Marcin Szpyrka, Piotr Matyasik,
Rafał Mrówka, Leszek Kotulski

Alvis modelling language

Draft, version 0.33

January 22, 2012

Contents

1

Introduction

The Phenomena, such as concurrency and non-determinism that are central to modelling embedded or distributed systems, turn out to be very hard to handle with standard techniques, such as peer reviewing or testing. Formal methods included into the design process may provide more effective verification techniques, and may reduce the verification time and system costs. Unfortunately, there is a gap between formal mathematical modelling languages and languages used in everyday engineering practice. Formal methods like Petri nets [1], [2], [3], [4], [5], [6], process algebras [7], [8], [9], [10], [11], [12] or time automata [13], [3] provide techniques for a formal specification and modelling of concurrent systems but they are very seldom used in real IT projects. Due to their specific mathematical syntax, these languages are treated as the ones suitable only for scientists.

Alvis is a new modelling language for developing concurrent (embedded) systems. The language is being developed within the confines of the Alvis project at AGH University of Science and Technology in Krakow, Poland. Beginning of the Alvis project dates back to April 2009. The aim of the project is to work out a language suitable for efficient modelling and formal verification of concurrent systems. Especially, we focus on embedded systems. Alvis [14], [15], [16] is a modelling language for real-time concurrent systems. The key concept of Alvis is *agent*. The name has been taken from the CCS process algebra [9] and denotes any distinguished part of the system under consideration with defined identity persisting in time. In contrast to process algebras, Alvis uses a high level programming language based on the Haskell syntax, instead of algebraic equations. Moreover, it combines hierarchical graphical modelling with high level programming language statements. An Alvis model is composed of three layers:

Graphical layer – is used to define data and control flow among agents. The layer takes the form of a hierarchical graph and supports both *top-down* and *bottom-up* approaches to systems development.

Code layer – is used to describe the behaviour of individual agents. It uses both Haskell functional programming language [17] and original Alvis statements.

System layer – depends on the model running environment i.e. the hardware and/or operating system. The layer is the predefined one. The layer is used for simulation and analysis purposes.

Alvis uses a very small number of graphical items and language statements. Our goal was to provide a flexible language with a small number of concepts, but with a possibility of a formal verification of models. An Alvis model semantic finds expression in a LTS graph (*labelled transition system*). Execution of any language statement is expressed as a transition between formally defined states of such a model. An LTS graph can be encoded using *Binary Coded Graphs* (BCG) format and the CADP toolbox [18] and model checking techniques [19] can be used to verify its properties.

1.1 Related works

Alvis has its origins in the CCS process algebra [9], [10] and the XCCS language [20]. The main result of the fact is the communication model used in Alvis that is similar to the one used in CCS and the rendes-vous mechanism used in Ada [21]. However, the Alvis language has many features in common with other modelling languages used in industry.

E-LOTOS is an extension of the LOTOS modelling language (Language Of Temporal Ordering Specification) [22]. The most important enhancements introduced in E-LOTOS are: the quantitative time, a new definition for data types and the construction of values of predefined types, modularity, functions, and processes in separate modules, controlling their visibility, module interfaces, and the definition of generic modules, very useful for code reuse and development distribution. E-LOTOS adds new operators like: the sequential composition operator that allows concatenation of two processes, the general parallel operator, the suspend/resume operator, operators to raise exceptions and to handle them, the renaming operator which allows renaming of actions and exceptions, and to modify their parameters. The main intention of the E-LOTOS extension was to enable modelling of the hardware layer of a system. Thus, in the specification, we can find such artifacts as interrupts, signals, and the ability to define events in time. With such extensions, E-LOTOS significantly expanded the possibility of using the algebra of processes, which is the starting point for the specification in this language.

It should be noted that the Alvis language has many features in common with E-LOTOS. First of all, Alvis as E-LOTOS is derived from process algebras. Alvis, like E-LOTOS, was intended to allow formal modelling and verification of distributed real-time systems. To meet the requirements, Alvis provides a concept of time and a delay operator. In contrast to E-LOTOS,

Alvis provides graphical modelling language. Moreover, Alvis toolkit supports a LTS graph generation, which significantly simplifies the formal verification of models.

System Modelling Language (SysML) [23] aims to standardize the process of a system specification and modelling. The original language specification was developed as an open source project on behalf of the International Council on Systems Engineering INCOS and the Object Management Group (OMG). SysML is a general purpose modelling language for systems engineering applications. In particular, it adds two new types of diagrams: requirement and parametric diagrams. The Alvis language has many common features with the SysML block diagrams and activity diagrams: ports, property blocks, communication among the blocks, hierarchical models. Unlike SysML, Alvis combines structure diagrams (block diagrams) and behaviour (activity diagrams) into a single diagram. In addition, Alvis defines formal semantics for the various artifacts, which is not the case in SysML. The concept of agent in Alvis corresponds with the SysML block definition. The formal semantics of Alvis allows you to create automated tools for verification, validation and runtime of Alvis models. SysML is a general-purpose systems modelling language, which covers most of the software engineering phases from analysis to testing and implementation. Alvis is focused on the structural model, the behavioural aspects of the system and formal verification of its properties. Its main area of application are distributed and embedded real-time systems. Alvis can be used as an extension to the software engineering process based on SysML.

Ada is the only ISO standard object-oriented concurrent real-time programming language [24], [21], [25]. Ada has been designed to address the needs of large-scale system development, especially for distributed and embedded systems. Ada is equipped with mechanisms for concurrent programming. The main concurrency constructs are tasks (processes), which model active entities, and protected objects, which model shared data structures that need to be accessed with mutual exclusion. Tasks can communicate with each other directly (using synchronous mechanism called *rendezvous*) or indirectly through protected objects. The Annex E of Ada defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program. A distributed system is an interconnection of one or more processing nodes and zero or more storage nodes. A few constructs in Ada were an inspiration while developing Alvis language. For example, protected objects have been used to define passive agents and the Ada *select statement* has been used to define the Alvis *select statement*. An Alvis model composed of few agents that work concurrently is similar to an Ada distributed system. Active agents can be treated as processing nodes, while passive agents as storage ones. The main difference between Alvis and Ada is the communication model. First of all, Alvis uses a simplified rendezvous mechanism with equal agents without distinguishing servers and clients. Moreover, Alvis does not support asynchronous procedure calling, a procedure uses an active agent context. Finally, Alvis in contrast to

Ada uses significantly less language statements and enables a formal verification.

SCADE [26] is a product developed by the Esterel Technologies company. It is a complex tool for developing a control software for embedded critical systems and for distributed systems. A system is described as an input to output transformation. In every cycle inputs are transformed to outputs according to a specification provided by functions: linear and discrete and state machine. SCADE allows system developer to choose from a large library of predefined components. The KCG code generator, which is a part of the SCADE suite, produces C code that has all the properties required for safety-critical software. SCADE also provides tools for checking system specification and verification of the developed model.

The Alvis approach is very different. The system in Alvis is represented as a set of communicating tasks which are continuously processing their instructions. Alvis also has no code generation phase, because it is an executable specification itself. Moreover, the system verification in Alvis is based on an LTS graph generation instead of specification-model consistency and statical code checking. SCADE and Alvis have also different approaches to types. The first one adopts simple static C language types due to specific runtime requirements, while the second one uses the Haskell type system.

1.2 Contents of the book

2

Communication diagrams

As it was already said, the key concept of Alvis is *agent* that denotes any distinguished part of the system under consideration with defined identity persisting in time. There are two kinds of agents in Alvis. *Active agents* perform some activities and are similar to tasks in Ada programming language [21], [25]. Each of them can be treated as a thread of control in a concurrent or distributed system. On the other hand, *passive agents* do not perform any individual activity, and are similar to protected objects (shared variables). Passive agents provide mechanism for the mutual exclusion and data synchronisation.

A communication diagram is a hierarchical graph whose nodes may represent both agents (*active* or *passive*) and parts of the model from the lower level. They are the only way in the Alvis modelling language, to point out agents that communicate one with another. Moreover, the diagrams allow programmers to combine sets of agents into modules that are also represented as agents (called *hierarchical ones*).

2.1 Nonhierarchical Communication Diagrams

Active agents are drawn as rounded boxes while passive ones as rectangles. An agent's identifier (name) is placed inside the corresponding shape. The first character of the identifier must be an upper-case letter. Other characters (if any) must be alphabetic characters, either upper-case or lower-case, digits, or an underscore. Alvis identifiers are case sensitive. Moreover, the Alvis keywords cannot be used as identifiers. Names of agents that are initially activated (represent running processes, see Chapter 5) are underlined. Graphical representation of Alvis agents (also hierarchical) is shown in Fig. 2.1.

An agent can communicate with other agents through *ports*. Ports are drawn as circles placed at the edges of the corresponding rounded box or rectangle. Each agent port must have a unique identifier (name) assigned, but ports of different agents may have the same identifier assigned. A port's identifier (name) is placed inside the corresponding rounded box/rectangle

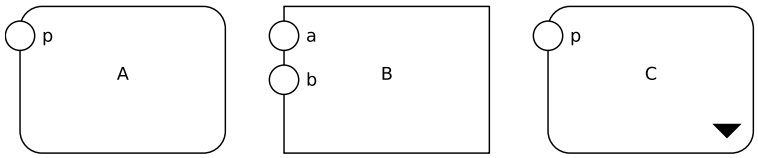


Fig. 2.1. Agents (from left): active, passive, hierarchical

next to the port. It must fulfill the same requirements as agents' identifiers but its first character must be a lower-case letter.

All ports have the same graphical representation regardless of the fact whether they are used as input or output ones. However, we can distinguished some subsets of ports based on the role they play in a model.

- *Border ports* are used for a communication with the environment of an embedded system. They are specified in the *environment* statement (see Chapter 3). Both active and passive agents can contain border ports. Ports that are not border ones, are called *internal ports*.
- *Procedure ports* are internal ports of passive agents that represent procedures – each of them is an argument of the *proc* statement (see Chapter 3).

Alvis agents can communicate with each other directly using the *connection mechanism* (communication channels). A *communication channel* is defined explicitly between two agents and connects two ports. A communication channel cannot connect two ports of the same agent. Communication channels are drawn as lines (or broken lines). An arrowhead points out the input port for the particular connection. Communication channels without arrowheads represent pairs of connections with opposite directions. Examples of communication channels are shown in Fig. 2.2.

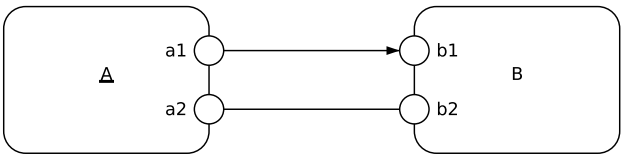


Fig. 2.2. One-way and two-way communication channels

Border ports can be used only for the communication with environment, thus they cannot used as elements of communication channels.

A connection between two active agents creates a synchronisation point between them. Connections between two active agents can be either one or two-way connection. An internal port with at least one two-way connection or at least one one-way connection such the port is the input port for the

connection is called an *input port*. Similarly, an internal port with at least one two-way connection or at least one one-way connection such the port is the output port for the connection is called an *output port*. Input and output border ports are distinguished based on the *in* and *out* clauses used in the *environment* statement (see Chapter 4).

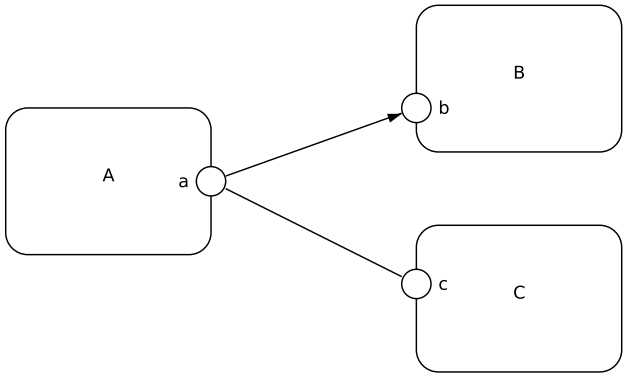


Fig. 2.3. Example of connections among active agents

An input port can be used an argument of the *in* statement. Similarly, an output port can be used an argument of the *out* statement (see Chapter 3). Let us consider the communication diagram shown in Fig. 2.3. Port b is an input but not output port. It means that it cannot be used to send any signals/values from agent B. On the other hand, port c is both an input and output port. Thus, it can play a double role in the model. It should be underlined that it is not necessary to used port c both as an argument of the *in* and *out* statements, but the two-way connection gives such an opportunity.

In the diagram represented in Fig. 2.3 signals or/and values can be send between agents A and C in any directions, while agent B can only collect signals or/and values sent by agent A. In other words, if agent A initialises a communication providing a signal/value to port a, the value can be collected (if suitable statements are used) by agent B or C. If agent A initialises a communication demanding a signal/value on port a, such a signal/value can be provided only by agent B.

Connections with passive agents *must be* one-way ones. There are to possibilities:

1. A connection between an active agent port and a passive agent procedure port – the active agent calls a procedure of the passive agent;
2. A connection between a passive agent non-procedure port and a passive agent procedure port – one passive agent calls, using a non-procedure port, a procedure of another passive agent.

Alvis procedures are divided into input and output ones. An input procedure takes one argument while an output one provides a single result. A port that represents an input procedure may be used in connections only as an input port. On the other hand, a port that represents an output procedure may be used only as an output port. In case of passive agents, non-procedures ports only can be both input and output ones.

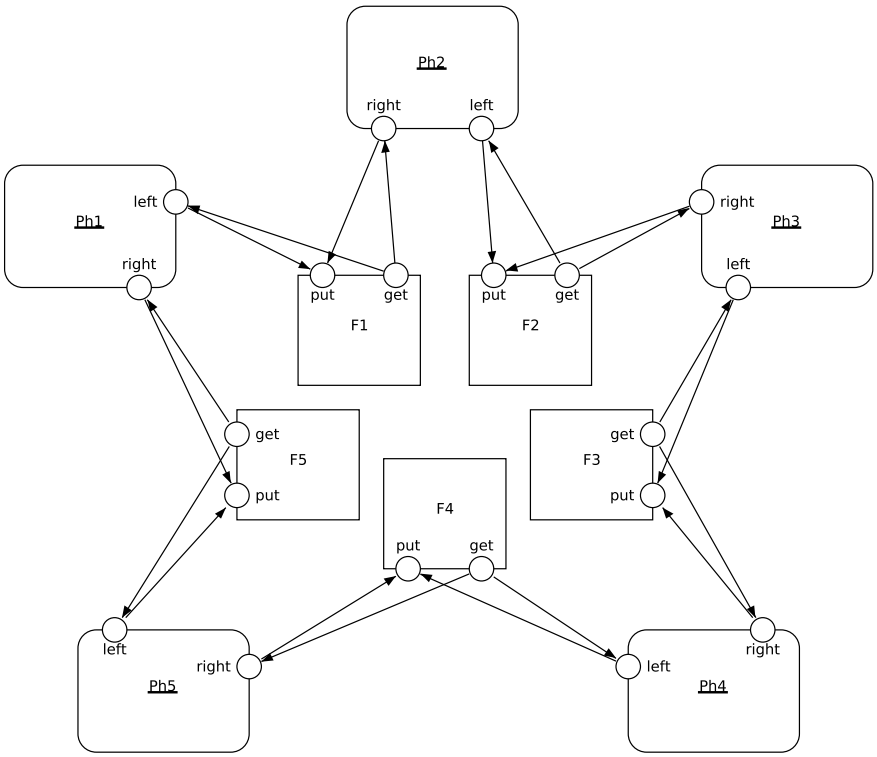


Fig. 2.4. Communication diagram of the dining philosophers problem

An example of a communication diagram for a dining philosophers problem is shown in Fig. 2.4. Let us recall the problem briefly. Five philosophers are sitting around a circular table. Each philosopher spends his life alternately thinking and eating. There is a large bowl of spaghetti in the center of the table. There are also five plates at the table and five forks set between the plates. Eating spaghetti requires the use of two forks. Each philosopher thinks. When he gets hungry, he picks up the two forks that are closest to him. If a philosopher gets the chance to pick up both forks, he eats for a while. After a philosopher finishes eating, he puts down the forks and starts to think.

Philosophers are represented by active agents with to ports used to to get and put right and left forks respectively. Forks are represented by passive agents with to procedures: *get* representing taking a fork from the table and *put* representing putting it back.

2.2 Hierarchical Communication Diagrams

For the effective modelling, Alvis communication diagrams enable distributing parts of a diagram across multiple subdiagrams called *pages*. Pages are combined using the so-called *substitution mechanism*. An agent on one level can be replaced by a page on the lower level. Such a substituted agent is called *hierarchical* one. On the other hand, a part of a communication diagram can be treated as a module and represented by a single agent on a higher level. Thus, communication diagrams support both *top-down* and *bottom-up* approaches.

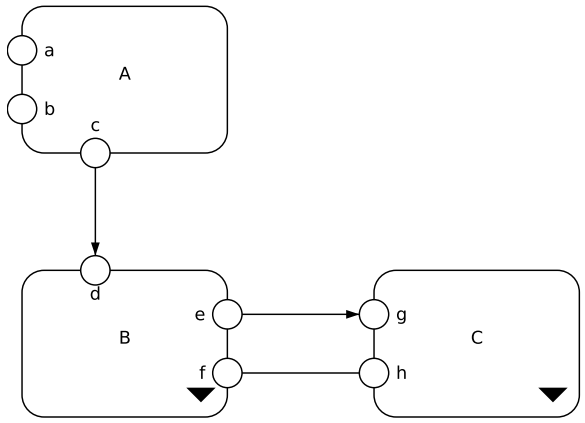


Fig. 2.5. Page D^1.

The substitution mechanism is based on a *binding function* π that maps ports of a hierarchical agent to ports with the same names (called *join ports*) on the corresponding page. If the function is a bijection the substitution is called a *simple* one. An example of the simple substitution is shown in Fig. 2.5 and 2.6. The page shown in Fig. 2.6 is assigned to agent B. It should be underlined that the join ports may be distributed among a few agents, but they cannot be connected with any ports.

A hierarchical agent can be replaced with its subpage using the following algorithm.

1. Remove the agent B from the page D^1 with all its connections.
2. Move the contents of the page D^2 onto the page D^1.

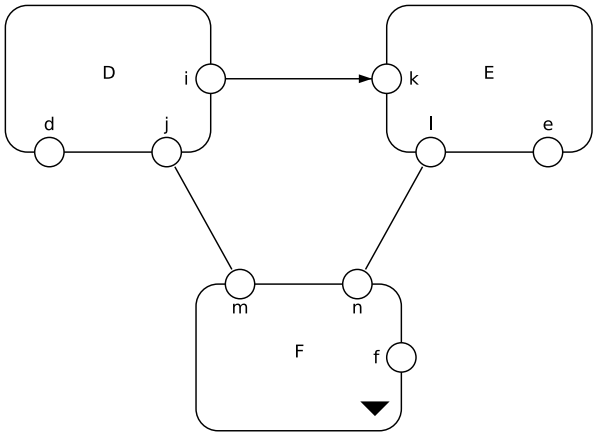


Fig. 2.6. Page D^2.

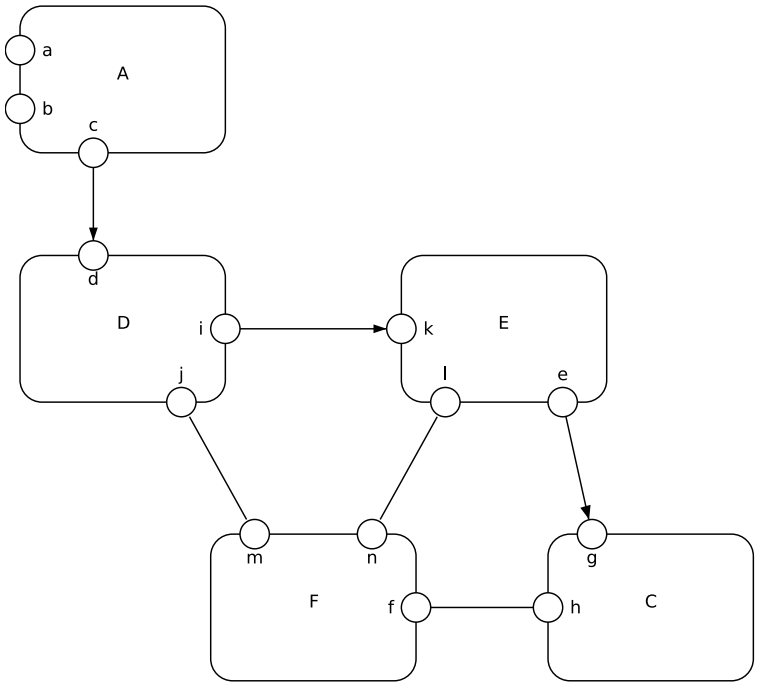


Fig. 2.7. Application of the simple substitution.

3. Add connections – If after removing of the agent B, from the page D^1, it has been removed a connection between ports $B.a$ and $X.p$, then we add

a connection between ports $X.p$ and $\pi(B.a)$ with the same direction as the removed one.

The result of application of the considered simple substitution is presented in Fig. 2.7. Elimination of all hierarchical agents provides an equivalent non-hierarchical communication diagram. The elimination does not influence a model properties, thus for theoretical considerations non-hierarchical models will be used.

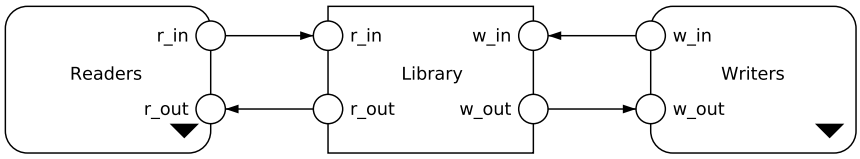


Fig. 2.8. Readers-Writers system – top level page of the communication diagram.

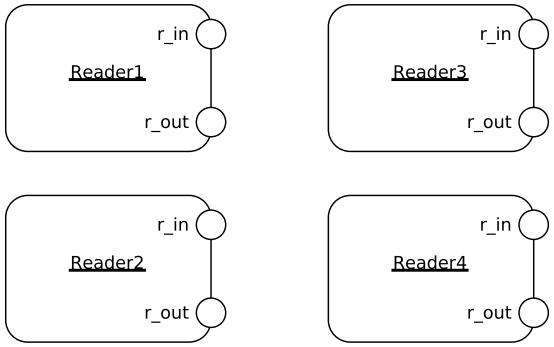


Fig. 2.9. Readers-Writers system – page *Readers*.

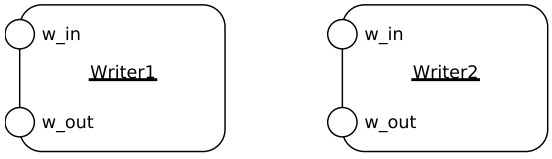


Fig. 2.10. Readers-Writers system – page *Writers*.

A port of a hierarhical agent may have assigned more than one join port on the subpage. In such a case the binding function π is not a bijection and the substitution is called an *extended* one. Of course all join ports that are connected with a port of the corresponding hierarchical agent must have the same names, and thus they must belong to different agents. (An agent cannot have two ports with the same name).

To represent extended substitutions let us consider the well-known readers-writers problem. We have two kinds of agents called *readers* and *writers* respectively that use a shared resource called *library* here. At most, one writer can use the library at any time, but a few readers can use it at the same time. The communication diagram for such a system with four readers and two writers is shown in Fig. 2.5, 2.9 and 2.10. The figures represent three pages that constitute a hierarchical communication diagram.

To present the way such pages are connected the so-called *page hierarchy graph* is used. The graph is shown in Fig. 2.11. Edges of a page hierarchy graph are labelled with names of hierarchical agents, while nodes represent pages in the corresponding model. The *System* node represents the top level page (so-called primary page) of the considered model. If a hierarchical communication diagram contains only one primary page it is a tree. Otherwise, it is a forest.

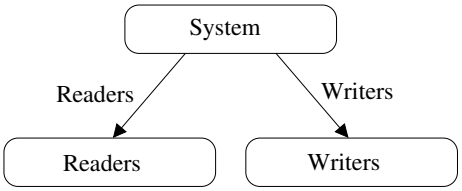


Fig. 2.11. Page hierarchy graph

Let us focus on the hierarchical agent *Readers* and its port *r_in*. There are four ports with the same name on the corresponding subpage. Thus, the port is connected with all of them. The result of replacing of agents *Readers* and *Writers* with their subpages is presented in Fig. 2.7. It is easy to see that we can change the number of readers or writers by changing the number of agents on corresponding pages. There is no need to change any connections in this model.

As it was already said, the elimination of hierarchical agents leads to a non-hierarchical diagram suitable for theoretical purposes. On the other hand, we can move a part of complex page into a new page and put a hierarchical agent into the original one. This reduces the size of a page an makes a model more readable. Moreover, as it was shown in the previous example, using hierarchy can reduce the number of connections significantly and makes a model more flexible – it is easier to make any changes in a model.

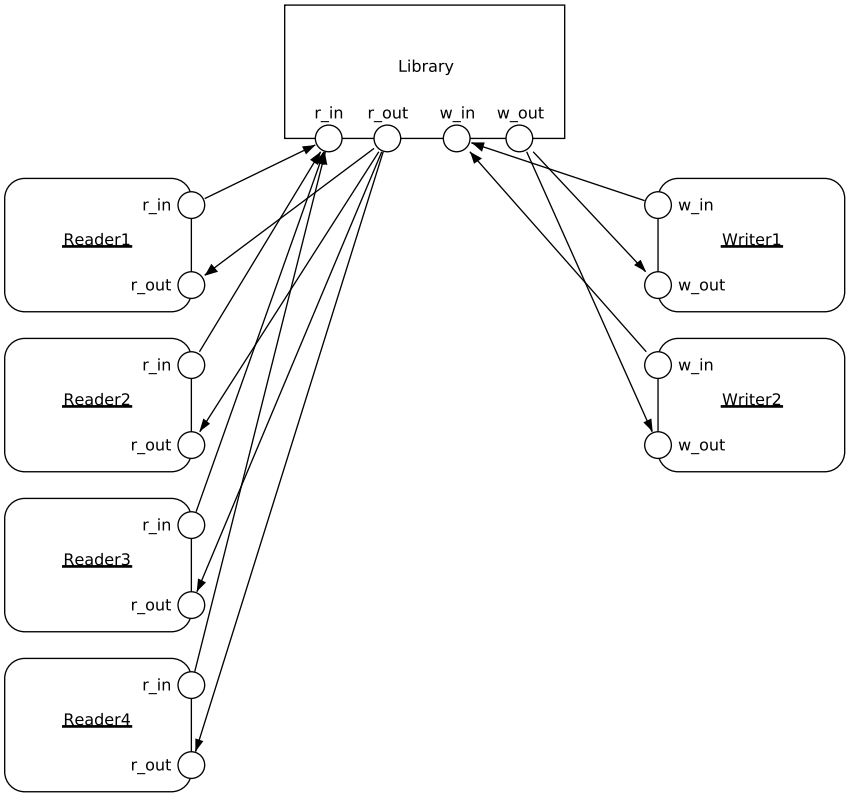


Fig. 2.12. Application of the extended substitution.

For more details on communication diagrams and their formal definitions see Chapter 5.

3

Code layer

This chapter provides description of the Alvis statements used in the code layer. The layer is used to describe the behaviour of individual agents in Alvis models. The layer uses Alvis behaviour description statements and some elements of the Haskell functional programming language. In spite of the fact that Alvis has its origin in CCS [10], [11], [9] and XCCS [20]. [12] process algebras, to make the language more convenient from the practical (engineering) point of view, algebraic equations and operators have been replaced with statements typical for high level programming languages. The code layer is used to:

- define data types used in the model under consideration,
- define functions for data manipulation,
- specify the considered embedded system environment,
- define behaviour of individual agents.

3.1 Code structure

The general structure of the code layer is presented in Listing 3.1. The *preamble* contains definitions of types, constants and functions used to manipulate data in a model. This part of the preamble is encoded in pure Haskell. Moreover, the preamble may contain specification of some environment activities that may be useful e.g. for an Alvis model simulation.

The *implementation* contains definitions of the agents' behaviour. This part is encoded using native Alvis statements, but the preamble contents is used to represent parameters values and to manipulate them. It contains at least one *agent block* as shown in Listing 3.2. It is possible to share one definition among a few agents. In such a case, a few agents' names are placed after the keyword *agent* separated by commas. If necessary, an agent's name is followed by its priority put inside round brackets. Priorities range from 0 to 9. Zero is the higher system priority.

```
-- Preamble:
--   types
--   constants
--   functions
--   environment specification

-- Implementation:
--   agents
```

Listing 3.1. Structure of the code layer

```
agent AgentName
{
  -- declaration of parameters
  -- agent body
}
```

Listing 3.2. Structure of an agent block

Both Haskell and Alvis are case sensitive languages. Haskell requires type names to start with an upper-case letter, and variable names to start with a lower-case letter. We follow Haskell footsteps. Moreover, Alvis requires agent names to start with an upper-case letter, and port names to start with a lower-case letter.

3.2 Types and parameters

Alvis uses the Haskell's type system. Types in Haskell are *strong*, *static* and can be automatically *inferred*. The *strong* property means that the type system guarantees that a program cannot contain errors coming from using improper data types, such as using a string as an integer. Moreover, Haskell does not automatically coerce values from one type to another. The *static* property means that the compiler knows the type of every value and expression at compile time, before any code is executed. Haskell's combination of strong and static typing makes it impossible for type errors to occur at runtime.

Selected basic Haskell types recommended to be used in Alvis are as follows:

- *Char* – Unicode characters.
- *Bool* – Values in Boolean logic (*True* and *False*).
- *Int* – Fixed-width integer values – The exact range of values represented as *Int* depends on the system's longest *native* integer.
- *Double* – Float-point numbers typically 64 bits wide and uses the system's native floating-point representation.

The most common composite data types in Haskell (and Alvis) are *lists* and *tuples* (see Listing 3.3). A *list* is a sequence of elements of the same

type, with the elements being enclosed in square brackets and separated by commas, while a *tuple* is a sequence of elements of possibly different types, with the elements being enclosed in parentheses and separated by commas. Haskell represents a text string as a list of *Char* values. Tuples containing different number of types of elements have disting types, as do tuple whose types appear in different orders.

```
[1,2,3,4]          -- type [Int]
['a','b','c']      -- type [Char] (String)
[True,False]       -- type [Bool]
(1,2)              -- type (Int,Int)
('a',True)         -- type (Char,Bool)
("abc",1,True)     -- type (String,Int,Bool)
```

Listing 3.3. Examples of Haskell composite data types

To make the source code more readable, one can introduce a synonym for an existing type as shown in Listing 3.4.

```
type AgentID = Int;
type InputData = (Int,Int,Int);
type TrafficSignal = (Char,Bool);
```

Listing 3.4. Synonyms for composite data types

A new data type is defined using the *data* keyword (see Listing 3.5). The identifier after the *data* keyword is the name of the new type, while the identifier after the = sign is called *value constructor* (data constructor). A value constructor is a special function, which name, as the name of type, must start with an uppercase letter. Types placed after a value constructor name are called *components* of the type. A value constructor is used to create a new value of the corresponding type as shown in the second line of Listing 3.5. A value constructor name can be the same as the type one.

```
data AgentDescription = AgentDesc Int String [String];
myAgent = AgentDesc 1 "Buffer" ["put","get"];
```

Listing 3.5. New composite data type

We can used more than one value constructor for one type. Such types in Haskell are called *algebraic data types*. Each value constructor is separated in the definition by the | sign. Each of an algebraic data type's value constructor

can take zero or more arguments. An algebraic data type can be used to define an enumeration type or a type with different variants of data (see Listing 3.6).

```
data Move = East | South | West | North;
type Point = (Double,Double);
data Shape = Circle Point Double
           | Rectangle Point Point;
```

Listing 3.6. Examples of Haskell algebraic data types

Constants are defined using parameterless Haskell functions as shown in Listing 3.7.

```
size = 10;
name = "Agent";
```

Listing 3.7. Examples of constants

The = symbol in Haskell code represents *meaning* – the name on the left is defined to be the expression on the right. This meaning of = is valid in the preamble. In the implementation part, the = symbol stands for the assignment operator.

Parameters are defined using the Haskell syntax. Each parameter is placed in a separate line. The line starts with a parameter name, then the :: symbol is placed followed by the parameter type. The type must be followed by the = symbol and the parameter initial value as shown in Listing 3.8.

```
size      :: Int          = 7;
queue     :: [Double]     = [];
inputData :: (Int, Char)  = (0, 'x');
```

Listing 3.8. Examples of parameters definitions

The assignment operator is also used as a part of the *exec* statement. The *exec* statement is the default one. Therefore, the *exec* keyword can be omitted. Thus, to assign a literal value 7 to an integer parameter x the first and the second statement presented in Listing 3.9 can be used. The assignment operator can also be followed by an expression. Alvis uses Haskell to define and manipulate data types. Thus, such an expression may take the form of a Haskell function call (see Listing 3.9).

```
exec x = 7;
x = 7;
x = x + 1;
x = rem x 3;
x = sqrt y;
```

Listing 3.9. Examples of using the *exec* statement

3.3 Communication statements

An agent can communicate with its outside world using *ports*. Each port can be used both as an input or an output one. The current role of a port is determined by two factors:

1. Connections to the port in the corresponding communication diagram – e.g. if a port p is used only as an input port for an one-way connection, it cannot be used as an output port.
2. Statements used in the code layer – e.g. if a port p is used only as an argument of the *in* statement, it is an input port even if all its connections are two-way ones.

Moreover, any communication through a port can be a pure synchronisation or a single value (probably of a composed type) can be sent/collected. A *pure synchronisation* is a communication without sending any values of parameters.

Alvis uses two statements for the communication. The *in* statement for collecting data and *out* for sending. Each of them takes a port name as its first argument and optionally a parameter name as the second. Parameters are not used for the pure communication. Syntax for these statements is given in Listing 3.10 (p stands for a port name and x stands for a parameter). The *in* statement assigns the collected value to its parameter, while the *out* statement sends the value of its parameter. Instead of a parameter, a constant can be used in the *out* statement.

```
in p;
in p x;
out p;
out p x;
```

Listing 3.10. Syntax of the *in*/*out* statements

There are two types of communication in Alvis. A communication between two active agents can be initialised by any of them. The agent that initialises it, performs the *out* statement to provide some information and waits for the second agent to take it, or performs the *in* statement to express its readiness to collect some information and waits until the second agent provides it.

On the other hand, a communication between an active and a passive agent can be initialised only by the former. Any procedure in Alvis uses only one either input or output parameter (or signal in case of parameterless communication). In case of an input procedure, an active agent calls the procedure using the *out* statement (and provides the parameter, if any, at the same time). If the corresponding passive agent is in the *waiting* mode and the procedure is accessible, the agent starts it in the active agent context. The passive agent collects the signal/parameter using the *in* statement, but it is not necessary to put the statement as the first procedure step. Similarly, in case of an output procedure, an active agent calls the procedure using the *in* statement. The passive agent returns the result using the *out* statement, but it is not necessary to put the statement as the last procedure step.

There is also possible to call a procedure of a passive agent from a procedure of another passive agent. This is similar to calling a procedure by an active agent. For more details see Chapter 5.

3.4 Loop statements and recursion

Alvis provides three kinds of *loop* statements. The first one is the most general *loop* statement as shown in Listing 3.11. It repeats its contents infinitely.

```
loop
{
  -- at least one statement inside
}
```

Listing 3.11. General *loop* statement

The second loop repeats its contents while the guard (g) is satisfied (see Listing 3.12). Guards are logical expressions, written in Haskell, placed inside round brackets. The loop is similar to the while loop in most languages – the guard is checked every time before entering the loop contents.

```
loop (g)
{
  -- at least one statement inside
}
```

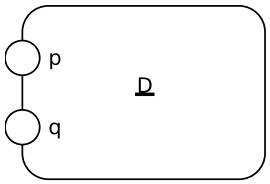
Listing 3.12. While *loop* statement

The last loop statement is the so-called *loop every* statement as shown in Listing 3.13. The loop repeats its contents every t time-units. The default time-unit in Alvis is millisecond.

```
loop (every t)
{
  -- at least one statement inside
}
```

Listing 3.13. *loop every* statement

Let us consider the agent D shown in Fig. 3.1. The agent collect an integer through its port p, doubles the parameter x value and sends its value through port q. This sequence of statements is repeated infinitely.




```
agent D
{
  x :: Int = 0;
  loop
  {
    in p x;
    x = 2 * x;
    out q x;
  }
}
```

Fig. 3.1. Graphical representation and implementation for agent D

Recursion is another mechanism used for looping in the Alvis language. Two language concepts are used for this purpose: *labels* and the *jump* statement. Labels in Alvis are identifiers followed by a colon. A label must start with a lower case letter. The statement is composed of the *jump* key word and a label name (without a colon). It is necessary to put at least one statement after a label. In other words, a label cannot be followed by a closing curly bracket. The *jump* statement is the key statement for translating algorithms from CCS to Alvis. For example, the behaviour of the agent D from Fig. 3.1 can be also defined as shown in Listing 3.14.

```
agent D
{
  x :: Int = 0;
  go:
    in p x;
    x = 2 * x;
    out q x;
    jump go;
}
```

Listing 3.14. Definition of agent infinite behaviour with the *jump* statement

3.5 Alternatives

Alvis provides a typical *if else* statement with optional *else* and *elseif* clauses. The general syntax of the conditional statement is shown in Listing 3.15 – $g1$, $g2$ and $g3$ stand for guards.

```
if (g1)
{
  -- at least one statement inside
}
elseif (g2)
{
  -- at least one statement inside
}
elseif (g3)
{
  -- at least one statement inside
}
-- ...
else
{
  -- at least one statement inside
}
```

Listing 3.15. Syntax of the conditional statement

In order to allow for the description of agents whose behaviour may follow different alternative paths, Alvis offers the *select* statement (see Listing 3.16). The statement is similar to the basic *select* statement from the Ada programming language [21], but there is no distinction between a server and a client. The statement may contain a series of *alt* clauses called *branches*. Each branch may be guarded. These guards divide branches into *open* and *closed* ones. A branch is called *open*, if it does not have a guard attached or its guard evaluates to *True*. Otherwise, a branch is called *closed*. To avoid indeterminism, if more than one branch is open the first of them is chosen to be executed.

If all branches are closed, the corresponding agent is postponed until at least one branch is open.

```
select {
  alt (g1)
  {
  -- at least one statement inside
  }
  alt (g2)
  {
  -- at least one statement inside
  }
  alt (g3)
  {
  -- at least one statement inside
  }
  -- ...
}
```

Listing 3.16. Syntax of the *select* statement

To postpone an agent for some time the *delay* statement is used. The statement is composed of the *delay* key word and a time period (default in milliseconds). The statement is also used to define time-outs. A branch may contain the *delay* as its guard (see Listing 3.17). In such a case, the third branch will be open after t time-units. Thus, if all branches are closed, the corresponding agent waits t time-units and follows the last branch. However, if at least one branch is open before the delay goes by, then the delay is cancelled. An example of code layer with a time-out functionality can be found in Section 4.2.

```
select {
  alt (g1) {...}
  alt (g2) {...}
  alt (delay t) {...}
}
```

Listing 3.17. Syntax of the *select* statement with a time-out

Another typical guard is composed of the *ready* function provided by Alvis system layers. The function takes as its argument a list of ports names of a given agent (with *in* or *out* keywords to point out the communication direction), and returns *True* only if at least one of these ports can be used for a communication immediately. In other words, the function returns true if:

- for an input border port (see Chapter 4), the port is accessible and a signal/value can be collected from the port immediately;
- for an output border port, the port is accessible and a signal/value can be send through the port immediately;
- for an input internal port (see Chapter 4), another agent has already provided a value to the port;
- for an output internal port, another agent has already required a signal/-value from that port.

```
select {
  alt (ready [in(a)]) {...}
  alt (ready [in(b)]) {...}
}
```

Listing 3.18. Example of the *select* statement with guards using the *ready* function

Let us consider the piece of code presented in Listinf 3.18. Suppose, both ports are internal one. After entering the *select statement*, the agent waits (if necessary) until another agent provides a signal/value to any of the ports *a* or *b*. After providing a signal/value to one of these ports, the corresponding branch is chosen.

3.6 Procedures

Passive agents in Alvis provide a mechanism for the mutual exclusion and data synchronisation. They are based on protected objects from the Ada programming language. Ports of a passive agent can be used as:

- *procedure ports* – the name of such a port is treated as a name of a procedure;
- border ports – such ports can be used to communicate with the environment inside procedures of the agent;
- internal ports – such ports are connected with another passive agents and are used to call other procedures inside procedures of the considered agent.

A communication with a passive agent is treated as a procedure call. It can be initialised either by an active agent or by a passive one from inside of its procedure. In case of an input procedure (a parameter is sent to the corresponding passive agent), it is called with the *out* statement. After a procedure is started, its performs its statements. It is necessary to put the *in* statement as one of them – the statement is used to collect the parameter, but it is not necessary to put the statement as the first procedure step. Similarly, in case of an output procedure, it is called with the *in* statement. It is necessary to put the *out* statement as one of its statements. It is used to provide the

result, but it is not necessary to put the statement as the last procedure step. In any case, a procedure is finished if its last statement has been performed or the *exit* statement has been performed. The *exit* statement can be used only after the *in/out* statement that corresponds to the procedure call.

```
proc (g) p { ... }
```

Listing 3.19. Syntax of the *proc* statement

The general syntax of the *proc* statement is shown in Listing 3.19 – g and p stand for the procedure guard and port respectively. A procedure is accessible if its guard evaluates to *True* and the agent is in the *Waiting* mode.

```
agent Ph1, Ph2, Ph3, Ph4, Ph5 {
  loop {
    in right;
    in left;
    out right;
    out left;
  }
}

agent F1, F2, F3, F4, F5 {
  taken :: Bool = False;

  proc (taken == False) get {
    taken = True;
    out get;
  }

  proc (taken == True)  put {
    taken = False;
    in put;
  }
}
```

Listing 3.20. Example of the *select* statement with guards using the *ready* function

Let us consider the model of dining philosophers (see Section 2.1 and Fig. 2.4). The code later for the model is shown in Listing 3.20. All philosophers share the same behaviour as forks do. Each fork provides two procedures *get* and *put*, but they cannot be accessible at the same time.

The *exit* statement can be also used inside an active agent code. In such a case, after performing the statement, the active agent finishes its activity.

Table 3.1. Alvis statements

Statement	Description
`cli`	Turns off the interrupts handlers.
`critical {...}`	Defines a critical section.
`delay t`	Delays an agent execution for a given number of time-units.
`exec x = expression`	Evaluates the expression and assigns the result to the parameter; the *exec* keyword can be omitted.
`exit`	Terminates an active agent or a passive agent procedure.

`if (g1) {...}`
`elseif (g2) {...}`
`elseif (g3) {...}`
`...`
`else {...}`
	Conditional statement.

`in p` `in p x`	Collects a signal/value through port p.
`jump label`	Transfers the control to the line of code identified with the *label*.
`jump far A`	Transfers the control to agent `A`.

`loop {...}`
`loop (g) {...}`
`loop (every t) {...}`
	Infinite loop.
	Repeats execution of the contents while the guard if satisfied..
	Repeats execution of the contents every t time-units.

`null`	Empty statement.
`out p` `out p x`	Sends a signal/value through the port p.
`proc (g) p {...}`	Defines the procedure for port p of a passive agent. The guard is optional.

`select {`
`  alt (g1) {...}`
`  alt (g2) {...}`
`  alt (g3) {...}`
`  ...`
`}`
	Selects one of alternative choices.

`start A`	Starts the agent A if it is in the *Init* state, otherwise do nothing.
`sti`	Turns on the interrupts handlers.

3.7 Other statements

An Alvis model contains a fixed number of agents. In other words, there is no possibility to create or destroy agents dynamically. If an active agent starts in the *init* mode, it is inactive until another agent activates it with the *start* statement. Active agents that are initially activated are distinguished in the communication diagram – their names are underlined.

Empty curly brackets are not allowed in Alvis. If necessary, the empty statement *null* can be put inside.

Border ports may be used to model interrupts handling in Alvis. It is possible to block collecting signals/values from the environment. The *cli* statement is used to block border ports (turn off the interrupts handlers), while *sti* is used to turn it on again.

The *critical* section is used to define a set of statements that cannot be interrupted. The statement is useful when α^1 system layer is used.

The last Alvis statement is *jump far*. It is used to transfers the control to agent given as the parameter of the statement. The *jump far* statement is useful for modelling scheduling functions on your own. The complete set of Alvis statements is given in Table 3.1. To simplify the syntax, the following symbols have been used. A stands for an agent name, p stands for a port name, x stands for a parameter, g, $g1$, $g2\ldots$ stand for guards (Boolean conditions), e stands for an expression and t stands for time-units.

4

Specification of model environment

An embedded system is one that is a part of a larger one. It is surrounded by other parts of the larger system that constitute the embedded system environment. Such an embedded system collects inputs that come from its environment (from sensors) and provide outputs that go to the environment (to controllers). To verify an embedded system formally we cannot separate it from its environment. Thus, if a formal language is used e.g. Petri nets [1], [27], [28], time automata [7], process algebra [10] etc., an embedded system model must include both the system and its environment. As a result of such a situation a model is often significantly more complex and the state explosion problem makes a formal verification difficult or even impossible.

Alvis has been designed especially for embedded systems and one of its main advantages is a possibility of a flexible specification of a behaviour of an embedded system's environment. Instead of designing the environment as a part of the model it is possible to specify signals generated or collected by the environment in a very simple way. This approach is also very useful from the analysis point of view. We can freely move the system border. In other words, we can start with modelling a very small part of the considered system in the first stage, moving all other parts to the environment. Then we can add more agents in the next stages. A state of a model is a sequence of agents' states. Moving some agents to the embedded system environment can reduce the model states space significantly.

4.1 Border ports

Alvis agents may contain ports that are not used in any connection. Such ports are called *border ports* and are used for a communication with the considered system environment. Border ports can be used both for collecting or sending some information to the embedded system environment. Properties of border ports are specified in the code layer preamble with the use of the *environment* statement. Each border port used as an input one is described with at least

one *in* clause. Similarly, each border port used as an output one is described with at least one *out* clause. Each clause inside the *environment* statement contains the following pieces of information:

- *in* or *out* key word,
- the border port name,
- a type name or a list of permissible values to be sent through the port,
- a list of time points, when the port is accessible,
- optionally some modifiers: *durable, queue, signal.*

It should be underlined that border ports are ports without any connections and specified inside the *environment* clause. Other ports will be called *internal* ones. A model may contain an internal port without any connections, but such a port is not specified inside the *environment* clause.

```
in a [1..4] [];
in b [1..4] (map (100*) [1..]);
in c [1..4] (map (100*) [1..]) signal;
in d [1..4] (map (100*) [1..]) durable;
in e [1..4] (map (100*) [1..]) queue;
in f [1..4] (map (100*) [1..]) signal durable;
in g [1..4] (map (100*) [1..]) signal queue;
```

Listing 4.1. Examples of input border ports' specification

Let us focus on the description of input border ports presented in Listing 4.1. Signals directions are considered from an embedded system point of view, thus all considered ports are used to send information from an environment to the corresponding embedded system. In each case, one of the values $1, 2, 3, 4$ (at random) can be collected through a port. However, the ports differ about the time points when values are accessible.

`a` A value from the port can be collected at any time point. An agent that performs the *in* statement receives the value immediately (never waits for it). Such border ports are useful for a modelling of input sensors whose values can be read at any time.

`b` Every 100 time-unit (by default milliseconds) a value is provided by the environment via the port. If none agent waits for it (*waiting* mode), the value is lost.

`c` The port behaves similar to the `b` one, but the signal may not be provided.

`d` Every 100 time-unit a value is provided by the environment via the port. The value is accessible for the corresponding embedded system until an agent collects it. If while waiting for a collecting the value, another one is sent via the port, the previous one is overwritten.

`e` The port behaves similar to the `d` one, but if while waiting for a collecting the value, another one is sent via the port, it is put into a FIFO queue.

`f` The port behaves similar to the `c` one, but the value is accessible for the corresponding embedded system until an agent collects it or it is overwritten.

`g` The port behaves similar to the `f` one, but the values are put into a FIFO queue.

The specifications of ports $b, \ldots, g$ ports use the Haskell `map` function and an infinite list. For more details (if necessary) see [17].

If a border port is used for a parameterless communication, then the first list is empty. If different kinds of signals can be sent through a border port, then more than one *in* clause must be used, but the time points list must be disjoint.

```
in a [] [1,3,6]
in a Bool [2,4,6,8]
```

Listing 4.2. Examples of a wrong border port specification

Let us consider the specification of a border port a presented in Listing 4.2. Such a specification is not allowed because the ambiguity that appears 6 time-units after the system start – at the same time a value-less signal and a Boolean value should be generated.

```
out a [1..4] [];
out b [1..4] (map (100*) [1..]);
out c [1..4] (map (100*) [1..]) signal;
out d [1..4] (map (100*) [1..]) durable;
out e [1..4] (map (100*) [1..]) queue;
out f [1..4] (map (100*) [1..]) signal durable;
out g [1..4] (map (100*) [1..]) signal queue;
```

Listing 4.3. Examples of output border ports' specification

Let us focus on the description of output border ports presented in Listing 4.1.

`a` Any of the values $1, 2, 3, 4$ can be sent through the port at any time point. An agent that performs the *out* statement sends the value immediately (never waits for the port accessibility).

`b` Any of the values $1, 2, 3, 4$ can be sent through the port every 100 time-units, but if the system is not ready to send a value then the opportunity is lost.

`c` The port behaves similar to the `b` one, but the accessibility of the port is not guaranteed.

`d` Any of the values $1, 2, 3, 4$ can be sent through the port every 100 time-units, but if the system is not ready to send a value then the environment waits for it.

`e` The port behaves similar to the `d` one, but the opportunities are put into a FIFO queue.

`f` The port behaves similar to the `d` one, but the accessibility of the port is not guaranteed.

`g` The port behaves similar to the `e` one, but the accessibility of the port is not guaranteed.

If a border port is used both as an input and an output one, then it must be described both with the *in* and *out* clauses. If different kinds of signals can be sent through a border port, then more than one *in* or *out* clause can be used, but the time points lists must be pairwise disjoint.

It should be underlined that only the *signal* modifier should be used in the final model of an embedded system. Other modifiers are defined mainly for the verification purposes, if reduced models are considered.

Border ports must have unique names in a model. The same name of a border port used twice means that two agents use the same border port.

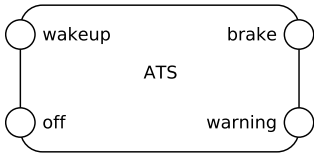
4.2 ATS system example

Trains could not run safely without signalling devices. Some automatic systems are used to transfer signals directly to a driver cab. A driver must always obey such a signal, but the possibility of human error can cause serious accidents. As it was already said, Automatic Train Protection (ATP) systems are used to guarantee a train safety even if a driver is not capable of controlling the train. Furthermore, computer systems can drive a train without a human support. The Automatic Train Stop considered here is used to check whether a driver controls the train. In the ATS system, a light signal is turned on every 60 seconds to check whether a driver controls the train. If the driver fails to acknowledge the signal within 6 seconds, a sound signal is turned on. Then, if the driver does not disactivate the signals within 3 seconds, using the acknowledge button, the emergency brakes are applied to stop the train automatically. An RTCP-net (Petri net) model of such a system is presented in [28]. To verify the Petri net model of the ATS system, it was necessary to add places and transitions that simulate the driver behaviour.

The default time unit in Alvis is 1 millisecond. However, due to the specific features of the system under consideration, we will use 1 second as the basic time unit. As a result of this assumption, we will omit durations of steps execution. A single step in this example takes about 1 or 2 milliseconds, so they do not influence the system properties.

We start with a model that contains only one agent called *ATS*. Other elements: the cab console, timer, brake etc. are elements of our system envi-

ronment. The model is shown in Fig. 4.1. Comments contain the numbers of steps.




```
environment {
  in wakeup    [] (map (60*) [0..]) durable;
  in off       [] [1..] signal;
  out warning [0,1,2] [];
  out brake    [] [];
}

agent ATS {
  loop {                                  -- 1
    in wakeup;                            -- 2
    out warning 1;                        -- 3
    select {                              -- 4
      alt (ready [in(off)]) {
        in off;                           -- 5
        out warning 0; }                  -- 6
      alt(delay 6) {
        out warning 2;                    -- 7
        select {                          -- 8
          alt (ready [in(off)]) {
            in off;                       -- 9
            out warning 0; }              -- 10
          alt (delay 3) {
            out brake;                    -- 11
            exit; }                       -- 12
        } } } } }
```

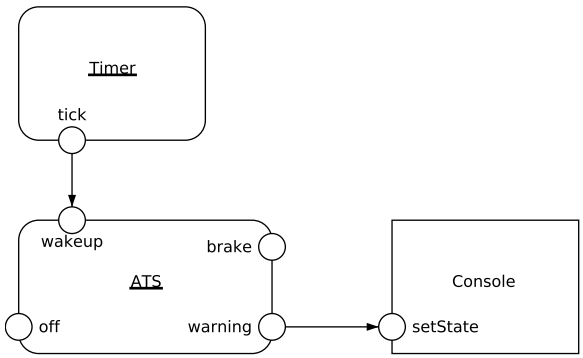
Fig. 4.1. ATS system – model 1

In the considered example all ports are border ones. However, the model is already suitable to verify properties of the *ATS* agent. The *off* signal can be generated any one second, but it does not influence the system behaviour, if it is generated before the system is waked up. It is possible to specify the *off* port behaviour in a more sophisticated way using Haskell functions as shown in Listing 4.4. In the presented example, the *off* signal may be generated only for 10 s after the *wakeup* signal.

```
offlist p0 p1 p2 n k
  | k <= n     = (p1 + k * p2)
               : offlist p0 p1 p2 n (k + 1)
  | otherwise = offlist p0 (p0 + p1) p2 n 1
offlist' = offlist 60000 61000 1000 10 1
```

Listing 4.4. Haskell function for generation time points list

In the second approach, we decided to treat the driver *console* as a part of the embedded system. Moreover, the system contains a timer that wakes it up every 60 s. The model is shown in Fig. 4.2.




```
environment {
  in off [] signal;
  out brake [] []; }

agent ATS {
  -- ...
}
agent Timer {
  loop (every 6000) {
    out tick; }
}
agent Console {
  state ::Int = 0;
  proc setState { in setState state ; }
}
```

Fig. 4.2. ATS system – model 2

The second model contains only two border ports. In spite of the fact that two new agents have been included into the model, the definition of the *ATS* agent is still the same. The driver console is represented by the *Console* passive agent with a single procedure used to set the console state. The *Timer*

agent is an active one with a *loop every* statement. The agent sends the *tick* signal every 60 s.

The presented examples of ATS systems allow one to find out the usefulness of the Alvis language for the design of embedded systems. The possibility of moving borders of an embedded system environment allows designers to develop a system increasing the number of its details in subsequent stages. Thus, we can design a small part of the target system and test its behaviour or verify it in a formal way. Then, a more detailed version of such a system can be designed and we check whether the new version is compatible with the old one.

On the other hand, the presented approach can be useful for verification of more complex models with a very big state space. We can divide such a model into a set of subsystems and verify each of them separately. For each subsystem, we treat other parts of the model as the subsystem environment.

5

Formal description of models

This chapter provides formal definitions of basic Alvis concepts like non-hierarchical and hierarchical communication diagrams, syntax and analysis operations etc. At the end of the chapter, a formal definition of an Alvis model is given. All these definitions are necessary to provide formal semantic for all Alvis statements in the following chapters.

5.1 Non-hierarchical diagrams

Let **A** denote an Alvis model (with a non-hierarchical communication diagram). Graphical and code layers of a model are closely related one to the other. Each active and passive agent from a communication diagram is described in the corresponding code layer and vice versa.

An agent can communicate with other agents through *ports*. Each agent port must have a unique identifier (name) assigned, but ports of different agents may have the same identifier assigned. Thus, each port in a model is identified using its name and its agent name. For simplicity, we will used the so-called *dot notation* – $X.p$ denotes port p of agent X.

Let $\mathcal{P}(X)$ denote the set of ports of an agent X. We can distinguish the following subsets of the set $\mathcal{P}(X)$:

- $\mathcal{P}_{border}(X)$ denotes the set of *border ports* of agent X i.e. ports that are specified in the *environment* statement.
- $\mathcal{P}_{internal}(X) = \mathcal{P}(X) - \mathcal{P}_{border}(X)$ denotes the set of *internal ports* of agent X.
- $\mathcal{P}_{in}(X)$ denotes the set of *input ports* of agent X. An input border port is a border port with at least one *in* specification. An input internal port is an internal port with at least one one-way connection leading to this port or with at least one two-way connection.
- $\mathcal{P}_{out}(X)$ denotes the set of *output ports* of agent X. An output border port is a border port with at least one *out* specification. An output internal port

is an internal port with at least one one-way connection leading from this port or with at least one two-way connection.

- $\mathcal{P}_{unc}(X) = \mathcal{P}_{internal}(X) - (\mathcal{P}_{in}(X) \cup \mathcal{P}_{out}(X))$ denotes the set of *unconnected ports.*
- $\mathcal{P}_{proc}(X) \subseteq \mathcal{P}_{internal}(X)$ denotes the set of procedure ports of agent X (for passive agents only) i.e. ports with defined the *proc* statement (names of such ports are treated as names of procedures).

For a set of agents W we define: $\mathcal{P}(W) = \sum_{X \in W} \mathcal{P}(X)$, $\mathcal{P}_{border}(W) = \sum_{X \in W} \mathcal{P}_{border}(X)$, etc. Moreover, let $\mathcal{P}$ denote the set of all model ports, $\mathcal{P}_{border}$ denote the set of all model border ports, etc.

Let $\mathcal{N}(X)$ denote the set of ports names of agent X, and $\mathcal{N}(W) = \sum_{X \in W} \mathcal{N}(X)$. For example, if a diagram contains only agents: X_1 with port p and X_2 also with port p, then $\mathcal{P} = \{X_1.p, X_2.p\}$, and $\mathcal{N}(\mathcal{P}) = \{p\}$.

Definition 5.1. *A* Non-hierarchical communication diagram *is a triple* $D = (\mathcal{A}, \mathcal{C}, \sigma)$*, where:*

- $\mathcal{A} = \{X_1, \ldots, X_n\}$ *is the set of* agents *consisting of two disjoint sets,* $\mathcal{A}_A$*,* $\mathcal{A}_P$ *such that* $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$*, containing* active *and* passive *agents respectively.*
- $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ *is the* communication relation*, such that*

$$\forall X \in \mathcal{A}\colon\ (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C} = \emptyset, \tag{5.1}$$

$$(\mathcal{P}_{border} \times \mathcal{P}) \cap \mathcal{C} = \emptyset \wedge (\mathcal{P} \times \mathcal{P}_{border}) \cap \mathcal{C} = \emptyset, \tag{5.2}$$

$$\mathcal{P}_{proc} \cap \mathcal{P}_{in} \cap \mathcal{P}_{out} = \emptyset, \tag{5.3}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_A) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow q \in \mathcal{P}_{proc}, \tag{5.4}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_A)) \cap \mathcal{C} \Rightarrow p \in \mathcal{P}_{proc}, \tag{5.5}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow$$
$$\Rightarrow (p \in \mathcal{P}_{proc} \wedge q \notin \mathcal{P}_{proc}) \vee (q \in \mathcal{P}_{proc} \wedge p \notin \mathcal{P}_{proc})\,. \tag{5.6}$$

Each element of the relation $\mathcal{C}$ *is called a* connection *or a* communication channel*.*

- $\sigma\colon \mathcal{A}_A \to \{False, True\}$ *is the* start function *that points out initially activated agents.*

Let us focus on the conditions from the abobe definition.

- (5.1) – A connection cannot be defined between two ports of the same agent.
- (5.2) – Border ports cannot be connected with any ports.
- (5.3) – Procedure ports are either input or output ones.
- (5.4), (5.5) – A connection between an active and a passive agent must be a procedure call.

[0] We will use two notations to denote ports in equations. A single lower-case letter e.g. p denotes a port p of some agent. If it is necessary to point out both a port name and agent name, the dot notation will be used e.g. $X.p$.

- (5.6) – A connection between two passive agents must a procedure call from a non-procedure port. From conditions (5.3)-(5.6) it follows that any connection with a passive agent must be one-way connection.

The start function σ makes possible delaying activation of some agents – We can make them active later with the *start* statement. Names of agents that are initially activated (represent running processes) are underlined in a communication diagram.

Let us focus on the ATS system model shown in Fig. 4.2. The model elements are defined as follows:

- $\mathcal{A}_A = \{ATS, Timer\}$,
- $\mathcal{A}_P = \{Console\}$,
- $\mathcal{P}_{border} = \{brake, off\}$,
- $\mathcal{P}_{internal} = \{setState, tick, wakeup, warning\}$,
- $\mathcal{P}_{in} = \{off, setState, wakeup\}$,
- $\mathcal{P}_{out} = \{brake, warning\}$,
- $\mathcal{P}_{proc} = \{setState\}$,
- $\mathcal{C} = \{(Timer.tick, ATS.wakeup), (ATS.warning, Console.setState)\}$,
- $\sigma(ATS) = \sigma(Timer) = True$.

5.2 Hierarchical communication diagrams

A communication diagram can be treated as a module and represented by a single agent at the higher level. Thus, communication diagrams support both *top-down* and *bottom-up* approaches. For the effective modelling Alvis communication diagrams enable distributing parts of a diagram across multiple subdiagrams called *pages*. Pages are combined using the so-called *substitution mechanism*. An agent at one level can be replaced by a page on the lower level, which usually gives a more precise and detailed description of the subsystem represented by the agent. Such a substituted agent is called *hierarchical* one. All ports of a hierarchical agent must appear on the corresponding subpage. A hierarchical communication diagram consists of a set of pages.

Hierarchical agents represent submodels. They are not defined in the model code layer. From the model verification point of view, the equivalent flat communication diagram is taken under consideration. We divide ports of hierarchical agents into three subsets based on the connections defined in the model: $\mathcal{P}_{in}(X)$, $\mathcal{P}_{out}(X)$, and $\mathcal{P}_{unc}(X)$. Ports of hierarchical agents cannot be defined as border or procedure ones.

Definition 5.2. *A* page *in a hierarchical communication diagram is a triple* $D^i = (\mathcal{A}^i, \mathcal{C}^i, \sigma^i)$*, where:*

- $\mathcal{A}^i = \{X_1^i, \ldots, X_n^i\}$ *is the set of* agents *with subsets of* active agents $\mathcal{A}_A^i$*,* passive agents $\mathcal{A}_P^i$*, and* hierarchical agents $\mathcal{A}_H^i$*, such that* $\mathcal{A}^i = \mathcal{A}_A^i \cup \mathcal{A}_P^i \cup \mathcal{A}_H^i$*, and* $\mathcal{A}_A^i$*,* $\mathcal{A}_P^i$*,* $\mathcal{A}_H^i$ *are pairwise disjoint.*

- $\mathcal{C}^i \subseteq \mathcal{P}^i \times \mathcal{P}^i$, *where* $\mathcal{P}^i = \sum_{X \in \mathcal{A}^i} \mathcal{P}(X)$, *is the* communication relation, *such that:*

$$\forall X \in \mathcal{A}^i \colon (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C}^i = \emptyset, \tag{5.7}$$

$$\left(\mathcal{P}^i_{border} \times \mathcal{P}^i\right) \cap \mathcal{C}^i = \emptyset \wedge \left(\mathcal{P}^i \times \mathcal{P}^i_{border}\right) \cap \mathcal{C}^i = \emptyset, \tag{5.8}$$

$$\mathcal{P}^i_{proc} \cap \mathcal{P}^i_{in} \cap \mathcal{P}^i_{out} = \emptyset, \tag{5.9}$$

$$\mathcal{P}^i_{proc} \cap \mathcal{P}(\mathcal{A}^i_H) = \emptyset \wedge \mathcal{P}^i_{border} \cap \mathcal{P}(\mathcal{A}^i_H) = \emptyset, \tag{5.10}$$

$$(p,q) \in \left(\mathcal{P}(\mathcal{A}^i_A) \times \mathcal{P}(\mathcal{A}^i_P)\right) \cap \mathcal{C}^i \Rightarrow q \in \mathcal{P}^i_{proc}, \tag{5.11}$$

$$(p,q) \in \left(\mathcal{P}(\mathcal{A}^i_P) \times \mathcal{P}(\mathcal{A}^i_A)\right) \cap \mathcal{C}^i \Rightarrow p \in \mathcal{P}^i_{proc}, \tag{5.12}$$

$$(p,q) \in \left(\mathcal{P}(\mathcal{A}^i_P) \times \mathcal{P}(\mathcal{A}^i_P)\right) \cap \mathcal{C}^i \Rightarrow$$
$$\Rightarrow \left(p \in \mathcal{P}^i_{proc} \wedge q \notin \mathcal{P}^i_{proc}\right) \vee \left(q \in \mathcal{P}^i_{proc} \wedge p \notin \mathcal{P}^i_{proc}\right), \tag{5.13}$$

$$(p,q) \in \left(\mathcal{P}(\mathcal{A}^i_P) \times \mathcal{P}(\mathcal{A}^i_H)\right) \cap \mathcal{C}^i \Rightarrow (q,p) \notin \mathcal{C}^i, \tag{5.14}$$

$$(p,q) \in \left(\mathcal{P}(\mathcal{A}^i_H) \times \mathcal{P}(\mathcal{A}^i_P)\right) \cap \mathcal{C}^i \Rightarrow (q,p) \notin \mathcal{C}^i. \tag{5.15}$$

  Each element of the relation $\mathcal{C}^i$ *is called a* connection *or a* communication channel.
- $\sigma^i \colon \mathcal{A}^i_A \to \{False, True\}$ *is the* start function *that points out initially activated agents.*

Let us focus on the conditions from the abobe definition.

- (5.7) – A connection cannot be defined between two ports of the same agent.
- (5.8) – Border ports cannot be connected with any ports.
- (5.9) – Procedure ports are either input or output ones.
- (5.10) – Hierarchical agents cannot have border or procedure ports.
- (5.11), (5.12) – A connection between an active and a passive agent must be a procedure call.
- (5.13) – A connection between two passive agents must be a procedure call from a non-procedure port.
- (5.14), (5.15) – A connection between a hierarchical and a passive agent must be a one-way connection.

The above definition treats hierarchical agents almost like active ones. However, connections with ports of hierarchical agents can make some substitution of pages illegal, i.e. after the transformation of a hierarchical diagram into the equivalent flat one, all connections must satisfy the conditions (5.1)-(5.6).

Let a hierarchical agent $X \in \mathcal{A}^i_H$ be given and let $\mathcal{P}^X_{join}(D^j)$ denotes the set of all *join ports* of the page D^j with respect to X, i.e.

$$\mathcal{P}^X_{join}(D^j) = \{X^j_k.p \in \mathcal{P}(D^j) \colon p \in \mathcal{N}(\mathcal{P}(X))\}. \tag{5.16}$$

In other words, $\mathcal{P}^X_{join}(D^j)$ is the set of all ports from the page D^j that names are the same as those of the hierarchical agent X.

An attempt to assign a page D^j to an hierarchical agent X results in the following set of hierarchical communication channels:

$$\begin{aligned} \mathcal{C}_X^j = \{(X_k^i.p, X_m^j.q)\colon (X_k^i.p, X.q) \in \mathcal{C}^i\} \cup \\ \cup \{(X_m^j.q, X_k^i.p)\colon (X.q, X_k^i.p) \in \mathcal{C}^i\} \end{aligned} \tag{5.17}$$

Definition 5.3. *Let a hierarchical agent $X \in \mathcal{A}_H^i$ and a page $D^j = (\mathcal{A}^j, \mathcal{C}^j, \sigma^j)$ be given. Agent X and page D^j satisfy the* simple substitution *requirements, iff*

$$card(\mathcal{P}(X)) = card(\mathcal{P}_{join}^X(D^j)), \tag{5.18}$$

$$\mathcal{P}_{join}^X(D^j) \subseteq \mathcal{P}_{unc}^j, \tag{5.19}$$

and the page $D' = (\mathcal{A}', \mathcal{C}', \sigma')$, where

- $\mathcal{A}' = \mathcal{A}^i \cup \mathcal{A}^j - \{X\}$,
- $\mathcal{C}' = \mathcal{C}^i \cup \mathcal{C}^j \cup \mathcal{C}_X^j - \{(p,q)\colon p \in \mathcal{P}(X) \vee q \in \mathcal{P}(X)\}$,
- $\sigma'(Y) = \begin{cases} \sigma^i(Y)\colon Y \in \mathcal{A}_A^i \\ \sigma^j(Y)\colon Y \in \mathcal{A}_A^j \end{cases}$,

satisfies all conditions from Definition 5.2.

If instead of the condition (5.18), the following condition is satisfied:

$$card(\mathcal{P}(X)) < card(\mathcal{P}_{join}^X(D^j)), \tag{5.20}$$

we say that agent X and page D^j satisfy the extended substitution *requirements.*

We will consider a *binding function* π that maps ports of a hierarchical agent to the join ports of the corresponding page. In the case of a simple substitution, the *binding function* π is a bijection. On the other hand, in the case of an extended substitution, one port of a hierarhical agent may have assigned more than one join port on the subpage.

Let us recall the definition of a labelled directed graph.

Definition 5.4. *A* labelled directed graph *is a triple $\mathcal{G} = (V, E, L)$, where:*

- *V is the set of* nodes.
- *L is the set of* edge labels.
- *$E \subseteq V \times L \times V$ is the set of* edges.

Definition 5.5. *A* hierarchical communication diagram *is a pair $H = (\mathcal{D}, \gamma)$, where:*

- *$\mathcal{D} = \{D^1, \ldots, D^k\}$ is the set of* pages *of the hierarchical communication diagram, such that sets of agents $\mathcal{A}^i$ $(i = 1, \ldots, k)$ are pairwise disjoint.*
- *$\gamma\colon \mathcal{A}_H \to \mathcal{D}$, where $\mathcal{A}_H = \bigcup_{i=1,\ldots,k} \mathcal{A}_H^i$, is the* substitution function, *such that:*

1. *γ is an injection.*
2. *For any $X_j^i \in \mathcal{A}_H$, X_j^i and $\gamma(X_j^i)$ satisfy the requirements of the simple or extended substitution.*
3. *Labelled directed graph $\mathcal{G} = (\mathcal{D}, E, \mathcal{A}_H)$ where $(D^i, X_k^i, D^j) \in E$ iff $\gamma(X_k^i) = D^j$ is a tree or a forest.*

The labelled directed graph defined above is called a *page hierarchy graph*. Nodes of such a graph represent pages, while edges (labelled with names of hierarchical agents) represent the substitution function γ. Each edge represents the page to which belongs the hierarchical agent (used as label) and the subpage associated with the agent.

We assume that system definition starts from a page or a set of pages, thus the number of pages must be greater than the number of hierarchical agents. Formally pages from the set $\mathcal{D} - \gamma(\mathcal{A}_H)$ are called *primary pages*, They are roots of trees that constitute a page hierarchy graph.

Following symbols are valid for hierarchical communication diagrams:

- $\mathcal{A}_A = \bigcup_{i=1,\ldots,k} \mathcal{A}_A^i$,
- $\mathcal{A}_P = \bigcup_{i=1,\ldots,k} \mathcal{A}_P^i$,
- $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$,
- $\sigma\colon \mathcal{A}_A \to \{\mathit{False}, \mathit{True}\}$ and $\forall i = 1, \ldots, k\ \forall X_j^i \in \mathcal{A}_A\colon \sigma(X_j^i) = \sigma^i(X_j^i)$,
- $\mathcal{C} = \bigcup_{i=1,\ldots,k} \mathcal{C}^i \cup \bigcup_{X \in \mathcal{A}_H \wedge \gamma(X) = D^j} \mathcal{C}_X^j$.

An example of the simple substitution is shown in Fig. 2.5 and 2.6. The page shown in Fig. 2.6 is assigned to the agent B. The following equalities hold.

- $\mathcal{P}(B) = \{B.d, B.e, B.f\}$
- $\mathcal{P}_{join}^B(D^2) = \{D.d, E.e, F.f\}$
- $\mathcal{N}(\mathcal{P}(B)) = \{d, e, f\} = \mathcal{N}(\mathcal{P}_{join}^B(D^2))$

Of course, the binding function *binds* ports with the same names.

An example of the extended substitution is shown in Fig. 2.5, 2.9 and 2.10. The page hierarchy graph for the readers-writers model is shown in Fig. 2.11. Both substitions used in the model are the extended ones. Let focus on the *Readers* agent. The following equalities hold:

- $\mathcal{P}(\mathit{Readers}) = \{\mathit{Readers.r\_in}, \mathit{Readers.r\_out}\}$
- $\mathcal{P}_{join}^{Readers}(\mathit{pReaders}) = \{\mathit{Reader1.r\_in}, \mathit{Reader1.r\_out}, \mathit{Reader2.r\_in}, \mathit{Reader2.r\_out}, \mathit{Reader3.r\_in}, \mathit{Reader3.r\_out}, \mathit{Reader4.r\_in}, \mathit{Reader4.r\_out}\}$
- $\mathcal{N}(\mathcal{P}(\mathit{Readers})) = \{\mathit{r\_in}, \mathit{r\_out}\} = \mathcal{N}(\mathcal{P}_{join}^{Readers}(\mathit{pReaders}))$

In this case, the binding function π is defined as follows:

- $\pi(\mathit{Readers.r\_in}) = \{\mathit{Reader1.r\_in}, \ldots, \mathit{Reader4.r\_in}\}$
- $\pi(\mathit{Readers.r\_out}) = \{\mathit{Reader1.r\_out}, \ldots, \mathit{Reader4.r\_out}\}$

Instead of *local* binding functions, we can consider one *global* function π:

$$\pi\colon \bigcup_{X\in\mathcal{A}_H} \mathcal{P}(X) \to 2^{\mathcal{P}}. \tag{5.21}$$

The function π satisfies the following conditions:

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \pi(X.p) \subseteq \mathcal{P}^X_{join}(\gamma(X)), \tag{5.22}$$

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \mathcal{N}(\pi(X.p)) = \{p\}. \tag{5.23}$$

If a communication diagram contains only simple substitutions, then the function (5.21) takes the simplified form:

$$\pi\colon \bigcup_{X\in\mathcal{A}_H} \mathcal{P}(X) \to \mathcal{P}, \tag{5.24}$$

and the condition (5.22):

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \pi(X.p) \in \mathcal{P}^X_{join}(\gamma(X)). \tag{5.25}$$

It can be useful to designate relations between hierarchical agent and agents belonging to its subpage.

Definition 5.6. *Let $X \in A_H$ and a page D^i such that $\gamma(X) = D^i$ be given. For any agent $Y \in A^i$ we say that X is* directly hierarchically dependent on *Y and we will denote it as $X \succ Y$.*

5.3 Hierarchy elimination

The possibility of substitution of an abstract description of an agent by a more detailed one represented by a submodel (subpage) it is very common in a system design. It is however difficult when we would like to understand (or verify) a behaviour of a whole system, associations among their components and so on. Thus, in this section we introduce the *flat* (non-hierarchical) abstraction of a system represented by its *hierarchical communication diagram*. In this representation we will use only agents and connections among them inherited from the *hierarchical communication diagram*.

Definition 5.7. *For any two agents $X \in \mathcal{A}_H$ and $Y \in \mathcal{A}$, X is said to be* hierarchically dependent on *Y, denoted as $X \succeq Y$, iff $X = Y_1 \succ \ldots \succ Y_k = Y$ for some $Y_1, \ldots, Y_n \in \mathcal{A}$.*

Definition 5.8. *A* flat representation *of a communication diagram $H = (\mathcal{D}, \gamma)$ is the triple $(\mathcal{F}, \mathcal{C}', \sigma')$ such that:*

1. $\forall X, Y \in \mathcal{F} \subseteq \mathcal{A}\colon X \neq Y \Rightarrow X \not\succeq Y$,
2. $\forall X \in \mathcal{A} - \mathcal{A}_H\ \exists Y \in \mathcal{F}\colon Y \succeq X$,

3. $\mathcal{C}' = \{(X.p, Y.q) \in \mathcal{C} \colon X, Y \in \mathcal{F}\}$,
4. $\sigma' = \sigma|_{\mathcal{A}_A \cap \mathcal{F}}$.

It is easy to check that the set of *primary pages* is a *flat representation* of a system represented by a hierarchical communication diagram.

We can move from one flat system representation to another, more detailed one, using the analysis operation.

Definition 5.9. *Let H be a hierarchical communication diagram, $(\mathcal{F}, \mathcal{C}', \sigma')$ be a flat representation of H, $X \in \mathcal{A}_H \cap \mathcal{F}$ and $\gamma(X) = D^i = (\mathcal{A}^i, \mathcal{C}^i, \sigma^i)$.* Analysis of the flat representation *$(\mathcal{F}, \mathcal{C}', \sigma')$ of the hierarchical diagram H in context of X is the flat representation $(\mathcal{F}^*, \mathcal{C}^*, \sigma^*)$ (denoted* $\mathrm{AN}(H, \mathcal{F}, X)$*), such that:*

1. $\mathcal{F}^* = \mathcal{F} - \{X\} \cup \mathcal{A}^i$,
2. $\mathcal{C}^* = \{(Z.p, Z'.q) \in \mathcal{C} \colon Z, Z' \in \mathcal{F}^*\}$,
3. $\sigma^* = \sigma|_{\mathcal{A}_A \cap \mathcal{F}^*}$.

Definition 5.10. *Let H be a hierarchical communication diagram, $(\mathcal{F}, \mathcal{C}', \sigma')$ be a flat representation of H, $Y \in \mathcal{F}$ and $\exists X \in A_H$ such that $X \succ Y$ and $\gamma(X) = D^i = (\mathcal{A}^i, \mathcal{C}^i, \sigma^i)$.* Synthesis of the flat representation *$(\mathcal{F}, \mathcal{C}', \sigma')$ of the hierarchical diagram H in context of Y is the flat representation $(\mathcal{F}^*, \mathcal{C}^*, \sigma^*)$ (denoted as* $\mathrm{SN}(H, \mathcal{F}, Y)$*) such that:*

1. $\mathcal{F}^* = \mathcal{F} - \mathcal{A}^i \cup \{X\}$,
2. $\mathcal{C}^* = \{(Z.p, Z'.q) \in \mathcal{C} \colon Z, Z' \in \mathcal{F}^*\}$,
3. $\sigma^* = \sigma|_{\mathcal{A}_A \cap \mathcal{F}^*}$.

Page D' (presented in Fig. 2.7) is a flat representation of the hierarchical system H defined by pages D^1 and D^2 (presented in Fig. 2.5 and 2.6) with the simple substitution mechanism. Flat representation generated by the $\mathrm{AN}(H, D^1, B)$ analysis operation (Fig. 2.7) is generated by the following algorithm.

1. Remove the agent B from the page D^1 with all its connections.
2. Move the contents of the page D^2 onto the page D^1.
3. Add connections – If after removing of the agent B, from the page D^1, it has been removed a connection between ports $B.a$ and $X_i^1.p$, then we add a connection between ports $X_i^1.p$ and $\pi(B.a)$ with the same direction as the removed one.

Page *System* (presented in Fig. 2.8) as a primary page is a flat representation of the hierarchical graph presented in Fig. 2.11 with pages *Readers* and *Writers* (presented appropriately in Fig. 2.9 and Fig. 2.10) with the extended substitution mechanism. Flat representation generated by the composition of the analysis operations $\mathrm{AN}(H, \mathrm{AN}(H, \mathit{System}, \mathit{Readers}), \mathit{Writers})$ is presented in Fig. 2.12. This operation is supported by nearly the same algorithm as above with one change (in the third step). If after removing of a hierarchical agent

X_j^i, it has been removed a connection between ports $X_j^i.p$ and $X_n^i.q$, then we add similar connections between port $X_n^i.q$ and all ports from the set $\pi(X_j^i.p)$.

Definition 5.11. *A flat representation* $(\mathcal{F}, \mathcal{C}', \sigma')$ *is called the* maximal flat representation *iff*

$$\forall X \in \mathcal{A}\ \exists Y \in \mathcal{F}\colon X \succeq Y. \tag{5.26}$$

Such a maximal flat representation does not contain hierarchical agents.

5.4 Models

Formally, we define an Alvis model as a triple with a hierarchical communication diagram as shown in Definition 5.12.

Definition 5.12. *An Alvis* model *is a triple* $\mathbf{A} = (H, B, \varphi)$*, where:*

- $H = (\mathcal{D}, \gamma)$ *is a* hierarchical communication diagram*,*
- B *is a syntactically correct* code layer*,*
- φ *is a* system layer.

Moreover, each non-hierarchical agent X belonging to the diagram H must be defined in the code layer, and each agent defined in the code layer must belong to the diagram.

For an Alvis model $\mathbf{A} = (H, B, \varphi)$*, its* equivalent non-hierarchical model *is a triple* $\overline{\mathbf{A}} = (D, B, \varphi)$*, where D is the maximal flat representation of H.*

It should be underlined that a syntactically correct code layer means also that only input ports may be used as arguments of *in* statements, and only output ports may be used as arguments of *in* statements.

Let us focus on system layers. The *system layer* is the predefined one and depends on the model running environment, i.e. the hardware and/or operating system. The layer is necessary for a model simulation and an LTS graph generation. Moreover, the layer provides some functions that are useful for the implementation of scheduling algorithms or for retrieving information about other agents states. Two system layers are considered in this book called α^0 and α^1 ones.

The α^0 system layer makes Alvis a formal modelling language for concurrent systems. The layer is based on the following assumptions:

- Each active agent has access to its own processor and performs its statements as soon as possible.
- The predefined α^0 scheduler function is called after each statement automatically and makes agents running as soon as possible.

- In case of conflicts, agents priorities are taken under consideration. If two or more agents with the same highest priority compete for the same resources, the system works indeterministically.
  A *conflict* is a state when two or more agents try to call a procedure of the same passive agent or two or more active agents try to communicate with the same active agent.

The semantic of Alvis models with α^0 system layer is considered in Chapter 6.

The α^1 layer is based on the following assumptions:

- All active agents share the same processor.
- The predefined α^1 scheduler function is called after each statement automatically and makes agents running as soon as possible.

We can consider different α^1 system layers that differ about the scheduling algorithm. α^1 system layers are the most suitable ones for embedded systems. The semantic of Alvis models with α^1 system layers is considered in Chapter 6.

6

Models with α^0 system layer

The α^0 system layer is the most universal one and makes Alvis similar to other formal languages like Petri nets, process algebras, time automata, etc. This chapter focuses on untimed version of Alvis models with α^0 system layer. This flavour of the Alvis language is suitable for modelling concurrent systems. Moreover, it is a good starting point for more complex considerations.

6.1 Code layer for untimed models

The Alvis language contains three statements that use time explicitly: *delay*, *loop every* and *select* with *delay* branches. These statements are forbidden in untimed models. Untimed models are also not allowed to use the *environment* statement thus *cli* and *sti* statements are not taken into considerations. On the other hand, if every active agent has access to its own processor then it is not necessary to consider critical sections or the *jump far* statement (to transfers the control to another agent). The set of allowed statements for Alvis untimed models with α^0 system layer is given in Table 6.1.

6.2 Agents state

As it was shown in Chapter 5, one can consider the maximal flat representation of a communication diagram instead of a hierarchical model. Thus, from now on, we will consider only $\overline{\mathbf{A}} = (D, B, \alpha^0)$ models. To define a state of an Alvis model, we need to define a state on a single agent.

Definition 6.1. *A* state of an agent X *is a tuple*

$$S(X) = (am(X), pc(X), ci(X), pv(X)), \tag{6.1}$$

where $am(X)$, $pc(X)$, $ci(X)$ *and* $pv(X)$ *denote mode, program counter, context information list and parameters values of the agent* X *respectively.*

Table 6.1. Alvis statements allowed in untimed models with α^0 system layer

Statement	Description
`exec x = expression`	Evaluates the expression and assigns the result to the parameter; the *exec* keyword can be omitted.
`exit`	Terminates an active agent or a passive agent procedure.

`if (g1) {...}`
`elseif (g2) {...}`
`elseif (g3) {...}`
`...`
`else {...}`
	Conditional statement.

`in p` `in p x`	Collects a signal/value through port p.
`jump label`	Transfers the control to the line of code identified with the *label*.
`loop {...}` `loop (g) {...}`	Infinite loop. Repeats execution of the contents while the guard if satisfied..
`null`	Empty statement.
`out p` `out p x`	Sends a signal/value through the port p.
`proc (g) p {...}`	Defines the procedure for port p of a passive agent. The guard is optional.

`select {`
`  alt (g1) {...}`
`  alt (g2) {...}`
`  alt (g3) {...}`
`  ...`
`}`
	Selects one of alternative choices.

`start A`	Starts the agent A if it is in the *Init* state, otherwise do nothing.

All possible modes and transitions among them are shown in Fig. 6.1.

finished – The mode means that an agent has finished its work or it has been terminated using the *exit* statement.

init – This is the default mode for agents that are inactive in the initial state. An agent can be activated by another one with the *start* statement.

running – The mode means that an agent is performing one of its statements.

taken – The mode means that one of the passive agent procedures has been called and the agent is executing it.

waiting – For passive agents, the mode means that the corresponding agent is inactive and waits for another agent to call one of its accessible procedures. For active agents, the mode means that the corresponding agent is waiting either for a communication with another active agent, or for a currently inaccessible procedure of a passive agent.

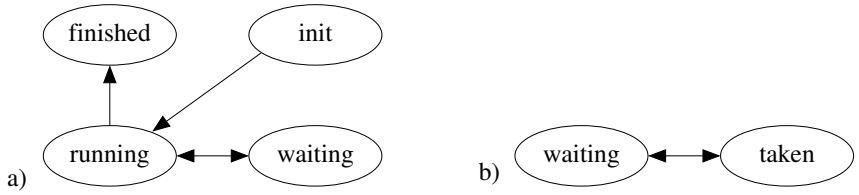


Fig. 6.1. Possible transitions among modes: a) active agents, b) passive agents.

The program counter points out the current statement of an agent i.e. the next statement to be executed or the statement that has been executed by an agent but needs a feedback from another agent to be completed (e.g. a communication between two active agents). Relationships between the mode and the program counter of an agent are shown in Table 6.2.

- We say that $pc(X)$ *points out* an *exec* (*exit, jump, null, start*) statement iff the next statement to be executed is an *exec* (*exit, jump, null, start*) statement.
- We say that $pc(X)$ *points out* an *if* statement iff the next statement to be executed is the evaluating of the guard and possibly entering one of the *if* statement alternatives.
- We say that $pc(X)$ *points out* a *loop* statement iff the next statement to be executed is the evaluating of the guard (if any) and possibly entering the *loop* statement.
- We say that $pc(X)$ *points out* a *select* statement iff the next statement to be executed is entering the *select* statement and possibly one of its branches.
- We say that $pc(X)$ *points out* an *in* or *out* statement iff the next statement to be executed is an *in* or *out* statement or the last executed statement is *in* or *out* and the agent is waiting for the communication to be completed (either with an active or a passive agent).

Table 6.2. Relationships between the mode and the program counter of an agent

$am(X)$	$pc(X)$
finished	0
init	0
running	current statement
taken	current statement of the called procedure
waiting (active agent)	current statement
waiting (passive agent)	0

The context information list contains additional information about the current state of an agent e.g. if an agent is the *waiting* mode, *ci* contains

information about events the agent is waiting for. Possible entries put into *ci* lists are given in Table 6.3. If an agent is the *init* or *finished* mode, its context information list is empty.

Table 6.3. Relationships between the mode and the context information list of an agent

agent X	$am(X)$	$ci(X)$ entry	description
active	running/ waiting	$proc(Y.b, a)$	X has called the $Y.b$ procedure via port a and this procedure is being executed in the X agent context
active	waiting	$in(a)$, $in(a\|T)$	X waits for a communication via port $X.a$ ($X.a$ is the input port for this communication); T is the type of the expected value
		$out(a)$, $out(a\|T)$	X waits for a communication via port $X.a$ ($X.a$ is the output port for this communication)
		$guard$	X waits for an open branch of a *select* statement
passive	taken	$proc(Y.b, a)$	X has called the $Y.b$ procedure via port a and this procedure is being executed in the same context as the X procedure
		$guard$	X waits for an open branch of a *select* statement
passive	waiting	$in(a)$	input procedure $X.a$ is accessible
		$out(a)$	output procedure $X.a$ is accessible

The *parameters values list* contains the current values of the agent parameters.

6.3 Model state

A model state is sequence (list) of all agents states.

Definition 6.2. *A* state *of a model* $\overline{\mathbf{A}} = (D, B, \varphi)$*, where* $D = (\mathcal{A}, \mathcal{C}, \sigma)$ *and* $\mathcal{A} = \{X_1, \ldots, X_n\}$ *is a tuple*

$$S = (S(X_1), \ldots, S(X_n)). \tag{6.2}$$

Definition 6.3. *The* initial state *of a model* $\overline{\mathbf{A}} = (D, B, \alpha^0)$ *is a tuple* S_0 *as given in (6.2), where:*

- $am(X) = running$ *for any active agent* X *such that* $\sigma(X) = True$*;*
- $am(X) = init$ *for any active agent* X *such that* $\sigma(X) = False$*;*
- $am(X) = waiting$ *for any passive agent* X*;*
- $pc(X) = 1$ *for any active agent* X *in the running mode and* $pc(X) = 0$ *for other agents.*

- $ci(X) = [\,]$ *for any active agent* X*;*
- *For any passive agent* X*,* $ci(X)$ *contains names of all accessible ports of* X *(i.e. names of all accessible procedures) together with the direction of parameters transfer, e.g.* $in(a)$*,* $out(b)$*, etc.*
- *For any agent* X*,* $pv(X)$ *contains* X *parameters with their initial values.*

Steps performed by a model are described using the *transition* idea. The set of all possible transitions for the considered Alvis models is given in Table 6.4.

Table 6.4. Set of transitions

	Symbol	Description
1	t_{exec}	performs an evaluation and assignment
2	t_{exit}	terminates an agent or a procedure
3	t_{if}	enters an if statement
4	t_{in}	performs communication (input side)
5	t_{jump}	jumps to a label
6	t_{loop}	enters a *while* or *infinite* loop
7	t_{null}	performs an empty statement
8	t_{out}	performs communication (output side)
9	t_{select}	enters a select statement
10	t_{start}	starts an inactive agent
11	t_{io}	performs communication (both sides)

To define formally results of transitions execution, we have to provide some mechanisms for code analysis. Let us define the following symbols.

- $B(X)$ – the X agent code definition (the agent block);
- $card(B(X))$ – the number of *steps* in $B(X)$;
- $B_i(X)$ for $i = 1, \ldots, card(B(X))$ – the name of the agent X i-th step, $B_i(X) \in \{exec, exit, if, in, jump, loop, null, out, select, start\}$.
- $\mathcal{N}(t)$ – the name of the transition t (possible values the same as for steps).
- If necessary am, pc, ci, pv will be indicated by indexes S, S' etc. to point out the state they refer to.

The set of all transitions available for a particular model will be denoted by $\mathcal{T}$. For example, the t_{start} is available for a model $\overline{\mathbf{A}} = (D, B, \alpha^0)$ iif $\exists X \in \mathcal{A}, \exists i \in \{1, \ldots, card(B(X))\}\colon B_i(X) = start$.

Let us focus on the *step* idea. It is necessary to distinguish between code statements and steps. More statements e.g. *exec*, *exit*, *in*, etc. are *single-step* statements. On the other hand, *if*, *loop* and *select* are *multi-step* statements. We use recursion to count the number of steps for multi-step statements. For each of these statements, the first step enters the statement interior. Then, we count steps of statements put inside curly brackets. For a given statement s, let $stepno(s)$ denote the number of the step related to s. For multi-step

```
agent A {
  i :: Int = 0;
  loop {                                -- 1
    select {                            -- 2
      alt (i == 0) { in p; i = 1;}      -- 3, 4
      alt (i == 1) { in q; i = 0;}      -- 5, 6
    }
    if(i == 1) { out p;}                -- 7, 8
    else { null; }                      -- 9
  }
}
```

Listing 6.1. Steps counting in Alvis code

statements, $stepno(s)$ denotes the number of the step connected with entering the statement interior.

Let us consider the piece of code shown in Listing 6.1. It contains 9 steps. The steps number are put inside comments. For example, the step 7 denotes entering the *if* statement, while the step 8 denotes the *out* statement. For passive agents, only statements inside procedures (i.e. inside curly brackets) are taken into consideration while counting steps.

To simplify the formal description of transitions, we need a function that determines the next program counter for an agent. For the purposes of this discussion *block* means a piece of a code inside curly brackets and *last block statement* means that the statement is the last one in the block and is followed by the closing curly bracket. Depending on the surrounding statement we will consider: *if blocks* (any of the blocks after *if*, *elseif* or *else* clauses), *loop blocks*, *branch blocks* (*alt* clauses), *procedure blocks* and *agent blocks* (a main agent's block).

Let us focus on code statements first. The *nextst* function (*next statement*) is used to determine the successor statement for a given one. The function returns *empty statement* if there is no a successor statement for the considered one. The number of the *empty statement* is equal to 0. This recursive function is based on the following rules:

1. If s is a *jump* statement then $nextst(s)$ is the first statement after the *jump* statement label.
2. If s is an *exit* statement then $nextst(s)$ is the empty statement.
3. If $s \in \{exec, if, in, loop, null, out, select, start\}$ and s is not the last block statement then $nextst(s)$ is the statement that follows s in the code layer.
4. If $s \in \{exec, if, in, loop, null, out, select, start\}$ and s is the last main block statement or the last procedure block then $nextst(s)$ is the empty statement.
5. If $s \in \{exec, if, in, loop, null, out, select, start\}$ and s is the last *if* (*loop*, *select*) block statement then $nextst(s) = nextst(s')$, where s' is the surrounding *if* (*loop*, *select*) statement.

A graph representation of the *nextst* function for the code presented in Listing 6.1 is shown in Fig. 6.2. For example, the next statement for the *exec* statement number 6 is the *if* statement (statement number 7).

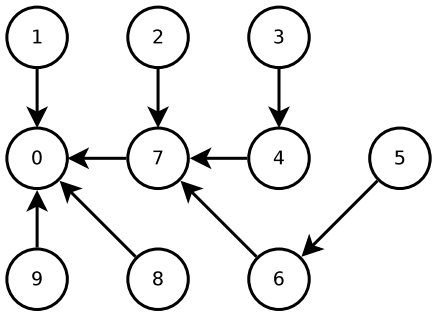


Fig. 6.2. Graph representation of the *nextst* function for the code presented in Listing 6.1.

A similar *nextpc* (next program counter) function determines the number of the next step (the next program counter for an agent). Similarly, we use concepts like *last block step*, *last if block step*, etc. to point out the last step inside a given code block. It is possible that there is no the last main block step e.g. if an agent behaviour is defined with an infinite loop (see Listing 6.1).

The *nextpc* function takes an agent X state as an argument and returns an integer in the range of 0 to $card(B(X))$. The function satisfies the following requirements for the current step t:

1. If $t = exit$ then $nextpc(S(X)) = 0$.
2. If $t \in \{exec, in, null, out, start\}$ then:
   - if t is not the last block step then:
     $nextpc(S(X)) = pc_S(X) + 1$;
   - if t is the last main block step or the last procedure block step then
     $nextpc(S(X)) = 0$;
   - if t is the last *loop* block step then:
     $nextpc(S(X)) = stepno(s)$,
     where s is the *loop* statement;
   - if t is the last branch block step then:
     $nextpc(S(X)) = stepno(nextst(s))$,
     where s is the surrounding *select* statement.
   - if t is the last *if* block step then:
     $nextpc(S(X)) = stepno(nextst(s))$,
     where s is the surrounding *if* statement.
3. If $t = jump$ step then $nextpc(S(X))$ returns the number of the first step after the corresponding label.
4. If $t = if$ then:

- $nextpc(S(X))$ is equal to the number of the first step inside the chosen *if* block, if such a block has been chosen;
- $nextpc(S(X)) = stepno(nextst(s))$, where s is the *if* statement otherwise.

5. If $t = loop$ then:
   - if the loop guard is satisfied or for an infinite loop $nextpc(S(X)) = pc_S(X) + 1$;
   - if the loop guard is not satisfied then:
     $nextpc(S(X)) = stepno(nextst(s))$,
     where s is the *loop* statement.
6. If $t = select$ then $nextpc(S(X))$ returns the number of the first step inside the chosen branch block.

A graph representation of the *nextpc* function for the code presented in Listing 6.1 is shown in Fig. 6.3.

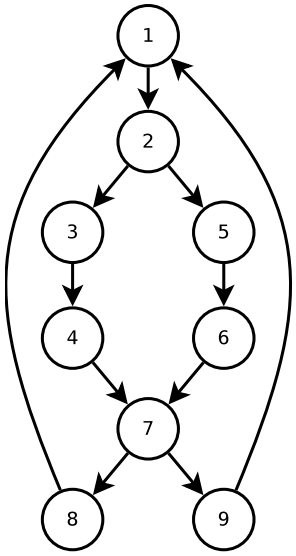


Fig. 6.3. Graph representation of the *nextpc* function for the code presented in Listing 6.1.

To describe a *ci* list modifications we will use the following operators:

- $e \in ci$ – returns *true* if the element e belongs to the *ci* list and *false* otherwise.
- $ci \oplus e$ – if $e \notin ci$ then adds the element e at the end of the list.
- $ci \ominus e$ – if $e \in ci$ then removes the element e from the list.

6.4 Transitions

We will consider behaviour of Alvis models at the level of detail of single steps. Each of transitions presented in Table 6.4 realises a single step. Each step is realised in the context of one active agent. Also procedures of passive agents are realised in context of active agents that called them. Firstly, we will focus on active agents only.

Definition 6.4. *Assume* $\overline{\mathbf{A}} = (D, B, \alpha^0)$ *is an Alvis model with the current state* $S = (S(X_1), \ldots, S(X_n))$ *and* $X_i \in \mathcal{A}_A$. *A transition* $t \in \mathcal{T}$ *is* enable *in the state* S with respect to X_i *(denoted as* $S{-}t(X){\rightarrow}$*) iff the following requirement holds:*

$$am(X_i) = running \ \wedge \ B_{pc(X_i)}(X_i) = \mathcal{N}(t). \tag{6.3}$$

The fact that a transition t is enable in a state S with respect to an agent X and the state S' that is the result of executing t in S will be denoted by $S{-}t(X){\rightarrow}S'$. If case of four transitions, an extended version of this notation will be used:

- $S{-}t_{start}(X, Y){\rightarrow}S'$, where Y is the argument of the corresponding *start* statement;
- $S{-}t_{in}(X.p, T){\rightarrow}S'$, $S{-}t_{out}(X.p, T){\rightarrow}S'$, where $X.p$ is the port used for the communication and T is the type of send/collected value. If necessary, the special *Empty* type will be used to denote a valueless communication.
- $S{-}t_{io}(X.p, Y.q, T){\rightarrow}S'$, where $X.p$ and $Y.q$ are the input and output ports for the communication respectively and T is the type of the transferred value.

This section describes the states that are results of executing all possible steps. We will limit the definitions to description of agents, which states change. Agent, which states remain unchanged are omitted in the description.

Let $pv_S(X)|_{v=a}$ denote the list of parameters values $pv_S(X)$, but with the parameter v assigned to a new value a. If $X \in \mathcal{A}_A$, $S{-}t_{exec}(X){\rightarrow}S'$, and a parameter v is assign a value a with the corresponding *exec* statement, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X)|_{v=a})$, if $nextpc(S(X)) \neq 0$,
- $S'(X) = (finished, 0, [\,], pv_{S'}(X))$, if $nextpc(S(X)) = 0$.

If $X \in \mathcal{A}_A$ and $S{-}t_{exit}(X){\rightarrow}S'$, then:

- $S'(X) = (finished, 0, [\,], pv_S(X))$,

If $t \in \{if, loop, null\}$, $X \in \mathcal{A}_A$ and $S{-}t(X){\rightarrow}S'$ then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$, if $nextpc(S(X)) \neq 0$.
- $S'(X) = (finished, 0, [\,], pv_S(X))$, if $nextpc(S(X)) = 0$.

If $X \in \mathcal{A}_A$ and $S{-}t_{jump}(X){\rightarrow}S'$, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$.

If $X \in \mathcal{A}_A$ and $S{-}t_{select}(X){\rightarrow}S'$, then:

- If at least one branch of the statement is open then $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$.
- If all branches are closed then $S'(X_i) = (waiting, pc_S(X_i), ci_S(X) \oplus guard, pv_S(X))$.

If $X, Y \in \mathcal{A}_A$ and $S{-}t_{start}(X,Y){\rightarrow}S'$, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$, if $nextpc(S(X)) \neq 0$.
- $S'(X) = (finished, 0, [\,], pv_S(X))$, if $nextpc(S(X)) = 0$.
- If $am_S(Y) = init$, then $S'(Y) = (running, 1, [\,], pv_S(Y))$, otherwise $S'(Y) = S(Y)$.

Steps of passive agents are always considered in the context of an active one. Thus, to define enable transitions for passive agents, it is necessary to consider behaviour of at least a pair of agents.

Definition 6.5. *Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model with the current state $S = (S(X_1), \ldots, S(X_n))$, $X \in \mathcal{A}$ and $Y \in \mathcal{A}_P$.*

- *We say that X is* (directly) performing input procedure *$Y.q$ via its port p iff $(X.p, Y.q) \in \mathcal{C}$, $proc(Y.q, p) \in ci_S(X)$ and $am_S(Y) = taken$.*
- *We say that X is* (directly) performing output procedure *$Y.q$ via its port p iff $(Y.q, X.p) \in \mathcal{C}$, $proc(Y.q, p) \in ci_S(X)$ and $am_S(Y) = taken$.*
- *We say that X is* performing a procedure *of an agent Y iff X is performing an input or output procedure of Y via one of its ports.*
- *We say that X is* indirectly performing input (output) procedure *$Y.q$ iff exist $X_k^1, \ldots, X_k^m$, $m > 0$ such that X is performing a procedure of X_k^1, X_k^1 is performing a procedure of X_k^2, ..., X_k^m is performing input (output) procedure $Y.q$ via one of its ports. For any passive agent Y performing one of its procedures, $context(Y)$ will denote the active agent X that directly or indirectly performs the procedure.*

Definition 6.6. *Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model with the current state $S = (S(X_1), \ldots, S(X_n))$ and $X \in \mathcal{A}_A$, $Y \in \mathcal{A}_P$ are agents such that X is directly or indirectly performing input (or output) procedure $Y.q$ via its port p. A transition $t \in \mathcal{T}$ is* enable *in the state S* with respect to *Y iff the following requirement holds:*

$$am(X) = running \ \wedge \ B_{pc(Y)}(Y) = \mathcal{N}(t). \tag{6.4}$$

Performing the *in* or *out* statements may influence more than one agent state. Very often two agents connected with a communication channel perform

their communication statements simultaneously. Such a communication is possible only if types of sending and collecting values are the same. Moreover, if a few agents try to communicate at the same time, their priorities are taken into consideration to determine pairs of agents that perform their communication statements simultaneously using the same communication channels. Similarly, if an agents calls an available procedure of a passive agent, states of two agents change (in spite of the fact that only one of them performs a step).

Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model with the current state $S = (S(X_1), \ldots, S(X_n))$. Let $type(X.p)$ denote the type of a procedure p argument for a passive agent X (The *Empty* type can be used if none argument is used.) In case of active agents, $type_S(X.p)$ will denote the type of sent (expected) argument for already performed *out* (*in*) step. Let us define the following set of pairs:

$$\begin{aligned}
Comm_S^{AA} = \{&(X,Y)\colon X,Y \in \mathcal{A}_A \wedge S{-}t_{in}(X.p,T){\rightarrow} \wedge \\
&\wedge\ S{-}t_{out}(Y.q,T){\rightarrow} \wedge (Y.q,X.p) \in \mathcal{C}\} \qquad (6.5)\\
Comm_S^{AP} = \{&(X,Y)\colon X \in \mathcal{A}_A \wedge Y \in \mathcal{A}_P \wedge S{-}t_{in}(X.p,T){\rightarrow} \wedge \\
&\wedge\ am(Y) = waiting \wedge out(q) \in ci(Y) \wedge \\
&\wedge\ type(Y.q) = T \wedge (Y.q,X.p) \in \mathcal{C}\} \cup \\
&\{(X,Y)\colon X \in \mathcal{A}_A \wedge Y \in \mathcal{A}_P \wedge S{-}t_{out}(X.p,T){\rightarrow} \wedge \\
&\wedge\ am(Y) = waiting \wedge in(q) \in ci(Y) \wedge \\
&\wedge\ type(Y.q) = T \wedge (X.p,Y.q) \in \mathcal{C}\} \qquad (6.6)\\
Comm_S^{PP} = \{&(X,Y)\colon X,Y \in \mathcal{A}_P \wedge S{-}t_{in}(X.p,T){\rightarrow} \wedge \\
&\wedge\ am(Y) = waiting \wedge out(q) \in ci(Y) \wedge \\
&\wedge\ type(Y.q) = T \wedge (Y.q,X.p) \in \mathcal{C}\} \cup \\
&\{(X,Y)\colon X,Y \in \mathcal{A}_P \wedge S{-}t_{out}(X.p,T){\rightarrow} \wedge \\
&\wedge\ am(Y) = waiting \wedge in(q) \in ci(Y) \wedge \\
&\wedge\ type(Y.q) = T \wedge (X.p,Y.q) \in \mathcal{C}\} \qquad (6.7)\\
Comm_S^{F} = \{&(X,Y)\colon X,Y \in \mathcal{A}_A \wedge S{-}t_{in}(X.p,T){\rightarrow} \wedge \\
&\wedge\ am(Y) = waiting \wedge out(q) \in ci(Y) \wedge \\
&\wedge\ type_S(Y.q) = T \wedge (Y.q,X.p) \in \mathcal{C}\} \cup \\
&\{(X,Y)\colon X,Y \in \mathcal{A}_A \wedge S{-}t_{out}(X.p,T){\rightarrow} \wedge \\
&\wedge\ am(Y) = waiting \wedge in(q) \in ci(Y) \wedge \\
&\wedge\ type_S(Y.q) = T \wedge (X.p,Y.q) \in \mathcal{C}\} \qquad (6.8)\\
Comm_S^{*} = {}& Comm_S^{AA} \cup Comm_S^{AP} \cup Comm_S^{PP} \cup Comm_S^{F} \qquad (6.9)
\end{aligned}$$

Next, we divide all agents enable for communication in the state S into two disjoint sets:

$$Comm_S^2 = \{X \in \mathcal{A}\colon \exists Y \in \mathcal{A} \wedge \\ \wedge \ \big((X,Y) \in Comm_S^{AA} \cup Comm_S^{AP} \cup Comm_S^{PP} \cup Comm_S^{F} \vee \\ \vee (Y,X) \in Comm_S^{AA}\big)\} \tag{6.10}$$

$$Comm_S^1 = \{X \in \mathcal{A}\colon \ (S{-}t_{in}(X.p,T){\rightarrow} \vee S{-}t_{out}(X.p,T){\rightarrow}) \wedge X \notin Comm_S^2\} \tag{6.11}$$

It's hardly possible that all agents from the set $Comm_S^1 \cup Comm_S^2$ can perform they communication steps simultaneously. Usually, agents compete for the same agents and we can observe some conflicts in a model.

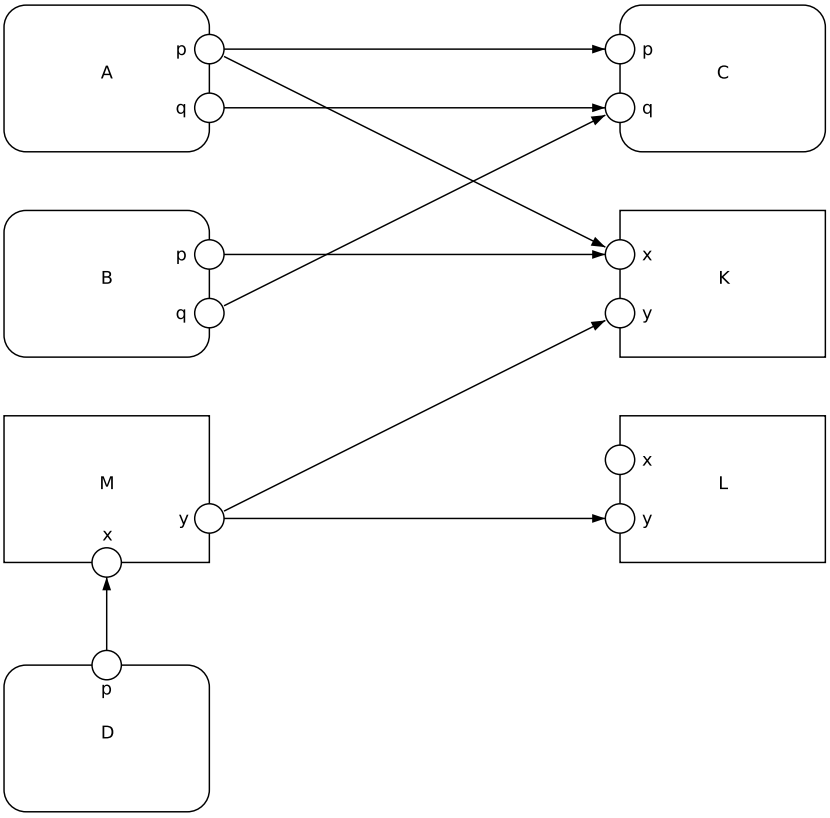


Fig. 6.4. Communication conflicts

Let us consider the communication diagram shown in Fig. 6.4. Suppose, that for a considered state S the following conditions hold:

- $S{-}t_{out}(A.p,T){\rightarrow}$,
- $S{-}t_{out}(B.p,T){\rightarrow}$,

- $S{-}t_{in}(C.p,T){\rightarrow}$,
- $am_S(D) = running$,
- $context(M) = D$, $S{-}t_{out}(M.y,T){\rightarrow}$,
- $am_S(K) = waiting$, $ci_S(K) = [in(x), in(y)]$, $type(K.x) = type(K.y) = T$,
- $am_S(L) = waiting$, $ci_S(L) = [in(x), in(y)]$, $type(L.x) = type(L.y) = T$.

Thus, we have:

- $Comm_S^{AA} = \{(C, A)\}$,
- $Comm_S^{AP} = \{(A, K), (B, K)\}$,
- $Comm_S^{PP} = \{(M, K), (M, L)\}$,
- $Comm_S^{F} = \emptyset$,
- $Comm_S^{2} = \{A, B, C, M\}$
- $Comm_S^{1} = \emptyset$.

It easy to see that there are conflicts in the state S: agents A and B compete for the procedure $K.x$, and agents B and M compete for an access to agent K.

Alvis uses a reverse priorities range. The code layer priorities range from 0 to 9, where 0 is the higher system priority. From the theoretical point of it is more convenient to use the pr function defined as follows

$$pr(X) = 9 - codePriority(X) \tag{6.12}$$

Using different agents priorities can eliminate most of potentials conflicts in a model. For example, if $pr(A) > pr(B)$ then there is no conflict between agents A and B, but it does not mean that agent A will perform the procedure $K.x$. Selecting communication steps that can be perform in a given state is based on Algorithm 1.

The output of the algorithm are two sets. The set $Comm_S^{2\prime} \subset Comm_S^2$ contains pair of agents representing communication steps that are to be perform in state S concurrently and concern pairs of agents. The set $Comm_S^{1\prime}$ contains agents that may perform communication steps on their own.

Let us go back to the model considered previously (see Fig. 6.4). Suppose, the code priority for all active agents is equal to 0 and for all passive agents to 1. Thus,

$$pr(A) = pr(B) = pr(C) = pr(D) = 9 \tag{6.13}$$

$$pr(K) = pr(L) = pr(M) = 8 \tag{6.14}$$

After the first performing of the *while* loop interior (see Algorithm 1) we have:

- $Comm'_S = \{A, B, C\}$,
- $Comm''_S = \{(C, A), (A, K), (B, K)\}$,
- $Comm_S = \{(C, A)\}$,

Algorithm 1 Selecting concurrent communication steps

$Comm_S = \emptyset$
calculate $Comm_S^{AA}$, $Comm_S^{AP}$, $Comm_S^{PP}$, $Comm_S^F$, $Comm_S^*$, $Comm_S^2$, $Comm_S^1$
▷ (see (6.5)-(6.11))
while $Comm_S^2 \neq \emptyset$ **do**
$Comm'_S = \{X \in Comm_S^2 \colon X \text{ has the highest priority in } Comm_S^2\}$
$Comm''_S = \{(X,Y) \in Comm_S^* \colon (X \in Comm'_S \vee Y \in Comm'_S)\}$
take a pair $(X',Y') \in Comm''_S$ with the highest sum of agents' priorities
▷ if there is a few such pairs take one of them
$Comm_S = Comm_S \cup \{(X',Y')\}$
$Comm_S^{AA} = Comm_S^{AA} - \{(P,Q) \colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
$Comm_S^{AP} = Comm_S^{AP} - \{(P,Q) \colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
$Comm_S^{PP} = Comm_S^{PP} - \{(P,Q) \colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
$Comm_S^F = Comm_S^F - \{(P,Q) \colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
calculate sets $Comm_S^*$, $Comm_S^2$ ▷ (see (6.9)-(6.10))
end while
calculate $Comm_S^1$ ▷ (see 6.11)
$Comm_S^{1\prime} = Comm_S^1 - Comm_S$
$Comm_S^{2\prime} = Comm_S$
▷ the new set $Comm_S^{1\prime}$ may contain more elements than initially $Comm_S^1$

- $Comm_S^{AA} = \emptyset$,
- $Comm_S^{AP} = \{(B,K)\}$,
- $Comm_S^{PP} = \{(M,K),(M,L)\}$,
- $Comm_S^F = \emptyset$,
- $Comm_S^* = \{(B,K),(M,K),(M,L)\}$,
- $Comm_S^2 = \{B,M\}$.

Then, after the second performing of the *while* loop interior we have:

- $Comm'_S = \{B\}$,
- $Comm''_S = \{(B,K)\}$,
- $Comm_S = \{(C,A),(B,K)\}$,
- $Comm_S^{AA} = Comm_S^{AP} = \emptyset$,
- $Comm_S^{PP} = \{(M,L)\}$,
- $Comm_S^F = \emptyset$,
- $Comm_S^* = \{(M,L)\}$,
- $Comm_S^2 = \{M\}$.

Finally, we have $Comm_S^{2\prime} = \{(C,A),(B,K),(M,L)\}$ and $Comm_S^{1\prime} = \emptyset$.

Suppose the priority function pr is defined as follows:

$$pr(A) = pr(B) = pr(C) = pr(D) = 8 \tag{6.15}$$

$$pr(K) = pr(L) = pr(M) = 9 \tag{6.16}$$

Then, while the first performing of the *while* loop interior we have:

- $Comm'_S = \{M\}$,
- $Comm''_S = \{(M, K), (M, L)\}$,

In such a case, there is an indeterministic choice between these two pairs. If (M, L) is chosen, then we have next another indeterministic choice between (A, K) and (B, K). Then, if (A, K) is chosen, we have finally, $Comm_S^{2'} = \{(M, L), (A, K)\}$ and $Comm_S^{1'} = \{B, C\}$.

Now, let us focus on performing communication steps. There are following possible cases:

1. $(X, Y) \in Comm_S^{2'} \cap Comm_S^{AA}$,
2. $(X, Y) \in Comm_S^{2'} \cap Comm_S^{AP}$,
3. $(X, Y) \in Comm_S^{2'} \cap Comm_S^{PP}$,
4. $(X, Y) \in Comm_S^{2'} \cap Comm_S^{F}$,
5. $X \in Comm_S^{1'} \cap \mathcal{A}_A$ and $S{-}t_{in}(X.p, T){\rightarrow}S'$,
6. $X \in Comm_S^{1'} \cap \mathcal{A}_A$ and $S{-}t_{out}(X.p, T){\rightarrow}S'$,
7. $X \in Comm_S^{1'} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p, T){\rightarrow}S'$, and $p \notin \mathcal{P}_{proc}(X)$,
8. $X \in Comm_S^{1'} \cap \mathcal{A}_P$, $S{-}t_{out}(X.p, T){\rightarrow}S'$, and $p \notin \mathcal{P}_{proc}(X)$,
9. $X \in Comm_S^{1'} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p, T){\rightarrow}S'$, and $p \in \mathcal{P}_{proc}(X)$,
10. $X \in Comm_S^{1'} \cap \mathcal{A}_P$, $S{-}t_{out}(X.p, T){\rightarrow}S'$, and $p \in \mathcal{P}_{proc}(X)$.

ad. 1

Suppose, $(X, Y) \in Comm_S^{2'} \cap Comm_S^{AA}$, $S{-}t_{in}(X.p, T){\rightarrow}$, x is the second argument of the corresponding *in* statement, $S{-}t_{out}(Y.q, T){\rightarrow}$, and value w of type T is sent. In such a case, instead of transitions t_{in} and t_{out}, the transition t_{io} is used to represent the communication. Let $S{-}t_{io}(X.p, Y.q, T){\rightarrow}S'$, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X)|_{x=w})$ if $nextpc(S(X)) \neq 0$.
- $S'(X) = (finished, 0, [\,], pv_S(X)|_{x=w})$ if $nextpc(S(X)) = 0$.
- $S'(Y) = (running, nextpc(S(Y)), ci_S(Y), pv_S(Y))$ if $nextpc(S(Y)) \neq 0$.
- $S'(Y) = (finished, 0, [\,], pv_S(Y))$ if $nextpc(S(Y)) = 0$.

If a valueless communication is considered then $pv_S(X)$ remains unchanged.

ad. 2

Suppose, $(X, Y) \in Comm_S^{2'} \cap Comm_S^{AP}$, $S{-}t_{in}(X.p, T){\rightarrow}S'$ (or $S{-}t_{out}(X.p, T){\rightarrow}S'$), and procedure $Y.q$ is called. Then:

- $S'(X) = (running, pc_S(X), ci_S(X) \oplus proc(Y.q, p), pv_S(X))$.
- $S'(Y) = (taken, 1, [\,], pv_S(Y))$.

ad. 3

Suppose, $(X,Y) \in Comm_S^{2'} \cap Comm_S^{PP}$, $S{-}t_{in}(X.p,T){\rightarrow}S'$ (or $S{-}t_{out}(X.p,T){\rightarrow}S'$), and procedure $Y.q$ is called. Then:

- $S'(X) = (taken, pc_S(X), ci_S(X) \oplus proc(Y.q,p), pv_S(X))$.
- $S'(Y) = (taken, 1, [\,], pv_S(Y))$.

ad. 4

This case is similar to the first case. The new state is defined in the same way. The only difference is that one of these agents has performed its communication step earlier.

ad. 5

Let $X \in Comm_S^{1'} \cap \mathcal{A}_A$, $S{-}t_{in}(X.p,T){\rightarrow}S'$. Then:

- $S'(X) = (waiting, pc_S(X), ci_S(X) \oplus in(p|T), pv_S(X))$.

If the port p is used to collect values of one type only, then the $in(p)$ entry is used instead of $in(p|T)$.

ad. 6

This case is similar to the previous one, but the $out(p|T)$ (or $out(p)$) entry is used.

ad. 7

Let $X \in Comm_S^{1'} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p,T){\rightarrow}S'$, $p \notin \mathcal{P}_{proc}(X)$ (X calls a procedure of another agent), and $Y = context(X)$. Then:

- $S'(X) = (taken, pc_S(X), ci_S(X) \oplus in(p|T), pv_S(X))$.
- $S'(Y) = (waiting, pc_S(Y), ci_S(Y), pv_S(Y))$.

If the port p is used to collect values of one type only, then the $in(p)$ entry is used instead of $in(p|T)$.

ad. 8

This case is similar to the previous one, but the $out(p|T)$ (or $out(p)$) entry is used.

ad. 9

Suppose $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p,T){\rightarrow}S'$, $p \in \mathcal{P}_{proc}(X)$, and $Y = context(X)$. This means that Y is directly or indirectly performing the procedure $X.p$. Performing the $S{-}t_{in}(X.p,T){\rightarrow}S'$ step means that the procedure collects its input parameter. Let x be the second argument of the corresponding *in* statement and a value w was sent while the procedure call. If the *in* statement is not the last procedure statement then:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X)|_{x=w})$.

If a valueless communication is considered then $pv_S(X)$ remains unchanged.

If the *in* statement is the last procedure statement then the procedure is finished. Suppose, Y is indirectly performing the procedure $X.p$ and exist $X^1, \ldots, X^m$, $m > 0$ such that Y is performing a procedure of X^1, X^1 is performing a procedure of X^2, ..., X^m is performing input procedure $X.p$ via port p^m.

Let $called_S(X,P)$ denote the set of agents that *potentially called* one of X procedures from the set $P \subseteq \mathcal{P}_{proc}(X)$:

$$\begin{aligned} called_S(X,P) = \{Z\colon\ & ((Z \in \mathcal{A}_A \wedge am_S(Z) = waiting \wedge guard \notin ci_S(Z)) \vee \\ & \vee\ (Z \in \mathcal{A}_P \wedge am_S(context(Z)) = waiting \wedge guard \notin ci_S(Z))) \\ & \wedge\ (\exists p \in P,\ \exists q \in \mathcal{P}(Z)\colon \\ & ((X.p, Z.q) \in \mathcal{C} \wedge in(q|type(X.p)) \in ci_S(Z)) \vee \\ & \vee\ ((Z.q, X.p) \in \mathcal{C} \wedge out(q|type(X.p)) \in ci_S(Z)))\} \end{aligned} \tag{6.17}$$

The term *potentially called* means that it is possible that port $Z.q$ is connected with a few ports and the communication via this port can be finalized as a communication with another active agent or another passive one (different than X). When performing of a procedure $X.p$ is finished, guards of all procedures of X are evaluated and a set P of available procedures is received. If at least one of them has been already potentially called i.e. $called_S(X,P) \neq \emptyset$ then X starts another procedure immediately. If $card(called_S(X,P)) > 1$ then one agent with the highest priority is chosen.

Suppose, $Z \in \mathcal{A}_A$ is the chosen agent that starts performing a procedure $X.p'$ via port r. Then:

- $S'(X) = (taken, 1, [\,], pv_S(X)|_{x=w})$.
- $S'(Z) = (running, pc_S(Z), ci_S(Z) \oplus proc(X.p', r), pv_S(Z))$.
- $S'(X^m) = (taken, nextpc(S(X^m)), ci_S(X^m) \ominus proc(X.p, p^m), pv_S(X^m))$, if calling the procedure $X.p$ was not the last procedure block step for X^m. Otherwise, the corresponding X^m procedure has finished and the new state for X^m and X^{m-1} is determined as previously for X and X^m (if an input procedure has been called) or as described at point 10 (if an output procedure has been called).

Suppose, Y is directly performing the procedure $X.p$ via its port q. Then, the new state for X (and Z if any) is defined as above, and:

- $S'(Y) = (running, nextpc(S(Y)), ci_S(Y) \ominus proc(X.p, q), pv_S(Y))$

Suppose, $called_S(X, P) = \emptyset$. Besides agents belonging to the set $called_S(X, P)$, there may exist agents that are waiting for accessibility of X procedures in order to fulfil their *select* statements guards. Let the set $callable_S(X, P)$ be defined as follows:

$$\begin{aligned} callable_S(X,P) = \{Z\colon ((Z \in \mathcal{A}_A \wedge am_S(Z) = waiting \wedge guard \in ci_S(Z)) \vee \\ \vee\ (Z \in \mathcal{A}_P \wedge am_S(context(Z)) = waiting \wedge \\ \wedge\ guard \in ci_S(Z))) \\ \wedge \text{ accessibility of procedures belonging to } P \\ \text{makes at least one branch of the corresponding} \\ select \text{ statement open}\} \end{aligned} \tag{6.18}$$

Thus, the state S' is defined as follows:

- $S'(X) = (waiting, 0, ci'_S(X), pv_S(X)|_{x=w})$, where $ci'_S(X)$ contains all ports from P together with the direction of parameters sending e.g. $in(p_1)$, $out(p_2)$, etc.
- States of agents $Y, X^1, \ldots, X^m$ are defined as previously.
- $S'(Z) = (running, nextpc(S(Z)), ci_S(Z) \ominus guard, pv_S(Z))$, for any $Z \in callable_S(X, P) \cap \mathcal{A}_A$.
- $S'(Z) = (taken, nextpc(S(Z)), ci_S(Z) \ominus guard, pv_S(Z))$, for any $Z \in callable_S(X, P) \cap \mathcal{A}_P$.
- $S'(Z') = (running, pc_S(Z'), ci_S(Z'), pv_S(Z'))$, for any $Z \in callable_S(X, P) \cap \mathcal{A}_P$ and $Z' = context(Z)$.

In all above cases, if a valueless communication is considered then $pv_S(X)$ remains unchanged.

ad. 10

Suppose $X \in Comm_S^{1'} \cap \mathcal{A}_P$, $S{-}t_{out}(X.p, T){\rightarrow}S'$, $p \in \mathcal{P}_{proc}(X)$, and $Y = context(X)$. As previously, Y is directly or indirectly performing the procedure $X.p$. Performing the $S{-}t_{out}(X.p, T){\rightarrow}S'$ step means that the procedure returns its result. Let x be the second argument of the corresponding *out* statement and a value w is sent.

Suppose, Y is directly performing the procedure $X.p$ and the *out* statement is not the last procedure statement. In such a case, the state S' is defined as follows:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X))$.
- $S'(Y) = (running, pc_S(Y), ci_S(Y), pv_S(Y)|_{x=w})$.

If a valueless communication is considered then $pv_S(X)$ remains unchanged.

Suppose, Y is indirectly performing the procedure $X.p$ and exist $X^1, \ldots, X^m$, $m > 0$ such that Y is performing a procedure of X^1, X^1 is performing a procedure of X^2, ..., X^m is performing output procedure $X.p$ via one of its ports. In such a case, the state S' is defined as follows:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X))$.
- $S'(X^m) = (taken, pc_S(X^m), ci_S(X^m), pv_S(X^m)|_{x=w})$.

As previously, if a valueless communication is considered then $pv_S(X)$ remains unchanged.

If the *out* statement is the last procedure statement then states of agents changes as described at point 9. The only difference is the direction of a value transfer i.e. the parameters value list that is updated belongs to the agent that called the corresponding procedure.

Results of transitions performing for passive agents are defined similarly as for active ones. The only difference is the problem of the last statement in a procedure block. For example, if $X \in \mathcal{A}_P$, $S{-}t_{exec}(X){\rightarrow}S'$, a parameter v is assign a value a with the *exec* statement, and the statement is not the last one in the corresponding procedure block, then the state S' is defined as follows:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X)|_{v=a})$.

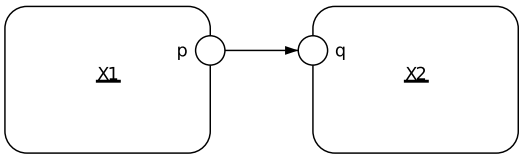
If the *exec* statement is the last one in the corresponding procedure block, then the procedure is finished and the state of the model changes as described previously for the t_{in} transition.

In similar way are defined new states for a transition $t \in \{if, jump, loop, null, select, start\}$. The *exit* statement always finishes the corresponding procedure. It cannot be placed before the procedure *in* (*out*) statement used to collect the procedure argument (return the procedure result).

6.5 LTS graphs

Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model. For a pair of states S, S' we say that S' is *directly reachable* from S iff there exists $t \in \mathcal{T}$ such that $S{-}t{\rightarrow}S'$. We say that S' is *reachable* from S iff there exist a sequence of states $S^1, \ldots, S^{k+1}$ and a sequence of transitions $t^1, \ldots, t^k \in \mathcal{T}$ such that $S = S^1{-}t^1{\rightarrow}S^2{-}t^2{\rightarrow}\ldots{-}t^k{\rightarrow}S^{k+1} = S'$. The set of all states that are reachable from the initial state S_0 is denoted by $\mathcal{R}(S_0)$.

States of an Alvis model and transitions among them are represented using a labelled transition system (LST graph for short). An LTS *graph* is directed graph LTS $= (V, E, L)$, such that $V = \mathcal{R}(S_0)$, $L = \mathcal{T}$, and $E = \{(S, t, S')\colon S{-}t{\rightarrow}S' \wedge S, S' \in \mathcal{R}(S_0)\}$. In other words, an LTS graph presents all reachable states and transitions among them in the form of the directed graph.




```
agent X1 {
  loop {        -- 1
    out p; }    -- 2
}

agent X2 {
  loop {        -- 1
    in q; }     -- 2
}
```

Fig. 6.5. Example 1.

To illustrate the idea of LTS graph let us consider two simple examples of Alvis models. The first model shown in Fig. 6.5 represents two active agents that communicate one with another. The $X1$ agent is a sender and $X2$ is a receiver. The LST graph for this model is shown in Fig 6.6. The graph is another approach to explain the rules of the Alvis communication between active agents.

The second model is presented in Fig. 6.7. It deals with a communication between an active and a passive or two passive agents. The states in the LTS graph illustrate the way agents states change while such a communication. The most interesting parts of these states are modes and context information lists.

The graphical form of LTS graphs presentation is very useful from users point of view. An LTS graph generated automatically for a model is stored in a textual file. For verification purposes such graphs are transformed into the *Binary Coded Graphs* (BCG) format. Finally, its properties are verified with the CADP toolbox [18]. CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking.

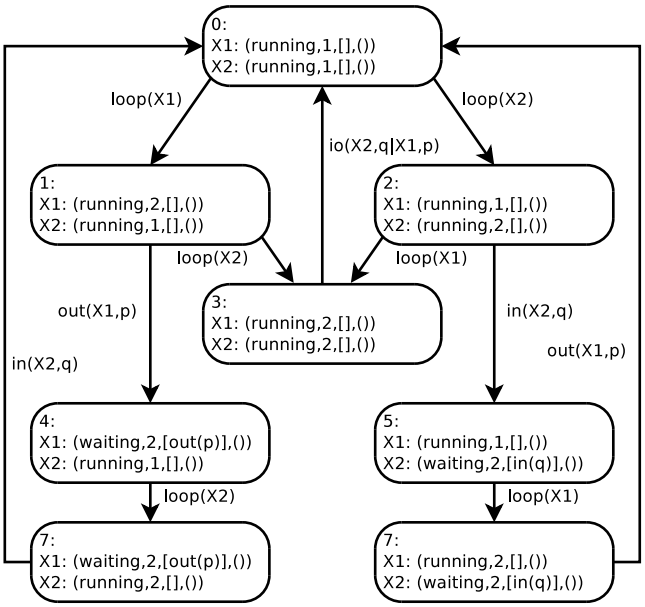
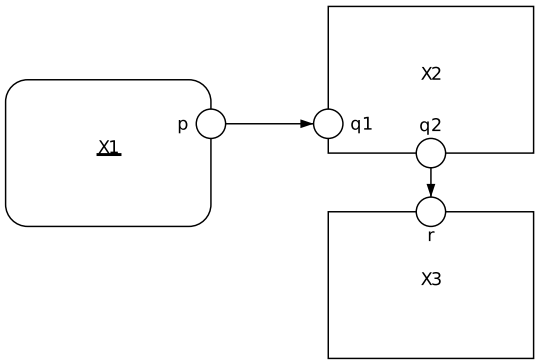


Fig. 6.6. Example 1 – LTS graph.




```
agent X1 {
  loop {                -- 1
    out p; }            -- 2
}

agent X2 {
  proc q1 { in q1;      -- 1
            out q2 }    -- 2
}

agent X3 {
  proc r { in r;        -- 1
           null; }      -- 2
}
```

Fig. 6.7. Example 2.

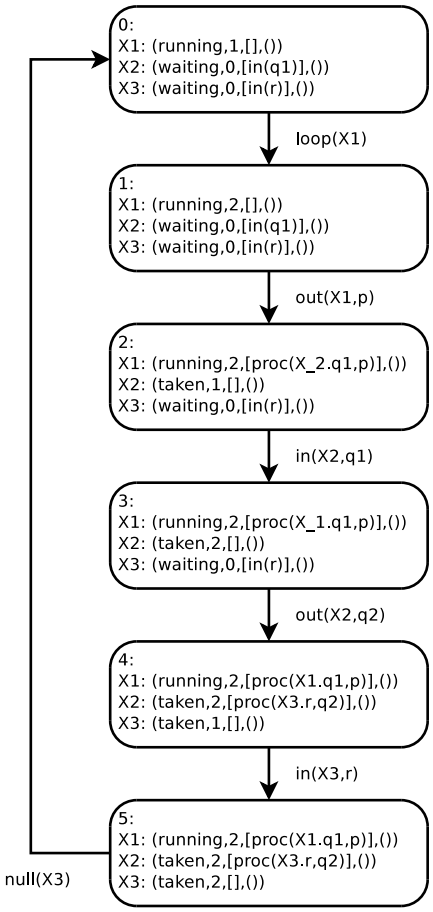


Fig. 6.8. Example 2 – LTS graph.

References

1. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
2. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1–3. Springer-Verlag, Berlin, Germany (1992-97)
3. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. Lecture Notes on Concurrency and Petri Nets **3098** (2004)
4. Szpyrka, M.: Analysis of RTCP-nets with reachability graphs. Fundamenta Informaticae **74**(2–3) (2006) 375–390
5. Szpyrka, M.: Petri nets for modelling and analysis of concurrent systems. WNT, Warsaw (2008) (in Polish).
6. Samolej, S., Rak, T.: Simulation and performance analysis of distributed internet systems using tcpns. Informatica (Slovenia) **33**(4) (2009) 405–415
7. Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science, Upper Saddle River, NJ, USA (2001)
8. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
9. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
10. Aceto, L., Ingófsdóttir, A., Larsen, K., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge, UK (2007)
11. Fencott, C.: Formal Methods for Concurrency. International Thomson Computer Press, Boston, MA, USA (1995)
12. Matyasik, P.: Design and analysis of embedded systems with XCCS process algebra. PhD thesis, AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Kraków, Poland (2009)
13. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994) 183–235
14. Szpyrka, M., Matyasik, P., Mrówka, R.: Alvis – modelling language for concurrent systems. In Bouvry, P., Gonzalez-Velez, H., Kołodziej, J., eds.: Intelligent Decision Systems in Large-Scale Distributed Environments. Volume 362 of SCI. Springer-Verlag (2011) 315–342
15. Szpyrka, M., Matyasik, P., Mrówka, R.: Practical approach to modelling and verification of concurrent systems with Alvis. In: Proc. of the 25th European Conference on Modelling and Simulation, Krakow, Poland (2011) 539–545

16. Szpyrka, M., Kotulski, L., Matyasik, P.: Specification of embedded systems environment behaviour with Alvis modelling language. In: Proc. of the 2011 International Conference on Embedded Systems and Applications ESA'11 (part of Worldcomp 2011), Las Vegas, Nevada, USA (July 18-21 2011)
17. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. O'Reilly Media, Sebastopol, CA, USA (2008)
18. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Computer Aided Verification (CAV'2007). Volume 4590 of LNCS., Berlin, Germany, Springer (2007) 158–163
19. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, London, UK (2008)
20. Balicki, K., Szpyrka, M.: Formal definition of XCCS modelling language. Fundamenta Informaticae **93**(1-3) (2009) 1–15
21. Barnes, J.: Programming in Ada 2005. Addison Wesley (2006)
22. ISO: Information processing systems, open systems interconnection LOTOS. Technical Report ISO 8807 (1989)
23. Object Management Group: OMG Systems Modeling Language (OMG SysML). (2008)
24. Ada Europe: Ada Reference Manual ISO/IEC 8652:2007(E) Ed. 3. (2007)
25. Burns, A., Wellings, A.: Concurrent and real-time programming in Ada 2005. Cambridge University Press (2007)
26. Esterel Technologies SA: Welcome to SCADE 6.0. (2007)
27. Jensen, K., Kristensen, L.: Coloured Petri nets. Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
28. Szpyrka, M., Szmuc, T.: Verification of automatic train protection systems with RTCP-nets. In Górski, J., ed.: Computer Safety, Reliability and Security. Volume 4166 of LNCS. Springer-Verlag (2006) 344–357