we use a Perl script that sets up the environment in which the ToolBus and all the tools that make up our application run. This environment is needed to distribute arguments given on the command line to the different tools.

10.5 Aggregation of GUIs

Except for the kernel, each tool has its own graphical user interface (gui). Because having several windows on the screen belonging to one application does not look very appealing and some of the windows can easily be obscured by windows from other applications, it is better to have a monolithic gui for the simulator. When implementing the gui as a separate tool all communications with this gui and the other tools have to be done over the ToolBus, increasing the complexity of the ToolBus script. These communications interfere with the protocol we specified and validated for the simulator as ToolBus application.

Tcl/Tk, the implementation language we use for the gui's, has a feature to indicate that a frame window is to serve as a container of another application and that a toplevel window is to be used as the child of such a container window. We can use this feature to aggregate the various gui's seemingly working as a monolithic gui, without interfering with the protocol we have specified for the simulator. When doing this, we actually make use of the window manager for the integration of the gui's as pictured in Figure 10-7. For a toplevel to act as a child of a container window, it needs the window id of the parent. So the aggregated gui implementation has to communicate a window id to each child. The ToolBus script must be supplied with an initialization phase that receives all the id's of the container windows from the aggregated gui and distributes them over the tools. Each tool now first receives its parent id before doing anything else. From there on, the ToolBus script operates as we have specified.

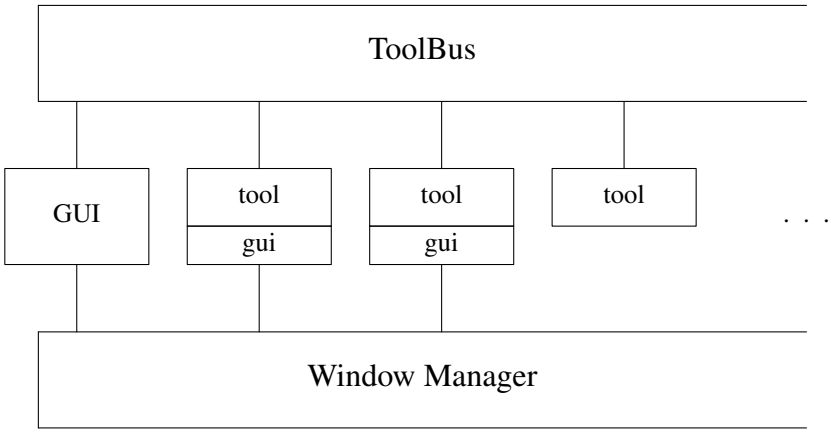


Figure 10-7. Aggregation of gui's and window manager interaction

Following this scheme, we have implemented a separate tool that does the layout of several container windows. This layout can be resized as a whole and some windows can be resized in relation to each other through the use of paned windows.[7] The resulting gui is shown in Figure 10-8. A user preferring a different layout can implement another version similar to this.

The aggregation of gui's can be generalized to a plug-in architecture where a main gui manages the other gui's according to some scheme. Such a plug-in architecture for gui's is used in the implementation of the Meta-Environment (Chapter 6 of [31]).

10.6 Extension with History Mechanism

In this section we describe the extension of the simulator with a history mechanism. The changes that have to be made to all levels of the design process are dealt with. This will show the impact of a software evolution process iteration on our design process.

10.6.1 Architecture Specification

The history actions consist of undo, redo, mark, and goto mark. The logical place for keeping a history is the kernel. We can let the kernel save the current state after every action it has done, but when running randomly this can use a lot of memory and usually with an undo the user wants to jump directly to the state before random mode was started. Since the kernel does not know when the simulator is running randomly, it has to be informed when to save the current state. The action undo, redo, and goto mark, can all be seen as a goto to a certain state. So it suffices to add only a save and goto request to the

7. A paned window consists of two horizontal or vertical panes separated by a movable sash, and each pane containing a window.

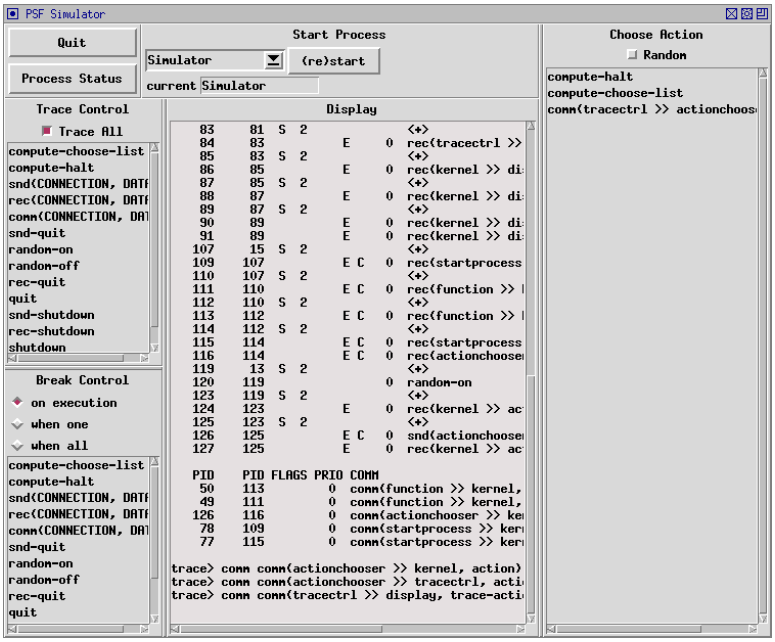


Figure 10-8. Aggregation of gui's

kernel. Below we show the changes for the kernel with existing code in grey.

```
Kernel = Kernel(true)
Kernel(wait) =
   (
      [wait = false] → (
         compute-choose-list .
         (
            action-choose-list .
            snd(kernel >> actionchooser, action-choose-list)
         +  halt .
            snd(kernel >> actionchooser, halt) .
            snd(kernel >> display, halt)
         )
      ) .
      Kernel(true)
   +  [wait = true] → (
         :
         :
      +  rec(actionchooser >> kernel, save) .
         Kernel(true)
      +  rec(actionchooser >> kernel, goto) .
         Kernel(false)
      )
   )
```

Note that we also send a halt to the actionchooser now. Previously, in this case there was nothing to do for the actionchooser, but now a history action can take place.

A history action can be seen as just another action that the user can choose from all possible

actions, so the logical place for such an action is in the actionchooser.

```
ActionChooser = Choose(false, false)
Choose(random, choose) =
      rec(kernel >> actionchooser, action-choose-list) .
         .
         .
   +  rec(kernel >> actionchooser, halt) .
      force-random-off .
      Choose(false, true)
   +  [choose = true] → (
         choose-action .
         snd(actionchooser >> kernel, action) .
            .
            .
      +  snd(actionchooser >> kernel, save) .
         Choose(random, true)
      +  [random = false] → (
            snd(actionchooser >> kernel, goto) .
            Choose(false, false)
         )
      )
   +  rec(kernel >> actionchooser, reset) .
      Choose(random, false)
   +  [random = true] → (
         random-off .
         Choose(false, choose)
      )
   +  [random = false] → (
         random-on .
         Choose(true, choose)
      )
```

We have to turn off the random mode on a halt so that a history action can be chosen. Note that the actionchooser can do a save also in random mode, what makes other history saving schemes possible, for instance every *n* steps.

10.6.2 ToolBus Application Specification

To obtain a ToolBus Application specification with added history mechanism from the architecture specification, we extend the mapping from section 10.3.1 with the following rules.

module Kernel

```
rec(actionchooser >> kernel, save)→
            tb-rec-msg(actionchooser, kernel, tbterm(save)) .
            tb-snd-do(KERNEL, tbterm(save))
rec(actionchooser >> kernel, goto)→
            tb-rec-msg(actionchooser, kernel, tbterm(goto)) .
            tb-snd-do(KERNEL, tbterm(goto))
```

module ActionChooser

```
save      → tb-rec-event(ACTIONCHOOSER, tbterm(save)) .
            tb-snd-ack-event(ACTIONCHOOSER, tbterm(save))
goto      → tb-rec-event(ACTIONCHOOSER, tbterm(goto)) .
            tb-snd-ack-event(ACTIONCHOOSER, tbterm(goto))
```

The adapter and the kernel tool can simply be extended with alternatives for handling a save

and goto as follows.

```
AKernel =
     :
     :
   + tooltb-rec(tbterm(save)) .
     tooladapter-snd(save) .
     AKernel
   + tooltb-rec(tbterm(goto)) .
     tooladapter-snd(goto) .
     AKernel

TKernel =
     :
     :
   + rec(save) .
     TKernel
   + rec(goto) .
     TKernel
```

The adaptation of the actionchooser tool is slightly more complicated because we have to distinguishing the cases when there is a list of actions available to choose from and when there is not.

```
TActionChooser = Choose(false)
Choose(random) =
     tooltb-rec(tbterm(action-choose-list)) .
     (
        [random = true] → (
           :
           :
        )
     + [random = false] → (
          tooltb-snd-event(tbterm(save)) .
          tooltb-rec-ack-event(tbterm(save)) .
          (
             tooltb-snd-event(tbterm(random-on)) .
             tooltb-rec-ack-event(tbterm(random-on)) .
             tooltb-snd-event(tbterm(action)) .
             tooltb-rec-ack-event(tbterm(action)) .
             Choose(true)
          + tooltb-snd-event(tbterm(action)) .
             tooltb-rec-ack-event(tbterm(action)) .
             Choose(random)
          + tooltb-rec(tbterm(reset)) .
             Choose(random)
          + History
          )
        )
     )
   + [random = false] → (
        :
        :
     )
   + [random = true] → (
        :
        :
     )
   + tooltb-rec(tbterm(reset)) .
     Choose(random)
   + [random = false] → (
        History
```

```
        )
History =
        tooltb-snd-event(tbterm(goto)) .
        tooltb-rec-ack-event(tbterm(goto)) .
        Choose(false)
```

The actionchooser tool only does a save when random mode is off, and so constrains the ToolBus process.

10.6.3 Implementation

In order to distinguish the different saves of history we need a unique id for every save. Then a goto sent by the actionchooser can be supplied with an id so that the kernel can jump to the right saved history.

The actionchooser needs to generate these id's. We have implemented the id's as natural numbers and use ordering for easy lookups by the kernel. A mark of a saved history is done in the actionchooser by pairing this mark with the id of that save.

The history mechanism in the kernel is based on the history mechanism of the old simulator with only a few adaptations since some functionality is taken over by the actionchooser.

The gui of the history mechanism is implemented as a separate part of the actionchooser as shown in Figure 10-9.

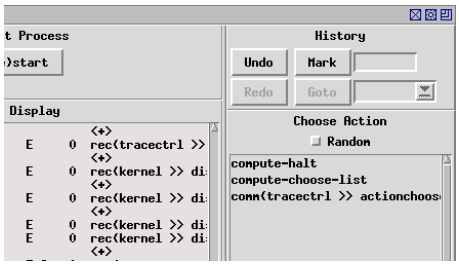


Figure 10-9. Aggregation of gui's with history

10.7 Coupling to Animation

The implementation of the old simulator coupled to the animation was done through the ToolBus as described in [19]. With that implementation the user could switch between choosing actions through the animation or from a list of actions. For our new implementation we have a choice from three possibilities:

- replacement of the actionchooser with the animation,
- use of two choosers controlled by the ToolBus,
- combination of the two choosers in one tool.

We choose to combine the two choosers, because both are implemented in Tcl/Tk and therefore the animation can be implemented as a toplevel window in the actionchooser with

easy control of both choosers, and without change in the graphical user interface. With this choice there is no need for adaptation of the architecture either.

10.8 Features Not Implemented

Here we mention the features of the old simulator that are not implemented by the new simulator because they are seldom used. We give some indications on how these features can be implemented.

weighted random
: Normally all actions have equal chance to be picked randomly. With weighted random the position of an action in the process tree is taken into account. For instance, an action in parallel with a process that spans many actions gets a very low chance to be chosen because of all these actions, but with weighted random its chance stays the same, and the actions of the parallel process get a combined weight equal to the weight of that one action.
This can easily be implemented by letting the kernel send weights with each action in the action-choose-list.

(re)load specification
: Because of the very short start up time of the old simulator, this feature is seldom used. The start up time of the new simulator does not differ much.
It can be implemented by letting the kernel do a clean up and start with a new specification or by shooting off the kernel and starting a new one.

trace to standard output / from standard input
: With trace to standard output every step of the simulator can be recorded and played back with trace from standard input. This can be used for demo's or for testing starting at a certain point every time, which also can be done with a mark on a saved history. Although these features are seldom used, they can be very convenient. Especially trace from standard input, because with that we can build applications with a stateless kernel for not too large simulations where a complete trace is fed to the kernel everytime together with a new action, such as a demo on the world wide web.
Trace to standard output can be implemented by embedding a monitor in the ToolBus that records all necessary actions, and trace from standard input can be put in place of the actionchooser.

10.9 Comparison of Implementations

In Table 10-1 we show the lines of code for the two implementations. The new implementation takes considerably less lines of code mainly because Tcl/Tk and Perl code as TB scripts are very expressive, but it is also caused by the reduction of the complexity of the code. The left out features also play a role here but not by a large amount.

The new implementation should be easier to maintain because of the reduction in lines of code and complexity, although it requires the knowledge of several more implementation languages. The specifications of the architecture and the simulator as ToolBus application

Table 10-1. Lines of code for the implementations

language	lines of code	
	old	new
C	21076	13884
Tcl/Tk		1550
Perl		179
ToolBus script		281
total	21076	15894

play an important role here, since they can be used not only to get familiar with the design but also for testing changes and new features.

The graphical user interface has improved a lot, but it can also easily be altered. It should not be difficult to make an implementation that can be customized according to the preferences of each user.

The division in components has made reuse of parts of the implementation far more easier. It can even be used as a framework for simulation of other languages similar to PSF or new versions of PSF by only providing a different kernel.

The trade-off is that the new implementation is considerably slower, about a factor of thirty. This is due to the fact that this implementation consists of many processes running at the same time and the context switching together with the inter-process communications take up a lot of time. For working interactively this is not a problem, but for large random simulations, for instance validation testing, it is too slow. This can be partly solved by running the new simulator on multiple cpu's, lowering the number of context switches considerably.

Chapter 11

An IDE for PSF

In Chapters 8 and 10 we gained experience on software (re-)engineering with the use of PSF for existing tools. It is possible that our knowledge of the existing tools has played a role in the (re-)engineering process. To find out if we are able to develop new tools without problems with the use of the PSF libraries for architecture and ToolBus application together with the refinement techniques, we develop an integrated development environment (IDE) for PSF. The main purpose for the IDE is to provide easy access to tools in the PSF Toolkit for users not familiar with the Toolkit, and users preferring an IDE to the command-line interface (CLI).

We use the same development process for the IDE as that described in Chapter 10 for the new implementation of the simulator. For the development of the architecture specification we use scenario's. We start with an architecture specification for a simple scenario and adapt this specification to incorporate other scenario's in a stepwise manner.

In section 11.1 we describe the requirements for the IDE. We develop an architecture specification for the IDE in section 11.2 and we refine this architecture specification into a ToolBus application specification in section 11.3 In section 11.4 we describe the implementation for the IDE.

11.1 Requirements for the IDE

Every user of the PSF Toolkit has his or her own preferences concerning the way the tools are applied. Some prefer to be in full control and use a command line approach, or automate the execution of the tools with script or make-like facilities. Others prefer the integration of the tools into one larger tool which automates the execution and provides control through a graphical user interface. The purpose of the IDE for PSF is to support the last group.

Functional Requirements

- Integration of editing facilities, compiler, simulator, and animation generator.
- Simple interaction with the user through a consistent graphical user interface.
- Providing clear information on the status of the development process.
- Hiding of the interaction between the tools.

Non-functional Requirements

- Modular design with easy to replace components.
- Use of existing tools in the PSF Toolkit. Any modifications necessary for the interaction between the tools should be as small as possible and should not alter the command-line interface of the tools.
- Extendible with other tools.

11.2 Architecture Specification of the IDE

We specify software architecture in PSF with the use of a PSF library providing architecture primitives. The primitives are `snd` and `rec` actions for communication, each taking a connection and a data term as argument. A connection can be build up with a connection function `>>` with two identifiers as arguments, each indicating a component. Processes describing the software architecture with these primitives can be set in an architecture environment, also provided by the PSF library. The architecture environment takes care of encapsulation to enforce the communication between the processes.

To develop a specification of the architecture for the IDE, we start with a simple scenario and try to specify an architecture for just this scenario. We adapt the specification step by step to incorporate other scenario's.

11.2.1 Scenario: one module specification

In this scenario our specification consist of only one module. The module can be edited and compiled until the IDE is stopped.

We need four components, a component for functions on the specification, an editor, a compiler, and a viewer for possible errors from compilation of the specification. We first specify the component identifiers and the data we use in our architecture.

```
data module IDEData
begin
    exports
    begin
        functions
            function : → ID
            editor : → ID
            compiler : → ID
            errorviewer : → ID
```

```
        edit-module : → DATA
        close-module : → DATA
        module-closed : → DATA
        module-written : → DATA
        compile : → DATA
        errors : → DATA
        no-errors : → DATA
    end
    imports
        ArchitectureTypes
end IDEData
```

The functions that can be invoked are to edit, close, and compile the module, and to quit the IDE. We specify the behaviour of this component as follows

```
process module Function
begin
    exports
    begin
        processes
            Function
    end
    imports
        IDEData,
        ArchitecturePrimitives
    atoms
        edit-module
        close-module
        compile
        push-quit
    definitions
        Function =
            (
                edit-module .
                snd(function >> editor, edit-module)
            +   close-module .
                snd(function >> editor, close-module)
            +   rec(editor >> function, module-closed)
            +   rec(editor >> function, module-written)
            +   compile .
                snd(function >> compiler, compile) . (
                    rec(compiler >> function, errors)
                +   rec(compiler >> function, no-errors)
                )
            ) *
            push-quit .
            snd-quit
end Function
```

The editor can receive requests for starting and closing an editing session, and has user actions for writing and closing the module. We specify the Editor component as below. In the remainder we restrict ourselves to show the process definition only and leaving out other sections and the modular structure unless we think it is necessary for better understanding of the specification.

```
Editor =
    rec(function >> editor, edit-module) .
    start-editor .
    Edit
Edit =
```

```
      rec(function >> editor, close-module) .
      close-editor .
      Editor
+     editor-close .
      snd(editor >> function, module-closed) .
      Editor
+     editor-write .
      snd(editor >> function, module-written) .
      Edit
```

The Compiler component reports either successful compilation or unsuccessful compilation and with the latter also reports the errors encountered.

```
Compiler =
      rec(function >> compiler, compile) . (
         snd(compiler >> function, errors) .
         snd(compiler >> errorviewer, errors)
      +  snd(compiler >> function, no-errors)
      ) * delta
```

The ErrorViewer component just displays the errors from compilation.

```
ErrorViewer =
      (
         rec(compiler >> errorviewer, errors)
      ) * delta
```

We specify the system consisting of the components in parallel as follows.

```
process module IDESystem
begin
   exports
   begin
      processes
         IDESystem
   end
   imports
      Function,
      Editor,
      Compiler,
      ErrorViewer
   definitions
      IDESystem =
            Function
         ||  Editor
         ||  Compiler
         ||  ErrorViewer
end IDESystem
```

We put this system in the architecture environment by means of binding the main process to the System parameter of the Architecture module from the Architecture library.

```
process module IDE
begin
   imports
      Architecture {
         System bound by [
            System → IDESystem
         ] to IDESystem
         renamed by [
            Architecture → IDE
         ]
```

```
        }
end IDE
```

The generated animation of this system is shown in Figure 11-1.

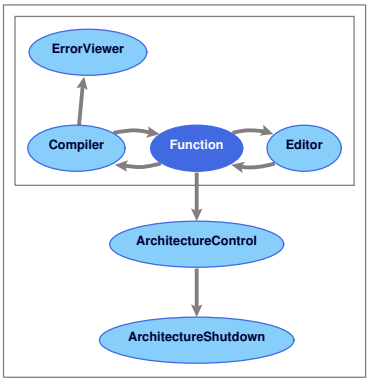


Figure 11-1. Animation of architecture for single module specifications

11.2.2 Scenario: multiple module specification

In the next scenario, we deal with a specification consisting of more than one module. To manage the modules we split the Function component into a new Function component with only a quit action and a module manager.

```
Function =
   push-quit .
   snd(function >> module-manager, quit)
```

The quit action is sent to the module manager, which can decide on what actions to take before the actual quitting of the system.

```
ModuleManager =
   (
      edit-module .
      (
         EventsEditorManager *
         snd(module-manager >> editor-manager, edit-module)
      )
   +  close-module .
      (
         EventsEditorManager *
         snd(module-manager >> editor-manager, close-module)
      )
   +  EventsEditorManager
   +  compile .
      snd(module-manager >> compiler, compile) . (
         rec(compiler >> module-manager, errors)
      +  rec(compiler >> module-manager, no-errors)
      )
   +  rec(function >> module-manager, quit) .
      snd-quit
   ) * delta
```

```
EventsEditorManager =
      rec(editor-manager >> module-manager, module-closed)
   +  rec(editor-manager >> module-manager, module-written)
```

We use the construction with the process EventsEditorManager to prevent deadlocks, which otherwise can occur when the module manager and the editor manager want to send messages to each other at the same time.

We also introduce an editor manager for dealing with multiple editors.

```
EditorManager =
   (
      rec(module-manager >> editor-manager, edit-module) .
      start-editor
   +  editor-close .
      snd(editor-manager >> module-manager, module-closed)
   +  editor-write .
      snd(editor-manager >> module-manager, module-written)
   +  rec(module-manager >> editor-manager, close-module) .
      close-editor
   ) * delta
```

At any time, the module manager can request to start or to close an editor for a module. An editor is expected to report on a closure and on writing of the module. We do not keep track on how many editors there are open at a certain moment, that is up to the implementation of the editor manager.

Our changes of the architecture to accommodate the scenario result in the generated animation as shown in Figure 11-2.

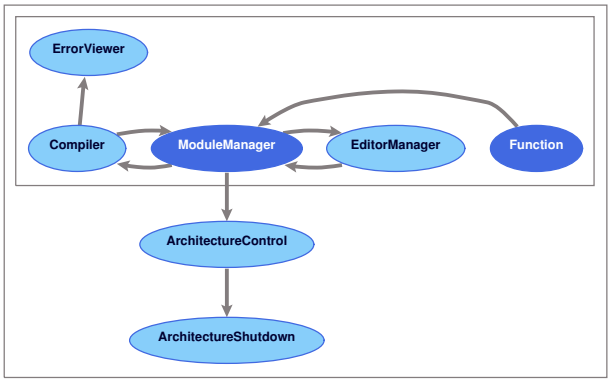


Figure 11-2. Animation of architecture for multi module specifications

11.2.3 Scenario: partial compilation

With the previous scenario, we compile the specification as a whole on every compile request. The PSF compiler however, only applies its steps (parsing, normalizing, flattening) if necessary for a module, based on the time stamps of the PSF module and the intermediate files. We want to make it possible for the module manager to issue parse, compile, and flatten requests whenever it wants. At the same time, we not only want the compiler to

respond to the request of the module manager, but also to act on its own. This scheme will be restricted by the implementation of the module manager and compiler (see section 11.3.2, page 149).

We alter the module manager and the compiler components to provide the described behaviour.

```
ModuleManager =
      :
      :
   +  parse .
      (
         EventsCompiler *
         snd(module-manager >> compiler, parse)
      )
   +  compile .
      (
         EventsCompiler *
         snd(module-manager >> compiler, compile)
      )
   +  flatten .
      (
         EventsCompiler *
         snd(module-manager >> compiler, flatten)
      )
   +  EventsCompiler
      :
      :
```

We replaced the previous compile action with separate parse, compile, and flatten actions. In the same manner as with events from the editor manager, we make a construction for dealing with events from the compiler in order to prevent deadlocks.

```
EventsCompiler =
      rec(compiler >> module-manager, parse-ok)
   +  rec(compiler >> module-manager, parse-uptodate)
   +  rec(compiler >> module-manager, parse-error)
   +  rec(compiler >> module-manager, compile-ok)
   +  rec(compiler >> module-manager, compile-uptodate)
   +  rec(compiler >> module-manager, compile-error)
   +  rec(compiler >> module-manager, flatten-ok)
   +  rec(compiler >> module-manager, flatten-uptodate)
   +  rec(compiler >> module-manager, flatten-error)
```

The compiler receives parse, compile, and flatten requests from the module manager and sends results of parse, compile, and flatten action to the module manager.

```
Compiler =
   (
      rec(module-manager >> compiler, parse)
   +  parse-ok .
      snd(compiler >> module-manager, parse-ok)
   +  parse-uptodate .
      snd(compiler >> module-manager, parse-uptodate)
   +  parse-error .
      snd(compiler >> module-manager, parse-error) .
      snd(compiler >> errorviewer, errors)
   +  rec(module-manager >> compiler, compile)
   +  compile-ok .
      snd(compiler >> module-manager, compile-ok)
```

```
   + compile-uptodate .
     snd(compiler >> module-manager, compile-uptodate)
   + compile-error .
     snd(compiler >> module-manager, compile-error) .
     snd(compiler >> errorviewer, errors)
   + rec(module-manager >> compiler, flatten)
   + flatten-ok .
     snd(compiler >> module-manager, flatten-ok)
   + flatten-uptodate .
     snd(compiler >> module-manager, flatten-uptodate)
   + flatten-error .
     snd(compiler >> module-manager, flatten-error) .
     snd(compiler >> errorviewer, errors)
   ) * delta
```

As mentioned above, we want the compiler to honour the requests of the module manager and also the possibility to act on its own. Therefore, we did not specify a relation between a request and sending the result of an action here.

11.2.4 Scenario: import modules from a library

Some of the modules which are imported may come from a library. We introduce a library manager for adding, deleting and re-ordering a list of libraries. Every time a change of this list occurs the list has to be sent to the compiler, so that it knows where to look for imported modules.

We specify the library manager as follows.

```
LibraryManager =
   (
      set-libraries .
      snd(library-manager >> compiler, set-libraries)
   ) * delta
```

To the specification of the compiler we add the following alternative.

```
   + rec(library-manager >> compiler, set-libraries)
```

11.2.5 Scenario: simulation

Next to compilation, an IDE should also support tools to act on the compiled specification, such as a simulator. These tools can act on a TIL (Tool Interface Language) specification. A TIL specification is the result of compilation of a PSF specification.

To the module manager we add the alternatives to send new and delete notifications on TIL specifications.

```
   + new-tilspecification .
     snd(module-manager >> simulator, new-tilspecification)
   + delete-tilspecification .
     snd(module-manager >> simulator, delete-tilspecification)
```

We add a simulator component to our specification.

```
Simulator =
   (
      rec(module-manager >> simulator, new-tilspecification)
```

```
+  rec(module-manager >> simulator, delete-tilspecification)
) * delta
```

The actual running of a simulator is an internal action of this component that we will specify on a lower level of our design.

The resulting animation is shown in Figure 11-3[8].

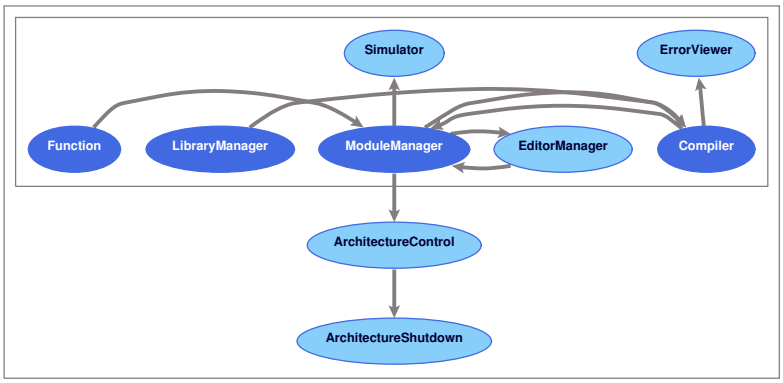


Figure 11-3. Animation of architecture with simulator

11.2.6 Scenario: simulation and animation

Simulation can be run together with an animation. Such an animation can be generated from a TIL specification by the animation generator of the PSF Toolkit. To use the animation generator in the IDE, the module manager also has to send new and delete notifications to a animation generator component. We specify the animation generator as follows.

```
AnimationGenerator =
   (
      rec(module-manager >> animation-generator,
         new-tilspecification)
   +  rec(module-manager >> animation-generator,
         delete-tilspecification)
   +  new-animation .
      snd(animation-generator >> simulator, new-animation)
   +  animationgeneration-error .
      snd(animation-generator >> simulator,
         animationgeneration-error) .
      snd(animation-generator >> errorviewer, errors)
   ) * delta
```

We also add alternatives to the simulator for receiving the notifications from the animation generator.

8. The layout of the components in the animation is generated by the program dot, which is part of the graph visualization software package Graphviz [22]. It is not always the best layout possible.

11.3 System Specification of the IDE

We derive a specification of the IDE as a ToolBus application from the specification of the architecture of the IDE by applying the implementation techniques *action refinement* and *process constraining*.

11.3.1 Action Refinement

We show the refinements for a vertical implementation of the architecture specification. We start with some default mappings, which are to be applied when there are no other mappings to apply.

```
snd($1 >> $2, $3)   → tb-snd-msg($1, $2, tbterm($3))
rec($1 >> $2, $3)   → tb-rec-msg($1, $2, tbterm($3))
```

The $\$n$ on the left hand side represent matched terms that have to be filled in on the right hand side. Below the mappings per module (component) are given.

Function

```
push-quit  → tb-rec-event(MODULEMANAGER, tbterm(quit)) .
             tb-snd-ack-event(MODULEMANAGER, tbterm(quit))
```

ModuleManager

```
edit-module  →
             tb-rec-event(MODULEMANAGER, tbterm(edit-module)) .
             tb-snd-ack-event(MODULEMANAGER,
                  tbterm(edit-module))
close-module →
             tb-rec-event(MODULEMANAGER, tbterm(close-module)) .
             tb-snd-ack-event(MODULEMANAGER,
                  tbterm(close-module))
parse      → tb-rec-event(MODULEMANAGER, tbterm(parse)) .
             tb-snd-ack-event(MODULEMANAGER, tbterm(parse))
compile    → tb-rec-event(MODULEMANAGER, tbterm(compile)) .
             tb-snd-ack-event(MODULEMANAGER, tbterm(compile))
flatten    → tb-rec-event(MODULEMANAGER, tbterm(flatten)) .
             tb-snd-ack-event(MODULEMANAGER, tbterm(flatten))
new-tilspecification    →
             tb-rec-event(MODULEMANAGER,
                  tbterm(new-tilspecification)) .
             tb-snd-ack-event(MODULEMANAGER,
                  tbterm(new-tilspecification))
delete-tilspecification →
             tb-rec-event(MODULEMANAGER,
                  tbterm(delete-tilspecification)) .
             tb-snd-ack-event(MODULEMANAGER,
                  tbterm(delete-tilspecification))
rec(compiler >> module-manager, $1) →
             tb-rec-msg(compiler, module-manager, tbterm($1)) .
             tb-snd-do(MODULEMANAGER, tbterm($1))
rec(editor-manager >> module-manager, $1)     →
             tb-rec-msg(editor-manager, module-manager,
                  tbterm($1)) .
             tb-snd-do(MODULEMANAGER, tbterm($1))
```

```
snd-quit   → snd-tb-shutdown
```

EditorManager

```
start-editor →
              tb-snd-do(EDITORMANAGER, tbterm(start-editor))
editor-close →
              tb-rec-event(EDITORMANAGER, tbterm(editor-close)) .
              tb-snd-ack-event(EDITORMANAGER,
                   tbterm(editor-close))
editor-write →
              tb-rec-event(EDITORMANAGER, tbterm(editor-write)) .
              tb-snd-ack-event(EDITORMANAGER,
                   tbterm(editor-write))
close-editor →
              tb-snd-do(EDITORMANAGER, tbterm(close-editor))
```

Compiler

```
rec(module-manager >> compiler, parse)  →
              tb-rec-msg(module-manager, compiler,
                   tbterm(parse)) .
              tb-snd-eval(COMPILER, tbterm(parse))
parse-ok   → tb-rec-value(COMPILER, tbterm(parse-ok))
parse-uptodate     →
              tb-rec-value(COMPILER, tbterm(parse-uptodate))
parse-error  →
              tb-rec-value(COMPILER, tbterm(parse-error))
rec(module-manager >> compiler, compile) →
              tb-rec-msg(module-manager, compiler,
                   tbterm(compile)) .
              tb-snd-do(COMPILER, tbterm(compile))
compile-ok → tb-rec-event(COMPILER, tbterm(compile-ok)) .
              tb-snd-ack-event(COMPILER, tbterm(compile-ok))
compile-uptodate   →
              tb-rec-event(COMPILER, tbterm(compile-uptodate)) .
              tb-snd-ack-event(COMPILER,
                   tbterm(compile-uptodate))
compile-error →
              tb-rec-event(COMPILER, tbterm(compile-error)) .
              tb-snd-ack-event(COMPILER, tbterm(compile-error))
rec(module-manager >> compiler, flatten) →
              tb-rec-msg(module-manager, compiler,
                   tbterm(flatten)) .
flatten-ok → tb-rec-event(COMPILER, tbterm(flatten-ok)) .
              tb-snd-ack-event(COMPILER, tbterm(flatten-ok))
flatten-uptodate   →
              tb-rec-event(COMPILER, tbterm(flatten-uptodate)) .
              tb-snd-ack-event(COMPILER,
                   tbterm(flatten-uptodate))
flatten-error →
              tb-rec-event(COMPILER, tbterm(flatten-error)) .
              tb-snd-ack-event(COMPILER, tbterm(flatten-error))
```

ErrorViewer

```
rec(compiler >> errorviewer, errors)     →
              tb-rec-msg(compiler, errorviewer, tbterm(errors)) .
              tb-snd-do(ERRORVIEWER, tbterm(errors))
```

```
rec(animation-generator >> errorviewer, errors)     →
              tb-rec-msg(animation-generator, errorviewer,
                   tbterm(errors)) .
              tb-snd-do(ERRORVIEWER, tbterm(errors))
```

LibraryManager

```
set-libraries →
              tb-rec-event(MODULEMANAGER,
                   tbterm(set-libraries)) .
              tb-snd-ack-event(MODULEMANAGER,
                   tbterm(set-libraries))
```

Simulator

```
rec(module-manager >> simulator, new-tilspecification)   →
              tb-rec-msg(module-manager, simulator,
                   tbterm(new-tilspecification)) .
              tb-snd-do(SIMULATOR, tbterm(new-tilspecification))
rec(module-manager >> simulator, delete-tilspecification)     →
              tb-rec-msg(module-manager, simulator,
                   tbterm(delete-tilspecification)) .
              tb-snd-do(SIMULATOR,
                   tbterm(delete-tilspecification))
rec(animation-generator >> simulator, new-animation)     →
              tb-rec-msg(animation-generator, simulator,
                   tbterm(new-animation)) .
              tb-snd-do(SIMULATOR, tbterm(new-animation))
rec(animation-generator >> simulator, animationgeneration-error)
           → tb-rec-msg(animation-generator, simulator,
                   tbterm(animationgeneration-error)) .
              tb-snd-do(SIMULATOR,
                   tbterm(animationgeneration-error))
```

AnimationGenerator

```
rec(module-manager >> animation-generator, new-tilspecification)
           → tb-rec-msg(module-manager, animation-generator,
                   tbterm(new-tilspecification)) .
              tb-snd-do(ANIMATIONGENERATOR,
                   tbterm(new-tilspecification))
rec(module-manager >> animation-generator,
   delete-tilspecification)
           → tb-rec-msg(module-manager, animation-generator,
                   tbterm(delete-tilspecification)) .
              tb-snd-do(ANIMATIONGENERATOR,
                   tbterm(delete-tilspecification))
new-animation →
              tb-rec-event(ANIMATIONGENERATOR,
                   tbterm(new-animation)) .
              tb-snd-ack-event(ANIMATIONGENERATOR,
                   tbterm(new-animation))
animationgeneration-error     →
              tb-rec-event(ANIMATIONGENERATOR,
                   tbterm(animationgeneration-error)) .
              tb-snd-ack-event(ANIMATIONGENERATOR,
                   tbterm(animationgeneration-error))
```

We rename all component modules and their main processes by putting a `P` in front of the

original names, indicating a Process in the ToolBus, to distinguish them from the tools for which we prefix with a `T`. For possible adapters to be used with a tool we use an `A` as prefix.

11.3.2 Constraining

We constrain the ToolBus processes obtained in the previous section with the specification of the tools. The specification of the tools is given in separate modules and each constraining of a ToolBus process is done as shown below for the module manager.

```
process module PT-ModuleManager
begin
   exports
   begin
      processes
         PT-ModuleManager
   end
   imports
      PModuleManager,
      TModuleManager
   definitions
      PT-ModuleManager = PModuleManager || TModuleManager
end PT-ModuleManager
```

In the following we give the specification of tools.

Function

```
TFunction =
   tooltb-snd-event(tbterm(quit)) .
   tooltb-rec-ack-event(tbterm(quit))
```

Module Manager

We decide that the module manager is responsible for the parsing of all modules, and that a compile request for a particular module is also a flatten request for this module. The compiler is responsible for compiling other modules this particular module depends on.

A compilation request for a module results in a series of messages on compilation results of the modules it depends on, ending in an error result or the result of the flattening of the module.

```
TModuleManager =
   (
      new-module
   +  delete-module
   +  tooltb-snd-event(tbterm(edit-module)) .
      tooltb-rec-ack-event(tbterm(edit-module))
   +  tooltb-snd-event(tbterm(close-module)) .
      tooltb-rec-ack-event(tbterm(close-module))
   +  tooltb-rec(tbterm(module-closed))
   +  tooltb-rec(tbterm(module-written))
   +  tooltb-snd-event(tbterm(parse)) .
      tooltb-rec-ack-event(tbterm(parse)) .
      (
         tooltb-rec(tbterm(parse-ok))
```

```
      + tooltb-rec(tbterm(parse-uptodate))
      + tooltb-rec(tbterm(parse-error))
      )
   + tooltb-snd-event(tbterm(compile)) .
      tooltb-rec-ack-event(tbterm(compile)) .
      (
         (
            tooltb-rec(tbterm(compile-ok))
         + tooltb-rec(tbterm(compile-uptodate))
         ) * (
            tooltb-rec(tbterm(compile-error))
         + tooltb-rec(tbterm(flatten-ok))
         + tooltb-rec(tbterm(flatten-uptodate))
         + tooltb-rec(tbterm(flatten-error))
         )
      )
   + tooltb-snd-event(tbterm(new-tilspecification)) .
      tooltb-rec-ack-event(tbterm(new-tilspecification))
   + tooltb-snd-event(tbterm(delete-tilspecification)) .
      tooltb-rec-ack-event(tbterm(delete-tilspecification))
   ) * delta
```

Editor Manager

For the editor manager we use recursion to keep track on the number of open editor sessions. This is necessary for internal actions of the manager to act on open sessions.

```
TEditorManager = TEditorManager(nat(^0))
TEditorManager(n) =
      tooltb-rec(tbterm(start-editor)) .
      TEditorManager(succ(n))
   + [gt(n, nat(^0)) = true] → (
         tooltb-snd-event(tbterm(editor-close)) .
         tooltb-rec-ack-event(tbterm(editor-close)) .
         TEditorManager(pred(n))
      + tooltb-snd-event(tbterm(editor-write)) .
         tooltb-rec-ack-event(tbterm(editor-write)) .
         TEditorManager(n)
      )
   + tooltb-rec(tbterm(close-editor)) .
      TEditorManager(pred(n))
```

Compiler

On a compile request for a particular module the compiler has to compile all the modules this particular module depends on. The compiler sends result messages for the compilation of each module separately. Compilation of modules is stopped immediately on an error result. When there is no compilation error, the module of the request is flattened and a result message of the flattening is send.
Note that we expect separate requests for parsing each module before a compile request. The compilation process sends an error result when it needs a module for which the parsing resulted in an error.

```
TCompiler =
   (
      tooltb-rec(tbterm(compile)) . (
         (
```

```
            compile-ok .
            tooltb-snd-event(tbterm(compile-ok)) .
            tooltb-rec-ack-event(tbterm(compile-ok))
         +  compile-uptodate .
            tooltb-snd-event(tbterm(compile-uptodate)) .
            tooltb-rec-ack-event(tbterm(compile-uptodate))
         ) * (
            compile-error .
            tooltb-snd-event(tbterm(compile-error)) .
            tooltb-rec-ack-event(tbterm(compile-error))
         +  flatten-ok .
            tooltb-snd-event(tbterm(flatten-ok)) .
            tooltb-rec-ack-event(tbterm(flatten-ok))
         +  flatten-uptodate .
            tooltb-snd-event(tbterm(flatten-uptodate)) .
            tooltb-rec-ack-event(tbterm(flatten-uptodate))
         +  flatten-error .
            tooltb-snd-event(tbterm(flatten-error)) .
            tooltb-rec-ack-event(tbterm(flatten-error))
         )
      )
   +  tooltb-rec(tbterm(parse)) . (
         parse-ok .
         tooltb-snd(tbterm(parse-ok))
      +  parse-uptodate .
         tooltb-snd(tbterm(parse-uptodate))
      +  parse-error .
         tooltb-snd(tbterm(parse-error))
      )
   ) * delta
```

Error Viewer

```
TErrorViewer =
   (
      tooltb-rec(tbterm(errors))
   ) * delta
```

Library Manager

```
TLibraryManager =
   (
      tooltb-snd-event(tbterm(set-libraries)) .
      tooltb-rec-ack-event(tbterm(set-libraries))
   ) * delta
```

Simulator

We use recursion to keep track on whether simulation is going on.

```
TSimulator = TSimulator(false)
TSimulator(simulating) =
      tooltb-rec(tbterm(new-tilspecification)) .
      TSimulator(simulating)
   +  tooltb-rec(tbterm(delete-tilspecification)) .
      TSimulator(simulating)
   +  tooltb-rec(tbterm(new-animation)) .
      TSimulator(simulating)
   +  tooltb-rec(tbterm(animationgeneration-error)) .
```

```
      TSimulator(simulating)
   +  [simulating = false] → (
         simulator-start .
         TSimulator(true)
      )
   +  [simulating = true] → (
         simulator-stop .
         TSimulator(false)
      +  simulator-quit .
         TSimulator(false)
      )
```

Animation Generator

```
TAnimationGenerator =
   (
      tooltb-rec(tbterm(new-tilspecification))
   +  tooltb-rec(tbterm(delete-tilspecification))
   +  tooltb-snd-event(tbterm(new-animation)) .
      tooltb-rec-ack-event(tbterm(new-animation))
   +  tooltb-snd-event(tbterm(animationgeneration-error)) .
      tooltb-rec-ack-event(tbterm(animationgeneration-error))
   ) * delta
```

11.3.3 The ToolBus Application

We compose the system by importing the constrained ToolBus processes as instances of the NewTool module from the ToolBus library, and merging them into the system process.

```
process module IDESystem
begin
   exports
   begin
      processes
         IDESystem
   end
   imports
         :
         :
      NewTool {
         Tool bound by [
            Tool → PT-ModuleManager
         ] to PT-ModuleManager
         renamed by [
            TBProcess → ModuleManager
         ]
      },
         :
         :
   definitions
      IDESystem =
            ModuleManager
         ||  EditorManager
         ||  Compiler
         ||  ErrorViewer
         ||  Simulator
         ||  AnimationGenerator
end IDESystem
```

We put the system in the ToolBus application environment by importing the system as

instance of the NewToolBus module from the ToolBus library.

```
process module IDE
begin
   imports
      NewToolBus {
         Application bound by [
            Application → IDESystem
         ] to IDESystem
         renamed by [
            ToolBus → IDE
         ]
      }
end IDE
```

The resulting animation is shown in Figure 11-4.

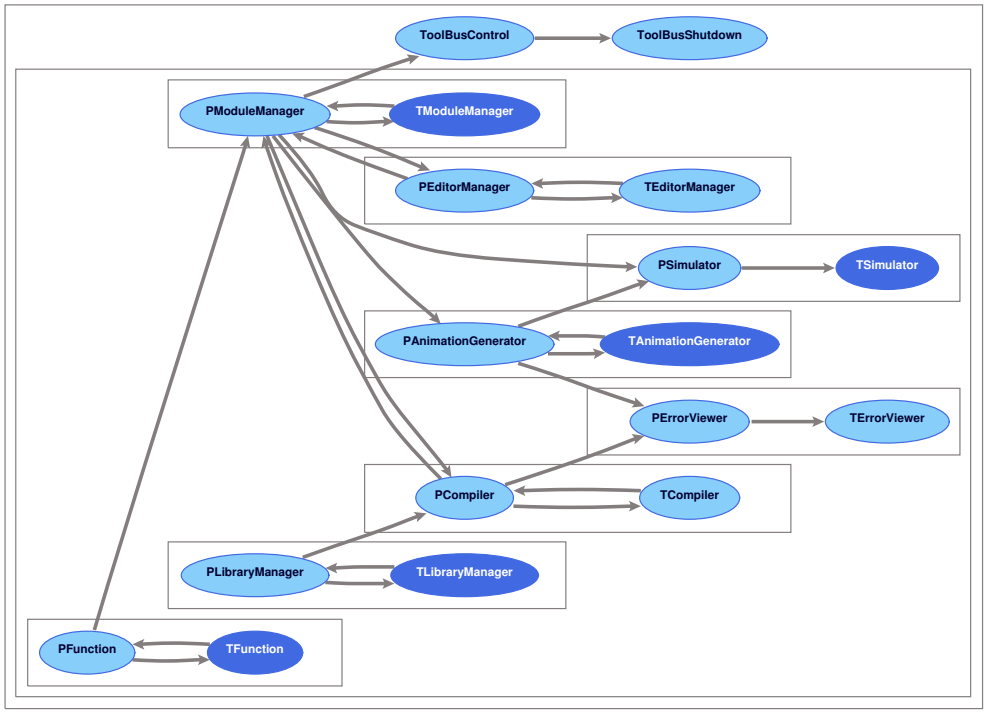


Figure 11-4. Animation of the IDE as ToolBus application

11.4 Implementation of the IDE

In the previous section we gave specifications of the tools which together make up the IDE. Although these specifications are rough, we consider them detailed enough to proceed with the implementation of these tools.

11.4.1 Implementation of the Tools

Function

We implemented the Function component in the language Tcl/Tk [50]. It consists of a single button for requesting to quit the IDE. We added another button to select a type of editor, since the implementation of the editor manager makes a choice of editor possible (see below). We updated the specifications to reflect this possibility.

Module Manager

The module manager controls the operation of the command issued by the user and gives information on the state of the partial compilation of the modules through a table. Tcl/tk is used as implementation language.

A PSF module must have a header and a trailer containing the module-name. This seems superfluous, since a file can only contain one module in the setting of the IDE. However, we have decided not to alter the module structure. Instead, we generate the header and trailer whenever the user requests a new module.

Editor Manager

An editor manager has to execute an editor on request, and to manage open sessions and directing interaction with the editors. Most development environments force an editor upon the user. It is possible that the user is not familiar with this editor and even has to know several editors if working with different development environments. Ideally, the user can choose an editor with which a development environment should interact.

We have chosen to reuse the work of de Jong and Kooiker [32]. They implemented an editor manager which supports interactive editing with the popular editors GNU Emacs [57] and Vim[9] [44]. The manager is implemented as a ToolBus tool and can be integrated in the IDE without any modifications. It is implemented in the C programming language [33].

Compiler

The compiler acts as a controller for the parser, compiler, and flattener from the PSF Toolkit. It keeps track of imports by extracting imported modules from a parsed module. The imports are used to decide on the order of compilation steps of the (intermediate) modules. The compiler is implemented in Perl [67].

In the setting of the IDE, the parser allows only one module per file and the name of the module and file must match. Instead of altering the parser to check on this, we implemented a separate check routine that the compiler invokes prior to the parser.

---

9. Vim is an improved version of vi, an editor distributed with most Unix-like operating systems.

Error Viewer

We implemented the error viewer in Tcl/Tk. It consists of a display and a button to clear the display.

Simulator

The simulator is a wrapper for the simulator from the PSF Toolkit and is implemented in Tcl/Tk.

Animation Generator

The animation generator is a wrapper for the animation from the PSF Toolkit that provides control over the many command-line options. It is implemented in Tcl/Tk.

11.4.2 ToolBus Script

The ToolBus script for controlling the separate tools of the simulator can be derived from the ToolBus processes in the specification of the simulator as ToolBus application. This transformation is done by hand mainly because in the specification recursion is used to hold the state of a process and in a ToolBus script this has to be done with iteration and state variables. Also the data terms have to be refined to contain arguments necessary for identifying the module the message relates to.

11.4.3 Aggregated GUI

Except for the editor manager and compiler, each tool has its own graphical user interface (gui). We aggregate these gui's in the same way we used with the new implementation for the simulator (see section 10.5). Integrating editor sessions in this gui is not a good idea, since we make use of existing editors, of which the gui's do not fit in the gui of the IDE very well. The resulting gui is shown in Figure 11-5.

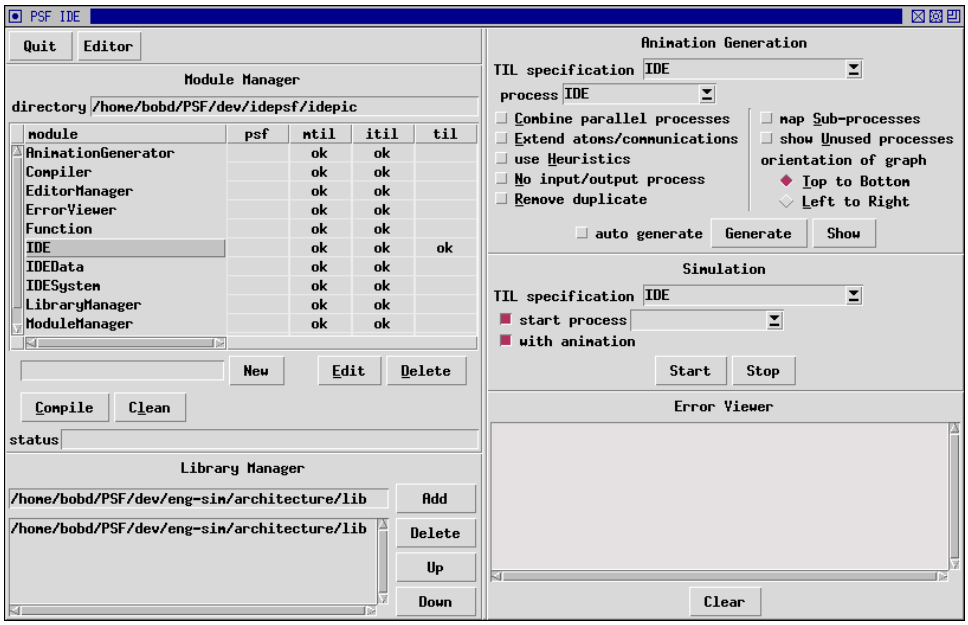


Figure 11-5. Aggregation of gui's

Chapter 12

A Process Algebra Software Engineering Workbench

In the previous chapters we (re-)engineered some tools for the PSF Toolkit. As part of the software development process we specified systems on higher levels of abstraction with the use of PSF and the PSF libraries for Architecture and ToolBus level design. We used the simulator in combination with the animation facilities of the PSF Toolkit for validating the specifications. Furthermore, we presented techniques to refine an architecture specification into a ToolBus specification using horizontal and vertical implementation relations.

In this chapter we describe our software development process more formally by presenting the tools we use in the development process in a Computer-Aided Software Engineering (CASE) setting. Furthermore, we generalize the refine step in the development process so that it can be applied on different levels of design. Several instances of the generalized refine step can be combined to form a software engineering environment.

We start with an introduction on CASE and the terminology we use in section 12.1, followed by the presentation of an environment based on the development of architecture and ToolBus specifications in section 12.2. In the sections 12.3, 12.4, and 12.5 we generalize the refine step in this environment towards a process algebra software engineering workbench that can be used to form software engineering environments. We end with some comments in section 12.6.

12.1 Computer-Aided Software Engineering

Since the early days of developing software, tools are used to assist in the development process. Initially these were the tools provided by the operating system, such as editors, compilers, and debuggers. With the demand for larger software systems, the development process became more complex and expensive, and a need to improve and control the development process arose. One of the technologies to achieve this is computerized

applications supporting and (partially) automating software-production activities. This resulted in many tools for all kinds of activities in the software development process, from editing and testing tools to management and documentation tools.

With the growth in development and use of these tools also the terminology to denote the function and activities of these tools has increased. This terminology is often confusing or misleading. To reason on CASE technology it is necessary to use a fixed terminology. We use the terminology as proposed by Fuggetta in [21] which has since then been used by many others. The following three categories are used for classifying CASE technology.

Tools
: support individual process tasks.

Workbenches
: support process phases or activities and normally consist of a set of tools with some degree of integration.

Environments
: support at least a substantial part of the software process and normally include several integrated workbenches.

12.2 The PSF-ToolBus Software Engineering Environment

Development of a software system starts with the specification of the architecture for the software system. This architecture specification consists of components that communicate with each other. We make use of the PSF Architecture library that provides the primitives we use in the architecture specification. The system consists of the components put together in parallel. The system is then put in an architecture environment. Since the system and the environment are always built in the same way, we can easily generate them. So specification of an architecture is limited to specification of the components. This gives an architecture workbench as shown in Figure 12-1. Objects to be specified are presented as **bold boxes**, workbench tools as ellipses, and generated objects as *slanted boxes*.

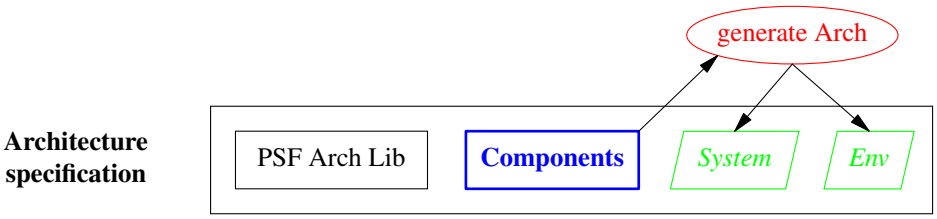


Figure 12-1. The Architecture Workbench

For specification at the ToolBus application level we can make a similar workbench, shown in Figure 12-2. Again we only need to specify the components, now with the use of the PSF ToolBus library, and the system and environment are again generated. To derive a ToolBus application specification from an architecture specification we refine the abstract actions in the architecture specification to sequences of actions on the ToolBus application

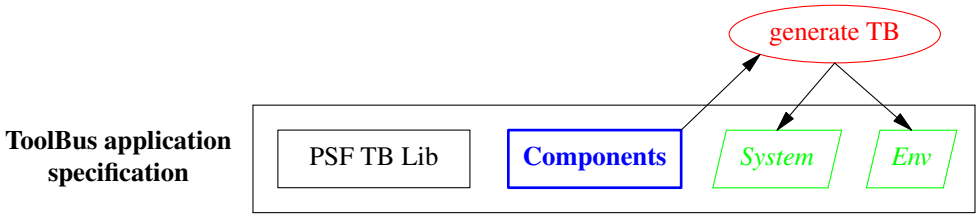


Figure 12-2. The ToolBus Workbench

level. The refinement can be done automatically by applying a set of mappings on the specification of the components, resulting in the specification of a set of ToolBus processes. We constrain these ToolBus processes with (abstract) specifications of the tools. Constraining can also be done automatically and results in the specification of the components for the ToolBus application specification.

We have implemented tools for automatic application of refinements and constraints. Their implementations are ad hoc, based on regular expression matching and a standard layout of the specifications. They definitely need to be improved in order to operate in a more general way.

Combining the workbenches for architecture level and ToolBus level design and integrating the refine and constrain steps, we get the PSF-ToolBus software engineering environment shown in Figure 12-3. We see that the components for the ToolBus application are generated, so we only have to specify the components on the architecture level and give proper mappings and constraints (the tools) to obtain a ToolBus application specification.

Also shown in Figure 12-3 is a generation step of a ToolBus-script from the components of the ToolBus application specification. This step is still to be made by hand since PSF specifications use recursion for setting the state of a process, and the ToolBus cannot handle recursive processes. ToolBus scripts use iteration with variable assignment for keeping track of the state of a process.

12.3 A Generalized PSF Software Engineering Workbench

The Tools in Figure 12-3 can also be ToolBus application specifications developed in a similar way. However, the refining of architecture specifications is not limited to the level of ToolBus application specifications. Other levels of design, and even several levels of refinement connected in series are possible.

We can generalize the refine step in the PSF-ToolBus software engineering environment resulting in the workbench shown in Figure 12-4. The refine and constrain tools are general enough to work on the different levels of the design. The generate Level$_X$ tool and PSF Level$_X$ Library can only be applied at level X.

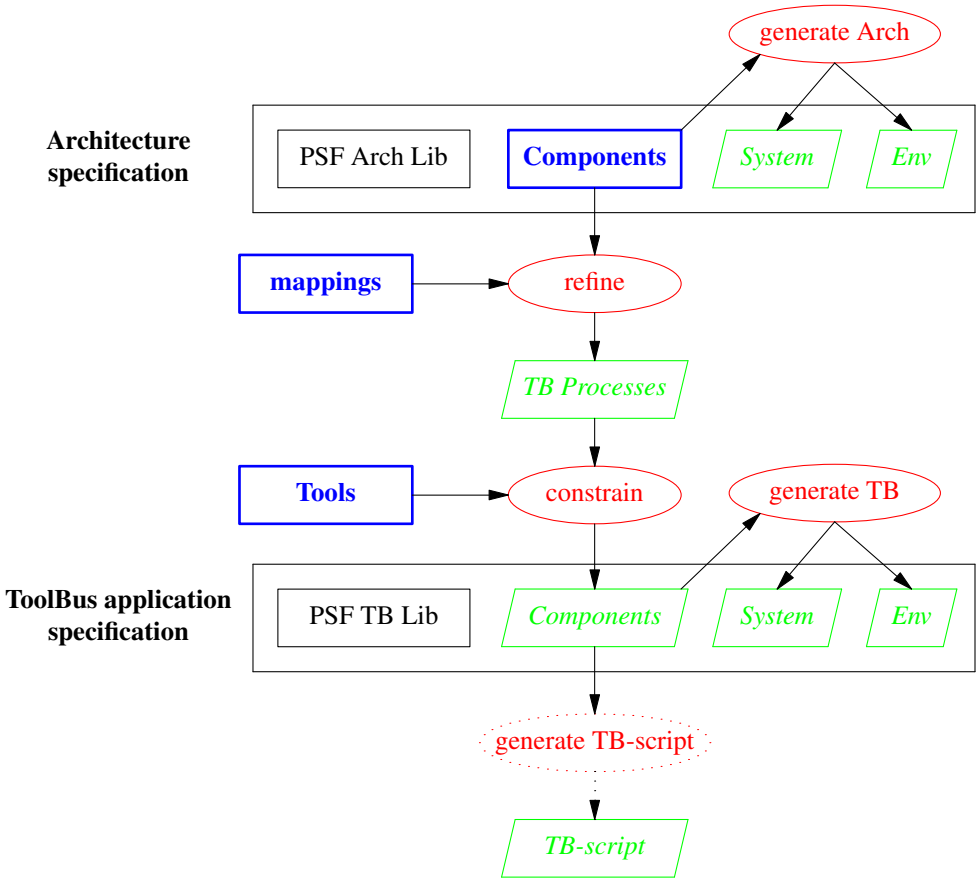


Figure 12-3. The PSF-ToolBus SE Environment

12.4 A Process Algebra Software Engineering Workbench

So far, we used PSF as process algebra language with the workbenches. However, similar workbenches can be set up for variants of PSF or process algebra based languages similar to PSF. By generalizing from PSF we obtain a Process Algebra Software Engineering Workbench.

If this process algebra language can be translated to TIL code, the simulator and animation tools from the PSF Toolkit can be used. If another intermediate language (or the process algebra language itself) is used, then a simulator and animation tool for this intermediate language have to be developed.

When using a different intermediate language, reuse of the simulator from the PSF Toolkit is possible. In the implementation of the simulator we only need to replace the kernel with a kernel for the intermediate language, thus reusing the design of the PSF simulator as

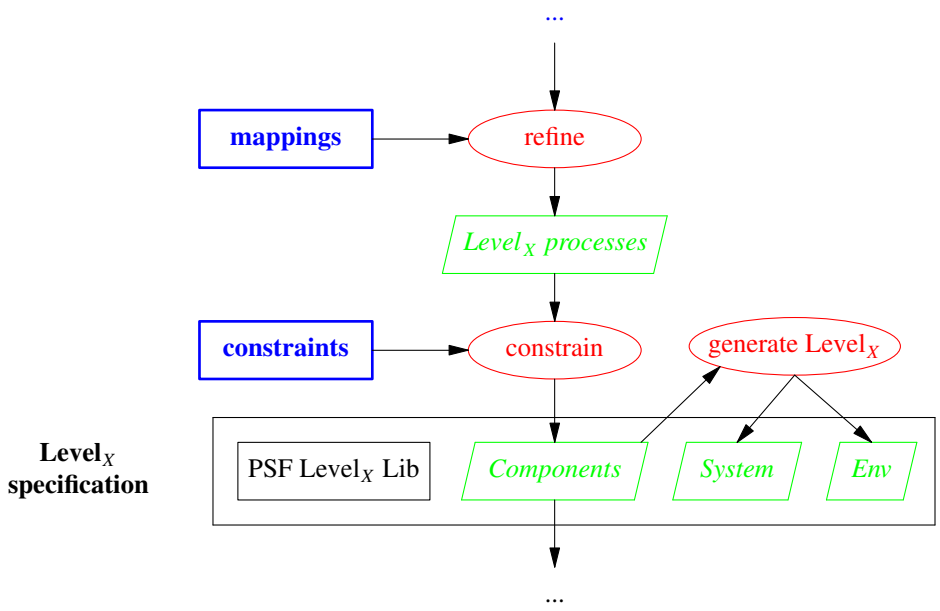


Figure 12-4. The PSF SE Workbench

presented in Chapter 10.

12.5 Forming an Environment

Several instances of the generalized workbench can be combined to form a software engineering environment. The instances can be connected in series. The specifications of the constraining processes can also be developed using instances of the generalized workbench, leading to an environment in which the workbenches are connected in parallel as well as in series.

12.6 Comments

Using several levels for design of software systems has some advantages, of which the most important one is that maintenance becomes easier. Adjustments and changes can be made at an appropriate abstract level of design and be worked down the lower levels. The influence of an adjustment or change on design decisions at the lower levels becomes clear in this process, and can be dealt with at the right level of abstraction. Also, the specifications are part of the documentation of the software system. Simulation of the specifications gives a good understanding of the design of the system, certainly in combination with animations generated from the specifications.

Another advantage is that not only parts of the implementation can be reused for other

systems, but also the design can be reused. This is especially useful in an environment where similar products are being made or incorporated in other products.

Using workbenches in the engineering of software systems increases the advantages mentioned above, since it improves the understanding of the design process. It also gives the opportunity to reason about the design process in a more abstract setting.

A disadvantage is that the design process can be more time consuming than strictly necessary for smaller software systems and for systems that are relatively easy to implement. However, software systems that are used over a long period of time need maintenance and evolve into larger and more complex systems. Re-engineering the design of such systems is far more work than a durable design right from the start.

Chapter 13

Related Work

In this chapter we briefly discuss some of the work that is related to software engineering with process algebra. There is much work done in this area, so we limit the description to work often referred to by others.

13.1 Architecture Description

In the literature several architecture description languages have been proposed, we mention Aesop [24], C2 [41], Darwin [36], MetaH [9], PADL [7], Rapide [35], SADL [45][46], UniCon [55], and Wright [3]. A comparison of several ADLs can be found in [42] (January 2000). Some of the ADLs are based on a process algebra, such as Wright, Darwin, and PADL. A more recent example is π-ADL [47], which is based on the higher-order typed π-calculus. ACME [25] is an architecture interchange language intended to support the mapping of architectural specifications from one ADL to another and, hence, enable integration of support tools across ADLs.

In the comparison from [42] it is argued that an ADL must explicitly model components, connectors, and their configurations. Here, a component is a unit of computation, a connector models interaction among components, and a configuration is the compostion of components and connectors. PSF is not an ADL in that it provides a concrete syntax and a formal, or semi-formal, semantics for specifying software architecture. The PSF Architecture Library however, makes it possible to specify software architecture in a certain style, in which the connectors are specified *in-line*, that is as arguments of the `snd` and `rec` actions in the specification of the components. A more explicit specification of connectors can be established by parameterizing the components and binding these parameters with the actual connectors on merging of the components, making the configuration more explicit as well.

LOTOS [10], a specification language based on (early versions of) the process algebras Calculus of Communicating Systems (CCS) [43] and Communicating Sequential Processes (CSP) [28] and that is comparable to PSF, is used in [27] for the formal description of

architectural styles as LOTOS patterns, and in [54] it is used as an ADL for the specification of middleware behaviour. In [8], an approach for checking deadlock freedom of software architectures, specified in PADL, is described using a mixture of the process algebras CCS and CSP.

13.2 Refinement

Refinement is a key issue in formal development of software. Support for refinement is essential in order to prevent mistakes in the development of a concrete specification from an abstract one. Most of the ADLs support architecture decomposition, but only a few support architecture refinement through different abstraction levels (see [42]).

SADL has been especially designed for supporting architecture refinement. In SADL, different levels of specifications are related by refinement mappings, but the refinement is only structural. Rapide is used to define architectures based on event processing. Refinement in Rapide is only behavioural, i.e. relating two architectures by mapping abstract events and concrete events.

π-ARL [48] is an architecture refinement language (ARL) based on rewriting logic. It enables the stepwise refinement of software architectures modelled with π-ADL and supports structural and behavioural refinement.

Our approach supports both structural and behavioural refinement in the form of the techniques for horizontal and vertical implementation presented in section 9.3. We have not defined a specific language to describe the refinements. However, the mappings of abstract actions have to be in a particular syntactic form in order to be applied automatically.

13.3 Formal Methods

Formal methods aim to provide full support for formal specification, analysis, and development of software systems and refer to mathematically based techniques to accomplish this. Formal development techniques such as B [2], VDM [20], and Z [15] provide refinement mechanisms, but they do not have support for architecture descriptions. FOCUS [60] does not address refinement itself, however a variant based on FOCUS proposes to refine Data-Flow Architectures (FOCUS/DFA) [51]. It is based on the addition and removal of connections and components.

The π-Method [49] has been built from scratch to support architecture-centric formal software engineering. It is based on the higher-order typed π-calculus and mainly built around the architecture description language π-ADL and the architecture refinement language π-ARL. Tool support comes in the form of a visual modeler, animator, refiner, and code synthesiser. The aim of this formal method is to provide full support for formal description and development.

13.4 Workbenches and Environments

To our knowledge there is no work done on generalizing software engineering workbenches and creating software engineering environments from instances of the generalized

workbenches. Such workbenches can easily be designed for the π-Method with the use of its supporting tools.

There are many meta software development environments with which an environment can be created by integrating a set of existing tools. Such integration can easily be developed with the PSF-ToolBus software engineering environment as is shown in Chapter 11. Here, an integrated development environment for PSF is created from the tools of the PSF Toolkit using the ToolBus to control the communication between the tools.

Part IV

Evaluation

Chapter 14

Conclusions

In Part I we set out the aims and scope for this thesis. We introduced the process algebra based language PSF, and its accompanying toolkit. Furthermore, we introduced the ToolBus which we use as coordination architecture for the software we develop.

In Part II we have extended the PSF Toolkit with an animation facility coupled to the simulator. Animations can be built up from standard objects provided by a library. These animations can be controlled by the simulator or by the animation itself, which means that a user can select actions to be executed by the simulator through the animation. We also developed a tool to generate an animation from a specification. Processes are presented by ellipses and communications between processes by arrows. Although all processes and communications are visualized in the same way, a generated animation still reflects the specification in that the nodes and arrows represent (parallel) processes and their communications in the specification. Therefore it can be very useful in the testing of the specification.

In Part III we experimented with applying PSF in the field of software engineering. From an existing specification for the PSF compiler we developed a ToolBus application specification making use of a PSF library. This PSF ToolBus library has been especially developed for the specification of ToolBus applications. We used the ToolBus application specification of the compiler to extract a specification of the architecture of the compiler by using abstraction. Furthermore, we built a parallel compiler by developing a specification of the architecture, a refined specification, and an implementation, with maximal reuse of the specification and implementation for the compiler as ToolBus application.

To support the specification of software architecture we developed a PSF library. We used this library to develop a new implementation for the simulator from the PSF Toolkit, starting with an architecture specification and refining this specification to obtain a specification of the simulator as ToolBus application. This refinement is based on vertical and horizontal implementation techniques. From the ToolBus application specification an implementation of the simulator was developed as ToolBus application. We extended the

new implementation with a history mechanism, illustrating that adding functionality to the finished product need not lead to any problems in our software development process.

So far we (re-)engineered existing tools. To fully test our software development process we engineered an IDE for PSF. Because we could make use of the existing implementation of the tools to be integrated, the focus was completely on the design phase for the IDE. For this integration, the tools needed no modifications at all. The components that make up the control of the IDE are kept small and are easily replaceable due to the use of the ToolBus as coordination architecture. In engineering the IDE we encountered no problems with the use of the PSF libraries for architecture specification and ToolBus application specification.

The result of our research is a software development process consisting of making an architecture specification, refining the specification into a ToolBus application specification, and making an implementation from the ToolBus application specification using the ToolBus as coordination architecture. We described our software development process more formally by presenting it in a CASE setting, resulting in the PSF-ToolBus Software Engineering Environment. We generalized the refinement step in this environment to the PSF Software Engineering Workbench. Several instances of this workbench can be used to form a software engineering environment. Generalizing from the specification language in the PSF Software Engineering Workbench we obtained the Process Algebra Software Engineering Workbench suitable to form software engineering environments for process algebra based languages similar to PSF.

14.1 PSF in the Field of Software Engineering

We showed that we can develop software architecture specifications with the use of the PSF Architecture library. The use of scenario's enabled a stepwise development. By applying vertical and horizontal implementation techniques we obtained ToolBus application specifications based on the PSF ToolBus library. From the ToolBus application specifications we developed implementations with the use of the ToolBus as coordination architecture.

We validated the specifications with the use of the simulator from the PSF Toolkit combined with animation. We think this is the strongest argument for using PSF in software engineering, because now it is possible to test already in the design phase instead of having to wait until the implementation phase.

The modular structure of PSF makes it possible to use and develop libraries. The libraries made it possible to develop workbenches for specifying architectures and ToolBus applications. The combination of the two workbenches with application of vertical and horizontal implementation tools resulted in the PSF-ToolBus Software Engineering Environment.

14.2 Support for Validation of Specifications

In the past we validated many specifications with the use of the PSF Toolkit. The simulator was heavily used in the validation, but for larger specifications it was difficult to keep track of the current state. The animation facility improved this a lot, especially the active

animation that shows which action belongs to which process. It was very helpful in the validation of the design specifications and in playing with ideas for the design.

The use of the PSF Toolkit in the development of tools for the PSF Toolkit shows that it is very useful, and the addition of an IDE made it more appealing to users that prefer a graphical user interface to a command line interface.

Because the compiler supports the use of libraries we could use the developed libraries for architectures and ToolBus applications as standard libraries. This makes it easier to use the same library for specification instead of a local copy, preventing inconsistencies between local copies. This also made the development of the workbenches easier.

14.3 Software Engineering

From our experience in software engineering with PSF we can list some advantages of this approach. We already mentioned the validation of design, that can reveal conceptual and logical errors in an early phase instead of only in the implementation phase. The validation of design is also useful in later adaptation and evolution of the software. Changes can be tested at a higher level of abstraction, giving insight into the impact of the change on the design and on the effort needed to implement the change.

The animation of the specification can be used in communication to the stakeholders, not only by showing a boxes-and-lines diagram but also by bringing it to live through simulation and animation of the specification. In that respect, the animation serves as a very abstract prototype. It is also useful for showing the design to new members of the design team. The animation of the specification in fact adds a dimension to the documentation of the system.

The validation of design gives an immediate feedback in the development process which can save a lot of effort and costs. Feedback from a later phase in the software development process can lead to redesign. In the redesign parts from the old design can be used. The reuse of parts of a design shows which parts in later phases need to be changed and which do not have to be changed. Of course, design decisions in a later phase can always lead to new implementation of parts.

The scenario's used in the design phase can be used as a base for developing test scenario's for testing the implementation. They can be seen as an abstract form of test scenario's.

A disadvantage can be that a thorough knowledge of process algebra is needed by the design team. However, the animation can help users with insufficient knowledge of process algebra. It could be of value if a user can build specifications with a visual language that hides some of the underlying process algebra.

14.4 Usage

Recently (June 2008), software engineering with PSF as described in this thesis has been used in a case study on a domotics (home automation) application [58]. In this work, a simulated hardware interface to be controlled through several user interfaces in parallel is designed and validated. This project was done as a thesis project by a Bachelor student in

Computer Science who had little knowledge of process algebra. It seemed that due to the simulation and animation tools for design specifications the project progressed easily. In particular, the validation of early designs provided sufficient information for improving these. After a first implementation of the domotics application it was extended with more features encountering no problems in the various stages of the development process. The simulation and animation of the architecture specification also appeared to be very convenient in the presentation of this work. It was concluded that the software engineering development process described in this thesis was useful. However, it was also concluded that the workbench tool for constraining needs some further work (see also section 16.4) because this tool is not capable of handling parameterized processes as used in the architecture specification of this particular domotics application.

Chapter 15

Industrial Application of Software Engineering with Process Algebra

---

We concluded that PSF is useful in software engineering and can even improve software engineering in the design phase. The question arises how to promote its use in industry. In the article *Ten Commandments of Formal Methods* [12] by Bowen and Hinchey (1995), it is concluded that it is difficult to get formal methods accepted by industry. That this is still true is concluded in the *Ten Years Later* version of this paper [13]. It must be said that over the last couple of years formal methods are being applied in industry, but mostly these are used by people from academia working (temporarily) in industry and not by their co-workers. Although that is a start and the industry gets acquainted with formal methods, it is not enough to give formal methods a regular base in industry.

The problem with introducing formal methods in industry is that users do not have sufficient training to understand the methods and often lack the experience in applying formal methods. Most of the formal methods and their tools are difficult to use. Another problem is to show the advantages of formal methods to an industrial audience. Below we address these problems with respect to software engineering with process algebra.

15.1 Design

A strong point of software engineering with PSF is the validation of design. It allows for stepwise development of the design by using scenario's. The scenario's together with the animations can be used to show the design decisions in the development process. The animation of design is very convenient in communication to stakeholders and can be used to show the advantages of this method.

15.2 Consistency of Design and Implementation

We think that validation of design is a necessity. Without it, faults in the design or poor

design decisions will find their way into the implementation phase. Fixing these faults in that stage by going back to the design phase can be very costly. In most cases this is fixed by altering the implementation without any alteration of the design. This makes maintenance more difficult (if not hardly possible) and even more costly. In this context, the increasing use of open source software [1] raises some concerns. All changes in open source software are made only on the implementation level since there is no design (available). In some cases there is a maintainer that decides which change gets into the main version, and thus can keep an eye on the design. In our opinion open source software needs to have an open design. Validation of design can help in keeping the design consistent with the implementation.

15.3 Training

We can conclude that in order to get process algebra being applied in industrial software engineering, the users need some training in process algebra and get experience with its application. The best way to establish that is to make process algebra part of the programmes for computer science education, although it will take several years to reach industry. A short course is necessary to impart people with some knowledge of process algebra. Animation can be a convenient tool in this course to show the behaviour of process algebra specifications.

15.4 Tools

For process algebra to be applied in industry, the tools can be a problem. The tools supporting process algebra need to be made more accessible and need to hide as much as possible from the details and complexities of process algebra in order to be used by people without a thorough knowledge of process algebra. At the moment of writing this thesis the user has to make a specification from which an animation is generated. The other way around could be better. A user starts with a box-and-line diagram and gives meaning to the boxes and lines in the form of a small specifications. From these small specifications a full-blown specification and animation can be generated on which the existing tools can work. In this way not only the result is appealing, also the way to get to the result might be appealing.

Chapter 16

Further Work

In this chapter we describe our thoughts on further work. Some of these thoughts are on extension of existing tools but also a proposal for a visual specification tool is made. Other thoughts are on the extension of the *Software Engineering with Process Algebra* process.

16.1 Animation

Larger animations can become quite cluttered. Possible reasons for this problem are the large number of processes and the communications between them, and the size of the animation that does not fit on the screen. With generated animations, part of this problem is due to the automatically generated layout, which is not always satisfactory. This is partially solved by clustering encapsulated processes (surrounded by a rectangle in the animations), but this usually takes more space. A solution might be to collapse such a cluster to a single process. All actions inside this cluster can then be considered local actions of this subsystem process and all communications of processes outside this cluster with processes inside the cluster become communications with the subsystem process. Typically this can be done for encapsulated clusters of processes that are also hidden. If during animation the user likes to see the internals of the subsystem process it can be expanded in the animation or in a different window.

All processes in the animation are the result of static analysis of the specification. This is sufficient for most specifications, but sometimes dynamic generation of (a part of) the animation is necessary in order to do justice to the intended behaviour of the specification. It can be quite some work to implement this feature, but we believe it is feasible.

16.2 PSF ToolBus Library

The current version of the PSF ToolBus Library only partly implements the actions in the ToolBus. A possible extension of the library may be the specification of notes, which are used in the ToolBus for the broadcast of messages and for asynchronous communication.

In order to be able to fully specify ToolBus scripts in PSF the functions and predicates have to be added to the library. To do this, the terms used in the ToolBus have to be specified in detail. These are now handled only symbolically.

16.3 System Models

In our research we used only the ToolBus as a target system model. We think that there are no problems using other models. To really convince ourselves of this and to support software development with process algebra for other models we have to develop PSF libraries for the specification of these models. Some of these models can be intermixed allowing for implementation of parts of the system in different models, but they can also be refinements of other models allowing for more levels of design.

Examples of other system models that we can think of are service oriented architectures (SOAs) and thread models for making efficient use of multi-core processors. In these models parallelism plays a role, but also models for non-parallel applications are possible, such as a function call mechanism. A function call mechanism consists after all of communication of the arguments to an object capable of performing a particular function, function execution, and communication of a return value back to the caller of the function.

16.4 Workbench Tools

In the PSF-ToolBus Software Engineering Environment we use a refine and a constrain tool. At the moment these are ad hoc tools based on matching of regular expressions and a standard layout of the specifications. These tools are automatically applied in the development of the tools described in this thesis, without the need for modification by hand. For general use however, certainly in industry, they have to be made more robust. The tools should accept any layout, be capable of handling all features of PSF, and be more flexible with the naming of processes. Possibly, tools for making the input (mappings and constraints) are needed to aid in the development process.

16.5 Visual Specification Language

Up to now we developed specifications and generated animations from these. As mentioned in section 15.4, for users with little knowledge of process algebra it can be convenient to start with a box-and-line diagram and give meaning to the boxes and lines in the form of small specifications. From the diagram and specifications a full-blown specification may be generated on which the current tools can work. To do this, a visual specification language is needed and a tool for editing specifications in this language. The full-blown specification can be hidden from the user by integrating the editing tool in a development environment.

Bibliography

[1] *Open Source Initiative*, url: http://www.opensource.org/.

[2] J.R. Abrial, *The B-Book: Assigning Programs to Meanings,* Cambridge University Press, 1996.

[3] R. Allen and D. Garlan, ‘‘A Formal Basis for Architectural Connection,’’ *ACM Transaction on Software Engineering and Methodology*, no. 6, pp. 213-249, 1997.

[4] J.A. Bergstra and J.W. Klop, ‘‘Process algebra for synchronous communication,’’ *Information and Control*, vol. 60, pp. 109-137, 1984.

[5] J.A. Bergstra, J. Heering, and P. Klint (eds.), ‘‘The Algebraic Specification Formalism ASF,’’ in *Algebraic Specification*, ACM Press Frontier Series, pp. 1-66, Addison-Wesley, 1989.

[6] J.A. Bergstra and P. Klint, ‘‘The discrete time ToolBus,’’ *Science of Computer Programming*, vol. 31, no. 2-3, pp. 205-229, July 1998.

[7] M. Bernardo, P. Ciancarini, and L. Donatiello, ‘‘Architecting Families of Software Systems with Process Algebras,’’ *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 4, pp. 386-426, 2002.

[8] M. Bernardo, P. Ciancarini, and L. Donatiello, ‘‘Detecting Architectural Mismatches in Process Algebra Descriptions of Software Systems,’’ in *Proceeedings 2nd Working IEEE/IFIP Conference on Software Architecture (WICSA’01)*, 2001.

[9] P. Binns, M. Engelhart, M. Jackson, and S. Vestal, ‘‘Domain-Specific Software Architectures for Guidance, Navigation, and Control,’’ *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, 1996.

[10] T. Bolognesi and E. Brinksma, ‘‘Introduction to the ISO specification language LOTOS,’’ *Computer Networks and ISDN Systems*, vol. 14, pp. 25-59, 1987.

[11] L. Bouge and N. Francez, ‘‘A Compositional Approach to Superimposition,’’ in *Proc. of the 14th ACM POPL’88*, pp. 240-249, 1988.

[12] J.P. Bowen and M.G. Hinchey, ‘‘Ten Commandments of Formal Methods,’’ *IEEE Computer*, vol. 28, no. 4, pp. 56-63, 1995.

[13] J.P. Bowen and M.G. Hinchey, ‘‘Ten Commandments of Formal Methods ... Ten Years Later,’’ *IEEE Computer*, vol. 39, no. 1, pp. 40-48, 2006.

[14] P. Combes, F. Dubois, and B. Renard, ‘‘An Open Animation Tool: Application to Telecommunication Systems,’’ *Computer Networks*, vol. 40, no. 5, pp. 599-620.

[15] J. Davies and J. Woodcock, *Using Z: Specification Refinement and Proof,* International Series in Computer Science, Prentice Hall, 1996.

[16] B. Diertens, *PSF Home Page*, url: http://www.science.uva.nl/˜psf/.

[17] B. Diertens, ‘‘New Features in PSF I - Interrupts, Disrupts, and Priorities,’’ report P9417, Programming Research Group - University of Amsterdam, June 1994.

[18] B. Diertens and A. Ponse, ‘‘New Features in PSF II - Iteration and Nesting,’’ report P9425, Programming Research Group - University of Amsterdam, October 1994.

[19] B. Diertens, ‘‘Simulation and Animation of Process Algebra Specifications,’’ report P9713, Programming Research Group - University of Amsterdam, September 1997.

[20] J. Fitzgerald and P. Larsen, *Modelling Systems: Practical Tools and Techniques for Software Development,* Cambridge University Press, 1998.

[21] A. Fuggetta, ‘‘A Classification of CASE Technology,’’ *IEEE Computer*, vol. 26, no. 12, pp. 25-38, 1993.

[22] E.R. Gansner and S.C. North, ‘‘An Open Graph Visualization System and its Applications to Software Engineering,’’ *Software -- Practice and Experience*, vol. 30, no. 11, pp. 1203-1233, 2000.

[23] H. Garavel, F. Lang, and R. Mateescu, ‘‘An overview of CADP 2001,’’ report 0254, INRIA, 2001.

[24] D. Garlan, R. Allen, and J. Ockerbloom, ‘‘Exploiting Style in Architectural Design Environments,’’ in *Proceedings SIGSOFT’94: Foundations of Software Engineering*, pp. 175-188, 1994.

[25] D. Garlan, R. Monroe, and D. Wile, ‘‘ACME: An Architecture Description Interchange Language,’’ in *Proceedings of CASCON’97*, 1997.

[26] J.F. Groote and A. Ponse, ‘‘Syntax and semantics of μCRL,’’ in *Proceeedings 1st Workshop on the Algebra of Communicating Processes (ACP’94)*, pp. 26-62, Springer, 1995.

[27] M. Heisel and M. Levy, ‘‘Using LOTOS Patterns to Characterize Architectural Styles,’’ in *Proc. of the 7th International Conference on Algebraic Methodology and Software Technology*, LNCS, 1999.

[28] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice Hall, 1985.

[29] International Organization for Standardization, *Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour,* ISO, 1989.

[30] ITU, ‘‘Message Sequence Chart (MSC),’’ ITU-T Z.120, International Telecommunications Union, Geneva, Switzerland, 2000.

[31] H.A. de Jong, ‘‘Flexible Heterogeneous Software Systems,’’ Ph.D. thesis, University of Amsterdam, 2007.

[32] H.A. de Jong and A.T. Kooiker, *My Favorite Editor Anywhere,* Lecture Notes in Computer Science, 3475, pp. 122-131, Springer-Verlag, 2005.

[33] B.W. Kernighan and D.M. Ritchie, *The C Programming Language,* second edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[34] A. Lopes and J.L. Fiadeiro, ‘‘Superposition: Composition vs Refinement of Non-deterministic, Action-Based Systems,’’ *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 3, pp. 352-366, 2002.

[35] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, ‘‘Specification and Analysis of System Architecture Using Rapide,’’ *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336-355, 1995.

[36] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, ‘‘Specifying Distributed Software Architectures,’’ in *Proc. of the 5th European Software Engineering Conf. (ESEC ’95)*, pp. 137-153, LNCS, 1995.

[37] S. Mauw, ‘‘PSF - A Process Specification Formalism,’’ Ph.D. thesis, University of Amsterdam, 1991.

[38] S. Mauw and G.J. Veltink (eds.), *Algebraic Specification of Communication Protocols,* Cambridge Tracts in Theoretical Computer Science 36, Cambridge University Press, 1993.

[39] S. Mauw and G.J. Veltink, ‘‘A Tool Interface Language for PSF,’’ report P8912, Programming Research Group - University of Amsterdam, October 1989.

[40] S. Mauw and J.C. Mulder, ‘‘A PSF Library of Data Types,’’ in *Proceedings of the ASF+SDF95 Workshop*, pp. 53-64, 1995.

[41] N. Medvidovic and D.S. Rosenblum, ‘‘A Language Environment for Architecture-Based Software Development and Evolution,’’ in *Proceedings 21st International Conference on Softwware Engineering (ICSE’99)*, pp. 44-53, 1999.

[42] N. Medvidovic and R. Taylor, ‘‘A Classification and Comparison Framework for Architecture Description Languages,’’ *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93, 2000.

[43] R. Milner, *Communication and Concurrency,* Prentice Hall, 1989.

[44] B. Molenaar, *Vim the editor*, url: http://www.vim.org/.

[45] M. Moriconi, X. Qian, and R.A. Riemenschneider, ‘‘Correct Architecture Refinement,’’ *IEEE Transactions on Software Engineering*, vol. 21, no. 4, 1995.

[46] M. Moriconi and R.A. Riemenschneider, ‘‘Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies,’’ technical report SRI-CSL-97-01, Computer Science Laboratory - SRI international, March 1997.

[47] F. Oquendo, ‘‘π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures,’’ *ACM Software Engineering Notes*, vol. 20, no. 3, 2004.

[48] F. Oquendo, ‘‘π-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures,’’ *ACM Software Engineering Notes*, vol. 29, no. 5, 2004.

[49] F. Oquendo, ‘‘π-Method: A Model-Driven Formal Method for Architecture-Centric Software Engineering,’’ *ACM Software Engineering Notes*, vol. 31, no. 3, 2006.

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

[51] J. Philipps and B. Rumpe, ‘‘Refinement of Pipe and Filter Architectures,’’ in *Proc. of FM’99*, LNCS, 1999.

[52] A. Rensink and R. Gorrieri, ‘‘Action Refinement,’’ in *Handbook of Process Algebra*, ed. J.A. Bergstra, A. Ponse, S.A. Smolka, pp. 1047-1147, Elsevier Science, 2001.

[53] A. Rensink and R. Gorrieri, ‘‘Vertical Implementation,’’ *Information and Computation*, vol. 170, no. 1, pp. 95-133, 2001.

[54] N.S. Rosa and P.R.F. Cunha, ‘‘A Software Architecture-Based Approach for Formalising Middleware Behaviour,’’ *Electronic Notes in Theoretical Computer Science*, vol. 108, pp. 39-51, 2004.

[55] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, ‘‘Abstractions for Software Architecture and Tools to Support Them,’’ *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314-335, 1995.

[56] I. Sommerville, *Software Engineering,* 7th edition, Pearson Education, 2004.

[57] R.M. Stallman, ‘‘Emacs the Extensible, Customizable Self-Documenting Display Editor,’’ *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pp. 147-156, 1981.

[58] D. Staudt, ‘‘Engineering a Domotics Application with PSF,’’ bachelor thesis, Programming Research Group - University of Amsterdam, June 2008.

[59] B. Stepien and L. Logrippo, ‘‘Graphic Visualization and Animation of LOTOS Execution Traces,’’ *Computer Networks*, vol. 40, no. 5, pp. 665-681.

[60] K. Stolen and M. Broy, *Specification and Development of Interactive Systems,* Monographs in Computer Science, Springer-Verlag, 2001.

[61] K.J. Turner and I.A. Robin, ‘‘An interactive visual protocol simulator,’’ *Computer Standards & Interfaces*, vol. 23, pp. 279-310, 2001.

[62] G.J. Veltink, ‘‘The PSF Toolkit,’’ *Computer Networks and ISDN Systems 25*, pp. 875-898, 1993.

[63] G.J. Veltink, ‘‘Tools for PSF,’’ Ph.D. thesis, University of Amsterdam, 1995.

[64] G.J. Veltink, ‘‘From PSF to TIL,’’ report P9009, Programming Research Group - University of Amsterdam, 1990.

[65] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren, ‘‘Control and Data Transfer in the Distributed Editor of the ASF+SDF Meta-environment,’’ report P9415, Programming Research Group - University of Amsterdam, May 1994.

[66] S.F.M. van Vlijmen and A. van Waveren, ‘‘On Generating Synchronous Interworkings from PSF Process Traces,’’ report P9304, Programming Research Group - University of Amsterdam, 1993.

[67] L. Wall, T. Christiansen, and R.L. Schwartz, *Programming Perl,* O’Reilly & Associates, Inc., 1996.

[68] J.J. van Wamel, ‘‘A Library for PSF,’’ report P9301, Programming Research Group - University of Amsterdam, 1993.

Appendix A

PSF Specifications

A.1 Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is a simple communication protocol that is often used as a test case, either for some algebraic formalism or for some tool for the analysis or verification of concurrent systems. Although it is simple, it contains many interesting ingredients from concurrency theory. ABP consists of a *Sender*, a *Receiver*, and the channels K and L.

First, *Sender* reads a message at its input port. This message is extended with a control bit to form a frame and this frame is sent along channel K. The sending of the frame proceeds until *Sender* receives an acknowledgement of a succesful transmission (the control bit) over channel L. After a succesful transmission *Sender* flips the control bit and starts all over again.

Channel K transmits frames from *Sender* to *Receiver*. There are two situations that can occur. The frame is properly transmitted, or the frame is corrupted during transmission.

Receiver reads a frame from channel K. It is assumed that *Receiver* is able to tell, e.g. by performing a checksum control, whether or not the frame has been corrupted. When the frame is correct, *Receiver* checks the control bit in the frame. If this bit matches the internal control bit of *Receiver*, the message in the frame is sent to the output port, and *Receiver* sends an acknowledgement with the control bit to *Sender* over channel L. *Receiver* then flips his internal control bit and waits for another frame. In all other cases, *Receiver* sends a negative acknowledgement (with a flipped control bit), and waits for a retransmission of the frame.

Channel L is used to transmit acknowledgements from *Receiver* to *Sender*. Like channel K, it is able to corrupt data. It is assumed that *Sender* can tell whether an acknowledgement has been corrupted.

ABP can be specified in PSF as follows.

```
data module Bits
begin
    exports
    begin
        sorts
            BIT
        functions
            0 :→ BIT
            1 :→ BIT
            flip : BIT → BIT
    end
    equations
    [B1] flip(0) = 1
    [B2] flip(1) = 0
end Bits

data module Data
begin
    exports
    begin
        sorts
            DATA
        functions
            'a :→ DATA
            'b :→ DATA
            'c :→ DATA
            'd :→ DATA
            'e :→ DATA
    end
end Data

data module Frames
begin
    exports
    begin
        sorts
            FRAME
        functions
            frame : BIT # DATA → FRAME
            frame-error :→ FRAME
    end
    imports
        Data, Bits
end Frames

data module Acknowledgements
begin
    exports
    begin
        sorts
            ACK
        functions
            ack : BIT → ACK
            ack-error :→ ACK
    end
    imports
        Bits
end Acknowledgements

process module ABP
begin
```

```
imports
   Bits, Data, Frames, Acknowledgements
atoms
   input : DATA
   send-frame : FRAME
   receive-ack-or-error : ACK
   receive-frame : FRAME
   send-frame-or-error : FRAME
   receive-frame-or-error : FRAME
   output : DATA
   send-ack : ACK
   receive-ack : ACK
   send-ack-or-error : ACK
   frame-comm : FRAME
   frame-or-error : FRAME
   ack-comm : ACK
   ack-or-error : ACK
processes
   Sender
   Receive-Message : BIT
   Send-Frame : BIT # DATA
   Receive-Ack : BIT # DATA
   K
   K : BIT # DATA
   Receiver
   Receive-Frame : BIT
   Send-Ack : BIT
   Send-Message : BIT # DATA
   L
   L : BIT
   ABP
sets
   of atoms
      H = { send-frame(f), receive-frame(f) | f in FRAME }
         + { send-frame-or-error(f), receive-frame-or-error(f)
            | f in FRAME }
         + { send-ack(a), receive-ack(a) | a in ACK }
         + { send-ack-or-error(a), receive-ack-or-error(a)
            | a in ACK }
      I = { frame-comm(f), frame-or-error(f) | f in FRAME }
         + { ack-comm(a), ack-or-error(a) | a in ACK }
   of BIT
      Bit-Set = { 0, 1 }
communications
   send-frame(f) | receive-frame(f) = frame-comm(f)
      for f in FRAME
   send-frame-or-error(f) | receive-frame-or-error(f) =
      frame-or-error(f) for f in FRAME
   send-ack(a) | receive-ack(a) = ack-comm(a) for a in ACK
   send-ack-or-error(a) | receive-ack-or-error(a) =
      ack-or-error(a) for a in ACK
variables
   f :→ FRAME
   b :→ BIT
   d :→ DATA
   a :→ ACK
definitions
   Sender = Receive-Message(0)
   Receive-Message(b) =
         sum(d in DATA, input(d) . Send-Frame(b,d))
   Send-Frame(b,d) = send-frame(frame(b,d)) . Receive-Ack(b,d)
```

```
        Receive-Ack(b,d) = (
                    receive-ack-or-error(ack(flip(b)))
                  + receive-ack-or-error(ack-error)
                  ) . Send-Frame(b,d)
                + receive-ack-or-error(ack(b)) .
                  Receive-Message(flip(b))

        K = sum(d in DATA, sum(b in Bit-Set,
              receive-frame(frame(b,d)) . K(b,d) ))
        K(b,d) = (
                  skip . send-frame-or-error(frame(b,d))
                + skip . send-frame-or-error(frame-error)
                ) . K

        Receiver = Receive-Frame(0)
        Receive-Frame(b) = (
                    sum(d in DATA,
                       receive-frame-or-error(frame(flip(b),d)))
                  + receive-frame-or-error(frame-error)
                  ) . Send-Ack(flip(b))
                + sum(d in DATA, receive-frame-or-error(frame(b,d)) .
                    Send-Message(b,d)
                  )
        Send-Ack(b) = send-ack(ack(b)) . Receive-Frame(flip(b))
        Send-Message(b,d) = output(d) . Send-Ack(b)

        L = sum(b in Bit-Set, receive-ack(ack(b)) . L(b) )
        L(b) = (
                  skip . send-ack-or-error(ack(b))
                + skip . send-ack-or-error(ack-error)
                ) . L

        ABP = hide(I, encaps(H, Sender || Receiver || K || L ))
    end ABP
```

A.2 Factory

The factory consist of an input and output, some stations and conveyer belts. It produces the products A and B, which take slightly different routes through the factory.

```
data module Products
begin
    exports
    begin
        sorts
            PRODUCT
        functions
            A : → PRODUCT
            B : → PRODUCT
    end
end Products

data module Stations
begin
    exports
    begin
        sorts
            STATION
        functions
            1 : → STATION
```

```
        2 : → STATION
        3 : → STATION
        4 : → STATION
        5 : → STATION
        6 : → STATION
        eq-stat : STATION # STATION → BOOLEAN
        next : STATION # PRODUCT → STATION
  end
  imports
     Booleans, Products
  variables
     x : → STATION
     y : → STATION
     p : → PRODUCT
  equations
  [1] eq-stat(x, x) = true
  [2] not(eq-stat(x, y)) = true
  [3] next(1, p) = 2
  [4] next(2, p) = 3
  [5] next(3, A) = 4
  [6] next(3, B) = 5
  [7] next(4, p) = 5
  [8] next(5, p) = 6
end Stations

process module Factory
begin
  imports
     Stations
  atoms
     input : PRODUCT
     output : PRODUCT
     read-input : PRODUCT
     send-input : PRODUCT
     comm-input : PRODUCT
     read-output : PRODUCT
     send-output : PRODUCT
     comm-output : PRODUCT
     to-belt : STATION # STATION # PRODUCT
     from-belt : STATION # PRODUCT
     comm-belt : STATION # STATION # PRODUCT
  processes
     Start
     Input
     Stations
     Station : STATION
     Output
  sets
     of PRODUCT
        PRODUCT-set = { A, B }
     of STATION
        STATION-set = { 1, 2, 3, 4, 5, 6 }
     of atoms
        H = { send-input(p), read-input(p),
              send-output(p), read-output(p),
              to-belt(x, y, p), from-belt(y, p)  | p in PRODUCT,
              x in STATION, y in STATION }
  communications
     send-input(p) | read-input(p) = comm-input(p)
        for p in PRODUCT
     send-output(p) | read-output(p) = comm-output(p)
```

```
            for p in PRODUCT
         to-belt(s1, s2, p) | from-belt(s2, p) = comm-belt(s1, s2, p)
            for s1 in STATION, s2 in STATION, p in PRODUCT
      variables
         s : → STATION
      definitions
         Start = encaps(H, Input ‖ Stations ‖ Output)
         Input = sum(p in PRODUCT-set, input(p) . send-input(p)) .
               Input
         Stations = merge(s in STATION-set, Station(s))
         Station(s) =
               [eq-stat(s, 1) = true] → (
                  sum(p in PRODUCT,
                     read-input(p) . to-belt(s, next(s, p), p)
                  ) . Station(s)
               )
            + [eq-stat(s, 6) = true] → (
                  sum(p in PRODUCT,
                     from-belt(s, p) . send-output(p)
                  ) . Station(s)
               )
            + [and(not(eq-stat(s, 1)), not(eq-stat(s, 6))) = true] →
               (
                  sum(p in PRODUCT,
                     from-belt(s, p) . to-belt(s, next(s, p), p)
                  ) . Station(s)
               )
         Output = sum(p in PRODUCT, read-output(p) . output(p)) .
               Output
end Factory
```

A.3 Scheduled Factory

The modules *Products* and *Stations* are the same as for the factory without scheduler.

```
process module Scheduler
begin
   exports
   begin
      atoms
         send-request : STATION # PRODUCT
         rec-request : STATION # PRODUCT
         comm-request : STATION # PRODUCT
         send-next : STATION # STATION
         rec-next : STATION # STATION
         comm-next : STATION # STATION
         send-start : PRODUCT
         rec-start : PRODUCT
         comm-start : PRODUCT
         send-end
         rec-end
         comm-end
      processes
         Scheduler
      sets
         of atoms
            HS = { send-request(s1, p), rec-request(s1, p),
               send-next(s1, s2), rec-next(s1, s2),
               send-start(p), rec-start(p), send-end, rec-end
               | s1 in STATION, s2 in STATION, p in PRODUCT }
```

```
    end
    imports
       Stations
    processes
       Next : STATION # PRODUCT # STATION
       SubScheduler : STATION # PRODUCT
    communications
       send-request(s, p) | rec-request(s, p) = comm-request(s, p)
          for s in STATION, p in PRODUCT
       send-next(s1, s2) | rec-next(s1, s2) = comm-next(s1, s2)
          for s1 in STATION, s2 in STATION
       send-start(p) | rec-start(p) = comm-start(p)
          for p in PRODUCT
       send-end | rec-end = comm-end
    variables
       s : → STATION
       n : → STATION
       p : → PRODUCT
    definitions
       Scheduler =
             sum(p in PRODUCT,
                rec-start(p) .
                (
                   SubScheduler(1, p)
                ||  Scheduler
                )
             )
       SubScheduler(s, p) =
                [not(eq-stat(s, 6)) = true] → (
                   rec-request(s, p) .
                   Next(s, p, next(s, p))
                )
             +   [s = 6] →
                   rec-end
       Next(s, p, n) =
             send-next(s, n) .
             SubScheduler(n, p)
end Scheduler

process module Factory
begin
    imports
       Stations,
       Scheduler
    atoms
       input : PRODUCT
       output : PRODUCT
       read-input : PRODUCT
       send-input : PRODUCT
       comm-input : PRODUCT
       read-output : PRODUCT
       send-output : PRODUCT
       comm-output : PRODUCT
       to-belt : STATION # STATION # PRODUCT
       from-belt : STATION # PRODUCT
       comm-belt : STATION # STATION # PRODUCT
    processes
       Start
       Input
       Stations
       Station : STATION
```

```
   Output
sets
   of PRODUCT
      PRODUCT-set = { A, B }
   of STATION
      STATION-set = { 1, 2, 3, 4, 5, 6 }
   of atoms
      H = { send-input(p), read-input(p),
            send-output(p), read-output(p),
            to-belt(x, y, p), from-belt(y, p)  | p in PRODUCT,
            x in STATION, y in STATION }
communications
   send-input(p) | read-input(p) = comm-input(p)
      for p in PRODUCT
   send-output(p) | read-output(p) = comm-output(p)
      for p in PRODUCT
   to-belt(s1, s2, p) | from-belt(s2, p) = comm-belt(s1, s2, p)
      for s1 in STATION, s2 in STATION, p in PRODUCT
variables
   s : → STATION
definitions
   Start = encaps(HS,
      Scheduler ‖ encaps(H, Input ‖ Stations ‖ Output))
   Input =
      sum(p in PRODUCT-set,
         input(p) .
         send-start(p) .
         send-input(p)
      ) . Input
   Stations = merge(s in STATION-set, Station(s))
   Station(s) =
         [eq-stat(s, 1) = true] → (
            sum(p in PRODUCT,
               read-input(p) .
               send-request(s, p) .
               sum(n in STATION,
                  rec-next(s, n) .
                  to-belt(s, n, p)
               )
            ) . Station(s)
         )
      +  [eq-stat(s, 6) = true] → (
            sum(p in PRODUCT,
               from-belt(s, p) .
               send-output(p)
            ) . Station(s)
         )
      +  [and(not(eq-stat(s, 1)), not(eq-stat(s, 6))) = true] →
         (
            sum(p in PRODUCT,
               from-belt(s, p) .
               send-request(s, p) .
               sum(n in STATION,
                  rec-next(s, n) .
                  to-belt(s, n, p)
               )
            ) . Station(s)
         )
   Output =
      sum(p in PRODUCT,
         read-output(p) .
```

```
                send-end .
                output(p)
            ) . Output
    end Factory
```

Summary

I Introduction

This thesis describes the project of applying process algebra as a formalism in software engineering and software re-engineering. As specification language the process algebra based language PSF (Process Specification Formalism) is used. This language is based on ACP (Algebra of Communicating Processes) and ASF (Algebraic Specification Formalism). PSF is accompanied by a Toolkit containing amongst other components a compiler and a simulator. The tools operate around the Tool Interface Language (TIL). As target software application architecture the ToolBus is used. The ToolBus is a coordination architecture developed at the University of Amsterdam and CWI. It utilizes a scripting language based on process algebra to describe the communication between software tools.

The work described in this thesis is motivated by applications of process algebra in software and software engineering. In software the ToolBus is used for the coordination of components using scripts. These scripts can get quite large and complex for larger software systems and the ToolBus provides limited possibilities for debugging the scripts. Specification of the ToolBus scripts in PSF makes it possible to use the PSF Toolkit to validate the specifications of the scripts. For specifying software architecture several Architecture Description Languages (ADLs) exist, some of which are based on a form of process algebra. Using PSF for the specification of software architecture makes it possible to validate the software architecture with the use of the PSF Toolkit.

II Animation of Process Algebra Specifications

Validation of specifications is done by simulation. Although the simulator from the PSF Toolkit is perfectly capable of simulating the behaviour of specifications, for complex specifications it can be difficult to keep track of what is going on. Visualization of the current state and of transitions between states is useful for a better understanding. For this reason, the PSF Toolkit is extended with an animation facility coupled to the simulator. Animations can be build up from standard objects provided by a library. These animations can be controlled by the simulator or by the animation itself, which means that a user can select actions to be executed by the simulator through the animation.

Animations created by hand have the disadvantage that whenever the specification changes, the animation has to be adapted. This makes animation difficult to use for testing,

especially for larger specifications. To overcome this problem, a tool is developed that generates an animation from a specification. Processes are presented by ellipses and communications between processes by arrows. Although all processes and communications are visualized in the same way, a generated animation still reflects the specification in that the ellipses and arrows can directly be related to (parallel) processes and their communications in the specification.

III Software Engineering with PSF

An existing specification for the PSF compiler is used as start-point of the experimentation with applying PSF in software engineering. From this specification a ToolBus application specification has been developed making use of a PSF library especially developed for the specification of ToolBus applications. From the ToolBus application specification of the compiler a specification of the architecture is extracted by using abstraction. This architecture specification served as the base for building a parallel compiler, which consists of developing a specification of the architecture, a refined specification, and an implementation, with maximal reuse of the specification and implementation for the compiler as ToolBus application.

To support the specification of software architecture a PSF library has been developed. This library is used to develop a new implementation for the simulator from the PSF Toolkit, starting with an architecture specification and refining this specification to obtain a specification of the simulator as ToolBus application. This refinement is based on implementation techniques for refinement and constraining. From the ToolBus application specification an implementation of the simulator has been developed as ToolBus application. The new implementation has been extended with a history mechanism, illustrating that adding functionality to the finished product need not lead to any problems in our software development process.

So far existing tools have been (re-)engineered. To fully test the software development process an Integrated Development Environment (IDE) for PSF has been engineered. Because use of the existing implementation of the tools to be integrated could be made, the focus was completely on the design phase for the IDE. For this integration, the tools needed no modifications at all. The components that make up the control of the IDE are kept small and are easily replaceable due to the use of the ToolBus as coordination architecture. In engineering the IDE there were no problems encountered with the use of the PSF libraries for architecture specification and ToolBus application specification.

The result of the research is a software development process consisting of making an architecture specification, refining the specification into a ToolBus application specification, and making an implementation from the ToolBus application specification using the ToolBus as coordination architecture. This software development process has been described more formally by presenting it in a Computer-Aided Software Engineering (CASE) setting, resulting in the PSF-ToolBus Software Engineering Environment. The refinement step in this environment has been generalized to the PSF Software Engineering Workbench. Several instances of this workbench can be used to form a software engineering environment. Generalizing from the specification language in the PSF Software Engineering Workbench the Process Algebra Software Engineering Workbench

can be obtained. This workbench is suitable to form software engineering environments for process algebra based languages similar to PSF.

IV Evaluation

This thesis shows that software architecture specifications can be developed using the PSF Architecture library. Furthermore, it shows that ToolBus application specifications can be obtained based on the PSF ToolBus library from these architecture specifications by applying refining and constraining as implementation techniques. From the ToolBus application specifications implementations can be developed with the use of the ToolBus as coordination architecture. The advantage of this process is that the specifications can be validated with the use of the simulator of the PSF Toolkit combined with animation. This can reveal conceptual and logical errors in an early phase of the software development process instead of only in the implementation phase. Another advantage is that animation of specifications can be used in communication to the stakeholders. It is also useful in showing the design to new members of the design team. A disadvantage can be that a thorough knowledge of process algebra is needed by the design team.

Although over the last couple of years formal methods are applied in industry, it is difficult to get formal methods accepted by industry. The problem is that users do not have sufficient training to understand the formal methods and that the tools are difficult to use. Another problem is to show the advantages of formal methods to an industrial audience. The strong point of software engineering with PSF is the validation of design. The animation of design is convenient in communication to stakeholders. Faults in the design or poor design decision can be very costly when they find their way into the implementation phase. Mostly these are fixed by altering the implementation but not the design, making maintenance more difficult and costly. Validation of design can help in keeping the design consistent with the implementation. The best way to give users experience with process algebra is to make it part of the programmes for computer science education, although it will take several years to reach industry. Animation can be a convenient tool in the education. The tools supporting process algebra need to hide as much as possible from the details and complexities of process algebra.

Further work can be done on the animations as they can become quite cluttered for large specifications. A solution might be to collapse sub-systems to a single process making all actions inside a sub-system local actions. The only system model used in this thesis is the ToolBus. Support for other system models in the form of PSF libraries has to be developed. Other system models can be Service Oriented Architectures (SOAs), thread models for making efficient use of multi-core processors, and function call mechanisms. The tools for refinement and constraining in the PSF-ToolBus Software Engineering Environment have to be made more robust for general application and tools can be developed to support the making of the refinements and constraints. To make software engineering with process algebra more attractive for users with little knowledge of process algebra a visual specification language can be of use.

Samenvatting

I Introductie

Dit proefschrift beschrijft het project van de toepassing van procesalgebra als formalisme in programmatuur-constructie en re-constructie. Als specificatietaal wordt de op procesalgebra gebaseerde taal PSF (Proces Specificatie Formalisme) gebruikt. Deze taal is gebaseerd op ACP (Algebra van Communicerende Processen) en ASF (Algebraische Specificatie Formalisme). PSF is voorzien van een Gereedschapsset die onder andere een compiler en een simulator bevat. De gereedschappen opereren rond de Gereedschap Verbindings Taal. Als doel voor de architectuur van de programmatuurapplicatie wordt de GereedschapsBus gebruikt. De GereedschapsBus is een coordinatie-architectuur ontwikkeld aan de Universiteit van Amsterdam en het CWI. Het gebruikt een op procesalgebra gebaseerde scripttaal om de communicatie tussen programmatuur-componenten te beschrijven.

Het beschreven werk in dit proefschrift vindt zijn motivatie in toepassingen van procesalgebra in programmatuur en programmatuurconstructie. In programmatuur wordt de GereedschapsBus gebruikt voor de coordinatie van componenten door middel van scripts. Deze scripts kunnen vrij groot en complex worden voor grotere programmatuur-systemen en de GereedschapsBus biedt slechts minimale middelen voor het foutvrij maken van de scripts. Specificatie van de GereedschapsBus-scripts in PSF maakt het mogelijk om de PSF-Gereedschapsset te gebruiken voor de validatie van de specificaties van de scripts. Voor het specificeren van programmatuurarchitectuur bestaan verschillende Architectuur Beschrijvings Talen, waarvan sommigen gebaseerd zijn op procesalgebra. Het gebruik van PSF voor de specificatie van programmatuurarchitectuur maakt het mogelijk om de programmatuurarchitectuur te valideren met behulp van de PSF-Gereedschapsset.

II Animatie van Procesalgebra Specificaties

Validatie van specificaties vindt plaats door middel van simulatie. De simulator uit de PSF-Gereedschapsset is zeker in staat om het gedrag van de specificaties te simuleren, maar bij complexe specificaties kan het moeilijk zijn de simulatie te volgen. Visualisatie van de huidige toestand en de overgangen tussen de toestanden kunnen bijdragen aan een beter begrip. Om deze reden werd de PSF-Gereedschapsset uitgebreid met een animatiefaciliteit die gekoppeld kan worden aan de simulator. Animaties kunnen worden opgebouwd uit standaardobjecten uit een bibliotheek. Deze animaties kunnen onder controle staan van de

simulator of van de animatie zelf. Het laatste betekent dat een gebruiker een door de simulator te executeren actie kan selecteren via de animatie.

Met de hand gecreeerde animaties hebben als nadeel dat wanneer de specificatie gewijzigd wordt, de animatie ook gewijzigd moet worden. Dit maakt het moeilijk om animatie te gebruiken voor testen, in het bijzonder voor grotere specificaties. Om dit probleem te verhelpen is er een gereedschap ontwikkeld dat een animatie genereert vanuit een specificatie. Hierbij worden processen afgebeeld als ellipsen en communicaties tussen de processen als pijlen. Ofschoon alle processen en communicaties op dezelfde wijze worden afgebeeld weerspiegelt de animatie toch de specificatie doordat de ellipsen en pijlen direct gerelateerd kunnen worden aan de (parallele) processen en hun communicaties in de specificatie.

III Programmatuurconstructie met PSF

Een bestaande specificatie voor de PSF-compiler werd gebruikt als startpunt voor het experimenteren met de toepassing van PSF in programmatuurconstructie. Vanuit deze specificatie werd een specificatie voor de GereedschapsBus-applicatie ontwikkeld waarbij gebruik werd gemaakt van een PSF-bibliotheek die speciaal ontwikkeld is voor de specificatie van GereedschapsBus-applicaties. Vanuit de specificatie voor de compiler als GereedschapsBus-applicatie werd de architectuur geëxtraheerd door gebruik te maken van abstractie. Deze architectuurspecificatie werd gebruikt als basis voor het bouwen van een parallelle compiler, wat bestaat uit het ontwikkelen van een specificatie van de architectuur, een verfijnde specificatie en een implementatie met maximaal hergebruik van de specificatie en implementatie voor de compiler als GereedschapsBus-applicatie.

Voor de ondersteuning van de specificatie van programmatuurarchitectuur werd een PSF-bibliotheek ontwikkeld. Deze bibliotheek werd gebruikt voor het ontwikkelen van een nieuwe implementatie voor de simulator uit de PSF-Gereedschapsset, beginnend met een specificatie van de architectuur en een verfijning hiervan om een specificatie van de simulator als GereedschapsBus-applicatie te verkrijgen. Deze verfijning is gebaseerd op implementatietechnieken voor verfijning en beperking. Vanuit de specificatie van de GereedschapsBus-applicatie werd een implementatie voor de simulator als GereedschapsBus-applicatie ontwikkeld. De nieuwe implementatie werd uitgebreid met een mechanisme voor historie om te illustreren dat toevoegen van functionaliteit aan een voltooid produkt niet tot problemen hoeft te leiden in ons ontwikkelingsproces voor programmatuur.

Tot dusverre werden bestaande gereedschappen ge(re)construeerd. Om het ontwikkelingsproces voor programmatuur volledig te testen werd een Geintegreerde Ontwikkelings Omgeving (GOO) voor PSF geconstrueerd. Omdat gebruik kon worden gemaakt van de bestaande implementaties van de te integreren gereedschappen kwam de focus op de ontwerp fase voor de GOO te liggen. Voor de integratie was geen aanpassing van de gereedschappen noodzakelijk. De componenten die de controle over de GOO vormen zijn klein gehouden en zijn gemakkelijk vervangbaar door het gebruik van de GereedschapsBus als coordinatie-architectuur. In de ontwikkeling van de GOO traden er geen problemen op met het gebruik van de PSF-bibliotheken voor specificatie van architectuur en GereedschapsBus-applicatie.

Het resultaat van dit onderzoek is een ontwikkelingsproces voor programmatuur bestaande uit het maken van een architectuurspecificatie, verfijning van de specificatie naar een specificatie van de GereedschapsBus-applicatie en het maken van een implementatie vanuit de specificatie van de GereedschapsBus-applicatie gebruikmakend van de GereedschapsBus als coordinatie-architectuur. Dit ontwikkelingsproces voor programmatuur is formeler beschreven door het te presenteren in een Computer Geassisteerde Programmatuur Constructie kader, resulterend in de PSF-GereedschapsBus Programmatuur Constructie Omgeving. De verfijnings stap in deze omgeving werd gegeneraliseerd tot de PSF Programmatuur Constructie Werkbank. Verscheidene instanties van deze werkbank kunnen worden gebruikt om een constructie-omgeving voor programmatuur te vormen. Generalisatie van de specificatietaal in de PSF Programmatuur Constructie Werkbank levert een Proces-Algebra Programmatuur Constructie Werkbank op. Deze werkbank is geschikt voor het vormen van constructie-omgevingen voor programmatuur van op procesalgebra gebaseerde talen vergelijkbaar met PSF.

IV Evaluatie

Dit proefschrift toont aan dat specificaties van programmatuurarchitectuur kunnen worden ontwikkeld met behulp van de PSF-bibliotheek voor architectuur. Verder toont het aan dat vanuit deze architectuurspecificaties door het toepassen van verfijning en beperking als implementatietechnieken specificaties van GereedschapsBus-applicaties verkregen kunnen worden gebaseerd op de PSF-bibliotheek voor deze GereedschapsBus-applicaties. Vanuit de specificaties van GereedschapsBus-applicaties kunnen implementaties ontwikkeld worden die gebruik maken van de GereedschapsBus als coordinatie-architectuur. Het voordeel van dit proces is dat de specificaties gevalideerd kunnen worden met behulp van de simulator uit de PSF-Gereedschapsset gecombineerd met animatie. Dit kan conceptuele en logische fouten onthullen in een vroege fase van het ontwikkelingsproces voor programmatuur in plaats van pas in de implementatiefase. Een ander voordeel is dat animatie van specificaties gebruikt kan worden in de communicatie met belanghebbenden. Animatie is ook bruikbaar in de presentatie van het ontwerp aan nieuwe leden van een ontwikkelingsteam. Een nadeel zou kunnen zijn dat degelijke kennis van procesalgebra nodig is bij het ontwikkelingsteam.

Hoewel in de laatste jaren formele methoden worden toegepast in de industrie, is het moeilijk om formele methoden geaccepteerd te krijgen door de industrie. Het probleem is dat gebruikers niet voldoende getraind zijn om de formele methoden te begrijpen en dat de gereedschappen moeilijk in het gebruik zijn. Een ander probleem is de voordelen van formele methoden duidelijk te maken aan een industrieel publiek. Het sterke punt van programmatuurconstructie met PSF is de validatie van ontwerp. De animatie van ontwerp is geschikt in de communicatie met belanghebbenden. Fouten in het ontwerp of slechte ontwerpbeslissingen kunnen zeer kostbaar worden als zij hun weg vinden naar de implementatiefase. Meestal worden deze hersteld door de implementatie aan te passen maar niet het ontwerp, wat onderhoud moeilijker en kostbaarder maakt. Validatie van ontwerp kan helpen in het consistent houden van ontwerp en implementatie. De beste manier om gebruikers vertrouwd te maken met procesalgebra is om het onderdeel te maken van de informatica studieprogramma's, hoewel het dan enige jaren duurt voordat het de industrie bereikt. Animatie kan een handig tool zijn bij de educatie. De gereedschappen

die procesalgebra ondersteunen dienen zo veel mogelijk de details en gecompliceerdheid van procesalgebra te verbergen.

Er kan verder gewerkt worden aan de animaties omdat deze nogal onoverzichtelijk kunnen worden voor grotere specificaties. Dit kan opgelost worden door het opvouwen van onderliggende systemen tot een enkel proces waarbij alle acties binnen een onderliggend systeem lokale acties worden. In dit proefschrift wordt alleen de GereedschapsBus gebruikt als systeemmodel. Ondersteuning voor andere systeemmodellen in de vorm van PSF-bibliotheken moet worden ontwikkeld. Andere systeemmodellen kunnen zijn: Service Georienteerde Architecturen, draadmodellen voor het efficient gebruik maken van meervoudige-kern processoren, en functie-aanroep mechanismes. De gereedschappen voor verfijning en beperking in de PSF-GereedschapsBus Programmatuur Constructie Omgeving dienen meer robuust gemaakt te worden voor algemene toepassing en gereedschappen kunnen worden ontwikkeld voor het maken van afbeeldingen en beperkingen. Om programmatuurconstructie met procesalgebra attractiever te maken voor gebruikers met weinig kennis van procesalgebra kan een visuele specificatietaal nuttig zijn.