Chapter 15
Alvis – Modelling Language for Concurrent Systems

Marcin Szpyrka, Piotr Matyasik, and Rafał Mrówka

Abstract. The chapter presents a description of a novel modelling language called Alvis defined for the design of concurrent especially real-time systems. Alvis combines the advantages of formal methods and practical modelling languages. Based on CCS and XCCS process algebras, Alvis utilizes flexible graphical modelling of interconnections among agents and a high level programming language used for the description of agents behaviour. Each agent in a model can use rule-based systems to support its decisions. A small set of language statements and graphical concepts make Alvis easy to learn and use. The possibility of a formal model verification, makes Alvis a formal modelling language. Alvis modelling environment creates in parallel a model of the considered system and a labelled transition system (LTS graph) that is its formal representation. The LTS graph can be formally verified with the CADP toolbox. A survey of main Alvis features from practical point of view, is given in the chapter.

15.1 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

Marcin Szpyrka
AGH University of Science and Technology, Department of Automatics, Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: mszpyrka@agh.edu.pl

Piotr Matyasik
AGH University of Science and Technology, Department of Automatics, Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: ptm@agh.edu.pl

Rafał Mrówka
AGH University of Science and Technology, Department of Automatics, Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: Rafal.Mrowka@agh.edu.pl

P. Bouvry et al. (Eds.): Intelligent Decision Systems, SCI 362, pp. 315–341.
springerlink.com

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 [18], [15], [7], [24], [26], [23], process algebras [8], [13], [17], [1], [10], [16] or time automata [3], [7] 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 [1] is a novel modelling language designed especially for concurrent systems. Our goal was to combine formal and practical modelling languages. Alvis is a successor of the XCCS language [5], [16], which is an extension of the CCS process algebra [17], [10], [1]. 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 distinguished parts of the system under consideration that are called *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 [22] 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. It gathers information about all agents in a model and their states. Moreover, it provides some of the model meta-data to agents.

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 is encoded using *Binary Coded Graphs* (BCG) format. The CADP toolbox [11] and model checking techniques [4] are used to verify its properties.

Rule-based systems are widely used in various kinds of computer systems, e.g. expert systems, decision support systems, monitoring and control systems, diagnostic systems, etc. They constitute an easy way of knowledge encoding and interpretation [12], [19], [20]. Alvis uses Haskell to define parameters, data types and data manipulation functions. Encoding a rule-based system as a Haskell function is the easiest way to include the system into the corresponding Alvis model. Thus, an agent in an embedded or a distributed system can use a rule-based system to take decisions upon data collected from its sensors.

[1] Project web site: `http://fm.ia.agh.edu.pl`

The aim of the chapter is to provide a description of both graphical and textual (code) parts of Alvis models and to present a method of encoding rule-based systems in Haskell and including them into Alvis models. The chapter is organised as follows. Section 15.2 provides a short comparison of Alvis with other languages used for embedded and distributed systems development. Section 15.3 deals with communication diagrams, while Section 15.4 provides a survey of Alvis language statements. System layers are described in Section 15.5. Section 15.6 deals with encoding rule-based systems using Haskell in order to include them into Alvis models. A small example of an Alvis model with a rule-based system included is given in Section 15.7. A short description of models states and transitions among them is presented in Section 15.8. A short summary is given in the final section.

15.2 Related Works

This section provides a short comparison of Alvis with other modelling languages used in industry for the embedded or distributed systems development.

E-LOTOS is an extension of the LOTOS modelling language (Language Of Temporal Ordering Specification) [14]. 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)[21] 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 [2], [6], [9]. 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 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.

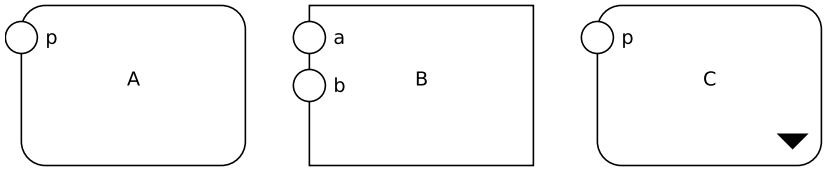

Fig. 15.1 Agents (from left): active, passive, hierarchical

15.3 Communication Diagrams

The key concept of Alvis is *agent*. The name has been taken from the CCS process algebra [17] and 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 [6], [9]. 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*).

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) are underlined. Hierarchical agents are indicated by black triangles. Graphical representation of Alvis agents is shown in Fig. 15.1.

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. There is no distinction between input and output ports on communication diagrams. Any port can be used as an input or output one. 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 next to the port. It must fulfill the same requirements as agents' identifiers but its first character must be a lower-case letter.

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. Communication channels are drawn as lines (or broken lines). An arrowhead points out the input port for the particular

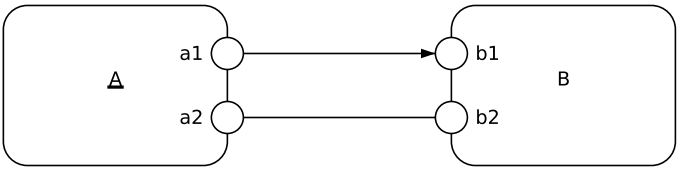

Fig. 15.2 One-way and two-way communication channels

connection. Communication channels without arrowheads represent pairs of connections with opposite directions. A connection between two active agents creates a synchronisation point between them. On the other hand, a connection between an active and a passive agent or between two passive agents is similar to a procedure call. Examples of communication channels are shown in Fig. 15.2.

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 active agent on one level can be replaced by a page on the lower level, which usually gives a more precise and detailed description of the activity represented by the agent. 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.

A hierarchical agent and its subpage are joined together using so-called *binding function* that maps ports of the hierarchical agent to the *join ports* of the corresponding subpage. The *join ports* of the subpage are those ports of agents from the page whose names are the same as those of the hierarchical agent. There are two kinds of substitution called *simple* and *extended* one. In the case of the simple substitution, the *binding function* is a bijection. It means that each port of the hierarchical agent has exactly one corresponding port in the subpage. On the other hand, in the case of the extended substitution, one port of the hierarchical agent may have more than one join port assigned on the subpage.

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 presented model uses the extended substitution mechanism.

Fig. 15.3 presents the main page of the communication diagram for the considered system. The *Readers* agent stands for the set of readers used in the model, while the *Writers* one stands for writers. The primary page will stay unchanged, if we decide to change the number of readers or writers in the model. Subpages for these hierarchical agents are shown in Fig. 15.4 and 15.5 respectively.

The structure of a communication diagram is represented using a labelled directed graph called *page hierarchy graph*. Nodes of such a graph represent pages, while arcs (labelled with names of hierarchical agents) represent the *substitution function* that maps hierarchical agents to their subpages. Of course the number of

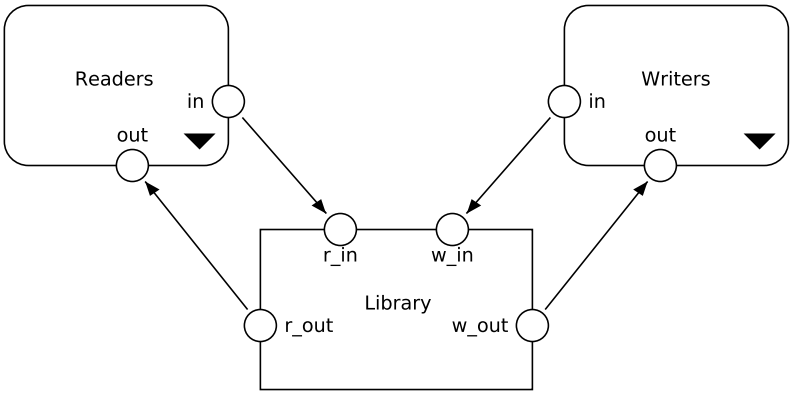

Fig. 15.3 Readers-Writers model – top level (primary) page

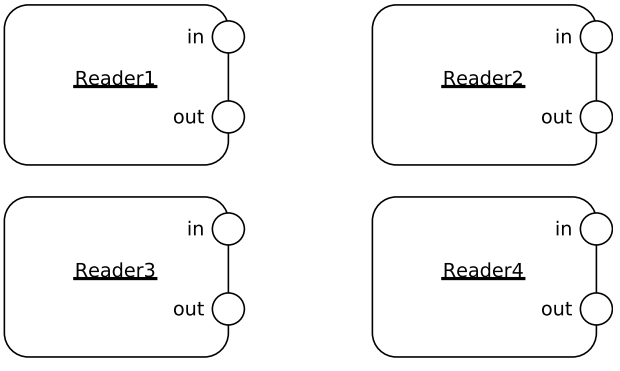

Fig. 15.4 *PReaders* subpage

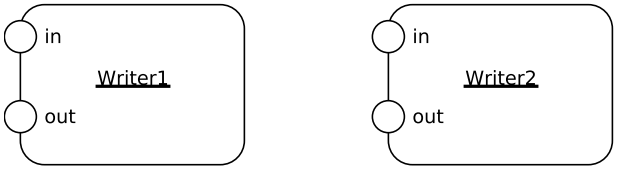

Fig. 15.5 *PWriters* subpage

pages must be greater than the number of hierarchical agents. Pages that are not used as subpages are called *primary pages*. They are roots of the trees that constitute the page hierarchy graph.

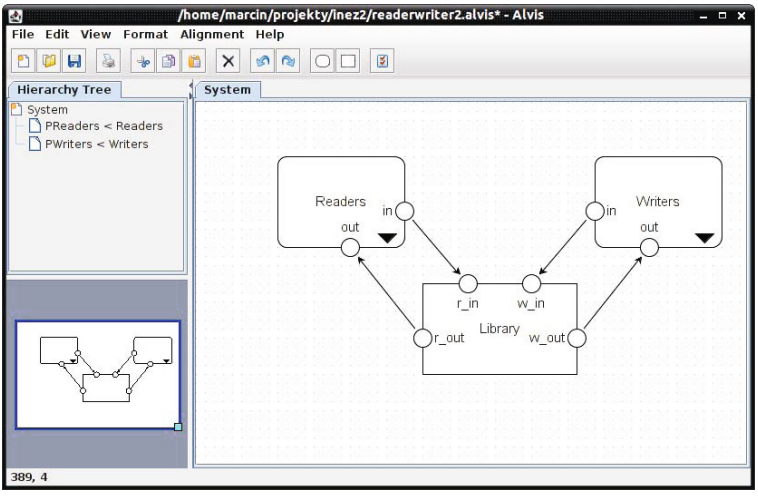

Fig. 15.6 Inez 2 editor – Readers-Writers model with page hierarchy graph

Modelling concurrent systems with Alvis is supported by so-called *Alvis Toolkit*. The toolkit, among other things, contains *Inez 2* editor for designing Alvis models[2]. A screenshot of the editor is shown in Fig. 15.6. The top-left part of the window contains the page hierarchy graph for the Readers-Writers model.

Both substitutions used in the considered model are the extended ones. The equivalent flat (non-hierarchical) communication diagram is presented in Fig. 15.7.

Ports that are not used in any connection are treated as the ones for communication with the considered system environment. It is possible to define in the code layer all details of signals that a systems collects from its environment or sends to it. Such ports are called *border* ones and they must have unique names.

15.4 Language Statements

As it has already been said, the *code layer* is used to describe the behaviour of individual agents. The layer uses both Haskell functional language and original Alvis statements. To be more convenient from the engineering point of view, Alvis uses statements typical for high level programming languages instead of algebraic equations used in the CCS process algebra.

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.

[2] All diagrams presented in the paper have been designed with the Inez 2 editor. For more details visit the Alvis website: `http://fm.ia.agh.edu.pl`

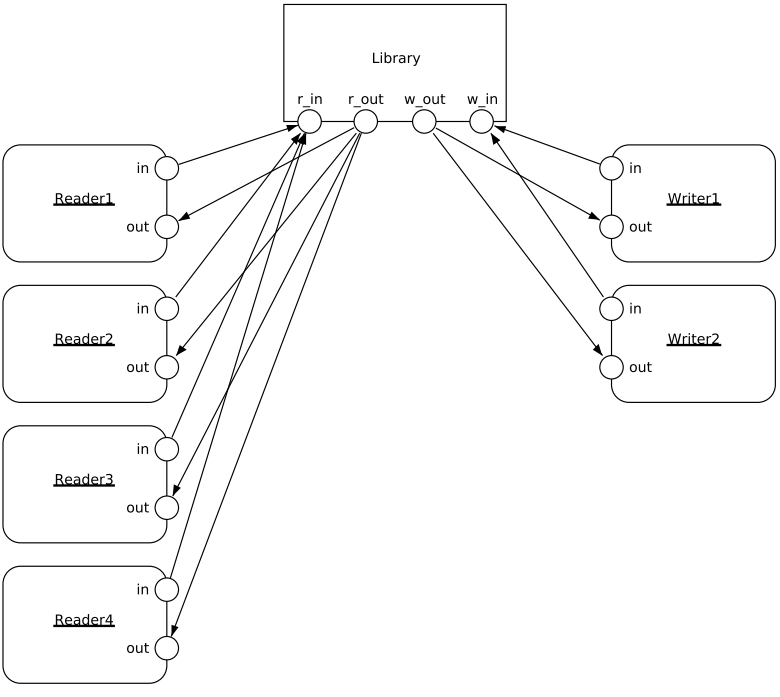

Fig. 15.7 Readers-Writers model – flat communication diagram

Inez 2 editor uses an XML file format to store both graphical and code layers. The general structure of the code layer is shown in Listing 15.1.

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

-- Implementation:
--   agents
```

Listing 15.1 Structure of the code layer

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 15.2. 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.

```
agent AgentName;
-- declaration of parameters
-- agent body
```

Listing 15.2 Structure of an agent block

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 15.3). 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.

```
[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 15.3 Examples of Haskell composite data types

To make the source code more readable, one can introduce a synonym for an existing type using the *type* keyword or define a new data type using the *data* keyword.

The *data* statement can be used to define also the so-called algebraic data types, e.g. an enumeration type. Examples of type synonyms and definitions of new data types are shown in Listing 15.4. Moreover, Haskell supports the structure data type. For more details see for example [22].

```
type AgentID = Int
type InputData = (Int,Int)  -- pair
type TrafficSignal = [Char] -- list
data AgentDescription = AgentDesc Int String [String]
data Move = East | South | West | North
```

Listing 15.4 Examples of type synonyms and definitions of new data types

Constants are defined using parameterless Haskell functions, e.g. `name = "A";`. 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 15.5.

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

Listing 15.5 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 15.6 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 15.6).

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

Listing 15.6 Examples of using the *exec* statement

Alvis provides a typical *if else* statement with optional *elseif* clauses. Statements performed within the *if*, *else* or *elseif* clauses must be put inside curly brackets.

Some Alvis statements contain so-called *guards*. Guards are logical expressions, written in Haskell, placed inside round brackets. They are used for example, as conditions for the *if else* statement. The general syntax of the conditional statement is shown in Listing 15.7 – $g1$, $g2$ and $g3$ stand for guards.

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

Listing 15.7 Syntax of the conditional statement

Recursion is one of the two mechanisms 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). The *jump* statement is the key statement for translating algorithms from CCS to Alvis.

Moreover, Alvis provides three kinds of *loop* statements, as shown in Listing 15.8. The first one is the most general *loop* statement. It repeats its contents infinitely. The second loop repeats its contents while the guard (g) is satisfied, the guard is checked every time before entering the loop contents. The loop is similar to the while loop in most languages. The last statement repeats its contents every *ms* milliseconds.

```
loop {...}
loop (g) {...}
loop (every ms) {...}
```

Listing 15.8 Syntax of loop statements

In order to allow for the description of agents whose behaviour may follow different alternative paths, Alvis offers the *select* statement (see Listing 15.9). The statement is similar to the basic *select* statement from the Ada programming language, 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.

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 in milliseconds. The statement is also used to define time-outs. A branch may contain the *delay* as its guard (see

```
select {
  alt (g1) {...}
  alt (g2) {...}
  alt (g3) {...}
  ...
}
```

Listing 15.9 Syntax of the *select* statement

Listing 15.10). In such a case, the third branch will be open after *ms* milliseconds. Thus, if all branches are closed, the corresponding agent waits *ms* milliseconds and follows the last branch. However, if at least one branch is open before the delay goes by, then the delay is cancelled. If necessary a branch may contain only the *null* statement.

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

Listing 15.10 Syntax of the *select* statement with a time-out

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 (i.e. one-way or two-way connections);
2. Statements used in the code layer.

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 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 15.11 (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.

Passive agents are used to store data shared among agents and to avoid the simultaneous use of such data by two or more agents. They provide a set of procedures that can be called by other agents. Each procedure has its own port attached and a communication with a passive agent via that port is treated as the corresponding procedure call. Depending on the communication direction, such a procedure may be used to send or collect some data from the passive agent. Each procedure

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

Listing 15.11 Syntax of the *in*/*out* statements

is defined with the *proc* statement that is followed by a guard (optionally) and the corresponding port name. The procedure is accessible for other agents only if the guard evaluates to *True*.

```
agent Buffer {
  i :: Int = 0;
  proc pop  { out pop i; }
  proc push { in  push i; }
}
```

Listing 15.12 Example of a passive agent definition

Alvis models can use so-called *border ports* i.e. ports without any connections that are treated as communication channels with the 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. Using *in* and *out* clauses, a designer can specify both values sent through the corresponding port and time points (in milliseconds), when the port can be used. 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.

If a border port is used both as an input and 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. If a border port is used for a parameterless communication, then the first list is empty. Similarly, if a border port is always accessible, then the second list is empty. Lists are defined using the Haskell language. In particular, it is possible to use infinite lists [22].

Border ports names must be unique in a model. It is possible to use a border port name more than once, but it means that more than one agent can send (or collect) signals through the same border port.

Let us consider the border ports presented in Listing 15.13.

p1 – at any time point one of the values 0 or 1 (at random) can be collected through the port;

```
in p1 [0,1] [];
in p2 Bool [];
in p3 [0,1,5] (map (10*) [0..]);
in p4 [1] [1000,2000,3000];
out p5 [0,1] [];
out p5 Bool [];
out p6 [] [];
```

Listing 15.13 Examples of border ports specification

p2 – at any time point a Boolean value can be collected through the port;
p3 – every 10 ms one of the values 0, 1 or 5 can be collected through the port;
p4 – three times at given time points the value 1 can be collected through the port;
p5 – at any time point a Boolean value or 0 or 1 can be sent through the port;
p6 – at any time point a parameterless signal can be sent through the port.

15.5 System Layers

As it has already been said, the *system layer* is the third 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. From the users point of view, the layer is the predefined one and it works in the read-only mode. Agents can retrieve some data from the layer, but they cannot directly change them. The system layer provides some functions that are useful for the implementation of scheduling algorithms or for retrieving information about other agents states. An example of such a system layer function is the *ready* function that 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.

A user can choose one of a few versions of the layer and it affects the model semantic. System layers differ about the scheduling algorithm and system architecture mainly. There are two approaches to the scheduling problem considered. System layers with α symbol provide a predefined scheduling function that is called after each step automatically. On the other hand, system layers with β symbol do not provide such a function. A user must define a scheduling function himself.

Both α and β symbols are usually extended with some indicators put in the superscript or/and subscript. An integer put in the superscript denotes the number of processors in the system. Zero is used to denote the unlimited number of processors. A symbol put in the subscript denotes the selected system architecture or/and chosen scheduling algorithm.

In this paper we will consider only the α^0 system layer. This layer makes Alvis a universal formal modelling language similar to Petri nets or process algebras. The α^0 system layer scheduler is based on the following assumptions.

- Each active agent has access to its own processor and performs its statements as soon as possible.
- The scheduler function is called after each statement automatically.
- 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 active 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 α^0 system layer is the most suitable one for distributed systems. In such a case, each agent is an autonomous system with its own processor. If embedded systems are considered, the β line of system layers can be more interesting. These system layers allow testing an embedded system with different scheduling algorithms.

15.6 Rule-Based Systems

Many active agents check sensors providing information about the system's environment and/or collects information from another agents and take actions depending on the collected data. An agent may be implemented to use a rule-based system to make decisions.

In the presented approach a rule-based system takes the form of a decision table with non-atomic values of attributes. Each cell of such a decision table should contain a formula, which evaluates to a Boolean value for condition attributes, and to a single value (that belongs to the corresponding domain) for decision attributes ([25]). It means that for any condition attribute we can write a formula that describes suitable values of the attribute in the corresponding rule. On the other hand, for any decision attribute we can write a formula that contains names of condition attributes and evaluates to a single value belonging to the domain of the decision attribute.

Let us consider the decision table presented in Table 15.1. The three condition attributes f, r and l stand for infra-red sensors readings. They can take any integer value from 0 to 255. The higher the value the further the object is. The front sensor f is mounted directly in the centreline of a robot while the left l and right r one are mounted 40 degrees to the left and to the right respectively. The *thres* (threshold) parameter distinguishes near and far objects and can be adjusted according to needs. The *rm* and *lm* attributes are motors directions where $+1$ means forward movement, -1 backward movement and 0 means stop. In the decision rule 6, the formula r means that any value of the attribute r is possible.

To be useful, a decision table should satisfy some qualitative properties, such as completeness, consistency (determinism), etc. Let us focus on the following three properties:

I. A decision table is considered to be *complete* if for any possible input situation at least one rule can produce a decision.

Table 15.1 Decision table for obstacle avoidance

	l	f	r	lm	rm
1	$l > thres$	$f > thres$	$r > thres$	$+1$	$+1$
2	$l \leq thres$	$f \leq thres$	$r \leq thres$	-1	-1
3	$l \leq thres$	$f \leq thres$	$r > thres$	$+1$	-1
4	$l \leq thres$	$f > thres$	$r > thres$	$+1$	0
5	$l > thres$	$f > thres$	$r \leq thres$	0	$+1$
6	$l > thres$	$f \leq thres$	r	-1	$+1$
7	$l \leq thres$	$f > thres$	$r \leq thres$	-1	$+1$

II. A decision table is *deterministic* (consistent) if no two different rules can produce different results for the same input situation.
III. A decision table is *optimal* if any rule belonging to it is independent. Let R be a complete and consistent set of decision rules. A rule r is *independent* if the set $R - \{r\}$ is not complete. A rule r is *dependent* if the rule is not independent.

The formal definitions of those properties can be found in [25]. The presented decision table has been verified after transformation to Haskell code.

A decision table can be treated as a function that takes values of condition attributes as its arguments and provides values of decision attributes as its result. Listing 15.14 presents Haskell implementation of the decision table shown in Table 15.1. The function maps a triple of type `Condition` to a pair of type `Decision`. For the sake of simplicity built-in Haskell types have been used as attributes domains. However, the types will be reduced to expected ranges while the decision table verification.

```
type Condition = (Int, Int, Int)
type Decision  = (Int, Int)

rbs :: Condition -> Decision
rbs (l,f,r) | l > 30  && f > 30  && r > 30  = (1,1)
rbs (l,f,r) | l <= 30 && f <= 30 && r <= 30 = (-1,-1)
rbs (l,f,r) | l <= 30 && f <= 30 && r > 30  = (1,-1)
rbs (l,f,r) | l <= 30 && f > 30  && r > 30  = (1,0)
rbs (l,f,r) | l > 30  && f > 30  && r <= 30 = (0,1)
rbs (l,f,_) | l > 30  && f <= 30            = (-1,1)
rbs (l,f,r) | l <= 30 && f > 30  && r <= 30 = (-1,1)
```

Listing 15.14 Decision table as the *rbs* function

Haskell functions can be defined piece-wise, meaning that we can write one version of a function for certain parameters and then another version for other parameters. Moreover, the so-called *pattern matching* can be used, in which a sequence of syntactic expressions called *patterns* is used to choose between a sequence of results of the same type. If the first pattern is matched, then the first result is chosen;

otherwise the second pattern is checked, etc. For the 6th rule, the *wild card* pattern _ (underscore) is used that matches any value. Using the wild card pattern we can indicate that we do not care what is present in part of a pattern. Of course, more than one wild card can be used in a single pattern.

The *rbs* function shown in Listing 15.14 has been defined using both patterns and guard expressions. Guard expressions are logical expressions used as the second step of choosing the appropriate piece of a function definition. In this case, guards have been defined using values of condition attributes – the symbol | is read as "such that".

```
states :: [Condition]
states = [(l,f,r) | l <- [0 .. 255], f <- [0 .. 255],
                    r <- [0 .. 255]]

allDecisions' :: Int -> Condition -> [(Int, Decision)]
allDecisions' i (l,f,r)
  | i == 1 && l > 30  && f > 30  && r > 30
    = [(1, (1,1))] +!+ allDecisions' (i + 1) (l,f,r)
  | i == 2 && l <= 30 && f <= 30 && r <= 30
    = [(2, (-1,-1))] +!+ allDecisions' (i + 1) (l,f,r)
  | i == 3 && l <= 30 && f <= 30 && r > 30
    = [(3, (1,-1))] +!+ allDecisions' (i + 1) (l,f,r)
  | i == 4 && l <= 30 && f > 30  && r > 30
    = [(4, (1,0))] +!+ allDecisions' (i + 1) (l,f,r)
  | i == 5 && l > 30  && f > 30  && r <= 30
    = [(5, (0,1))] +!+ allDecisions' (i + 1) (l,f,r)
  | i == 6 && l > 30  && f <= 30
    = [(6, (-1,1))] +!+ allDecisions' (i + 1) (l,f,r)
  | i == 7 && l <= 30 && f > 30  && r <= 30
    = [(7, (-1,1))]
  | i > 7
    = []
  | otherwise
    = allDecisions' (i + 1) (l,f,r)

allDecisions :: Condition -> [(Int, Decision)]
allDecisions = allDecisions' 1

notCovered :: Condition -> Bool
notCovered (l,f,r) = allDecisions (l,f,r) == []

notCoveredStates :: [Condition]
notCoveredStates = filter notCovered states
```

Listing 15.15 Completeness verification code

Let us focus on the completeness property. To check the property, we have to generate the state space for the rule-based system under consideration. An argumentless

function *states* is used for this purpose (see Listing 15.15). The function takes the attributes domains under consideration and generates a list of all admissible input states. To verify a Haskell implementation of a decision table, we have to define another function that determines all possible decisions for an input situation or generates an empty list, if no decision can be undertaken. The *allDecisions* function is used for this purpose. The function checks all rules and generates the list of all possible decisions for a given input state. The list contains pairs – a decision rule number and the corresponding result. The + ! + operator, used in the code, states for lists concatenation with dropping duplicates.

The result of the completeness analysis is a list of input states that are not covered by decision rules. To check whether an input state is covered the *notCovered* function is used (see Listing 15.15). The function is used by the *notCoveredStates* function, which filters not covered states from the list generated by the *states* function.

The other properties are verified in similar way. The results of the considered rule-based system verification are shown in Listing 15.16.

```
*RBSRobot> notCoveredStates
[]
*RBSRobot> notDeterministicStates
[]
*RBSRobot> independentRules
[1,2,3,4,5,6,7]
```

Listing 15.16 Verification results – GHCi shell log

Overall, the function *rbs* presented in Listing 15.14 is ready to be included into an Alvis model.

15.7 Alvis Model Example

This section presents an Alvis model of a wheels' motors control system (WMC system for short) for a mobile robot. The system uses the *rbs* function presented in the previous section.

The primary (top level) page of the model is given in Fig. 15.8. It contains a single hierarchical agent that stands for the whole system. All ports of the agent are border ports used to collect sensors readings or to send control decisions to the system environment. The subpage assigned with the hierarchical agent is presented in Fig. 15.9. It contains two agents that work concurrently. The *Obstacle* agent is responsible for collecting infrared sensors readings and taking decisions for the *Movement* agent. The *Obstacle* agent uses the rule-based system presented in Section 15.6. The *Movement* agent controls the wheels motors.

The preamble of the model code layer is presented in Listing 15.17. The *rbs* function is placed inside it.

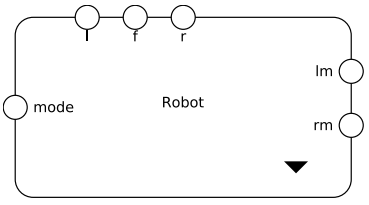

Fig. 15.8 Primary page of the WMC system

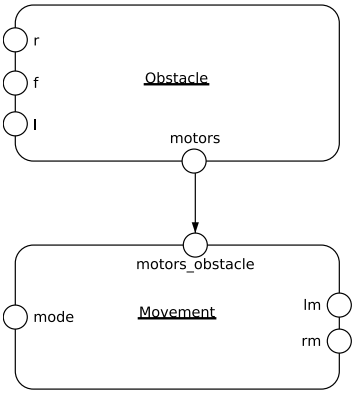

Fig. 15.9 Subpage of the WMC system

```
type Move = (Int, Int);
data Mode = Stop | Forward | Collide | Obstacle;
type Condition = (Int, Int, Int);
type Decision  = (Int, Int);

rbs :: Condition -> Decision;
rbs (l,f,r)
-- ...

environment {
  out l [0..255] [];
  out f [0..255] [];
  out r [0..255] [];
  out mode Mode [];
  in lm Int [];
  in rm Int [];
}
```

Listing 15.17 WMC system code layer – preamble

The *Obstacle* agent definition is given in Listing 15.18. The main statement in the code is an infinite loop. The agent collects infrared sensors readings, uses the

```
agent Obstacle {
  m  :: Motors = (0,0);
  lv :: Int = 255;
  fv :: Int = 255;
  rv :: Int = 255;
  loop {
    in l lv;
    in f fv;
    in r rv;
    m = rbs (l,f,r);
    if(ready [out(motors)]) {
      out motors m;
    }
  }
  delay 20;
}
```

Listing 15.18 WMC system code layer – agent *Obstacle* definition

rbs function to take a decision, send the decision through the *motors* port and finally delays 20 milliseconds.

```
agent Movement{
  rmv :: Int = 0;
  lmv :: Int = 0;
  m :: Motors = (0,0);
  mod :: Mode = Stop;
  loop {
    if(ready [in(mode)]) {
      in mode mod; }
    if (mod == Forward) {
      out lm 1;
      out rm 1; }
    elseif(mod == Collide) {
      in motorsObstacle m;
      lmv = snd m;
      rmv = fst m;
      out lm lmv;
      out rm rmv; }
    elseif(mod == Obstacle) {
      in motorsObstacle m;
      lmv = fst m;
      rmv = snd m;
      out lm lmv;
      out rm rmv; }
    else {
      out lm 0;
      out rm 0; }
}
```

Listing 15.19 WMC system code layer – agent *Movement* definition

The *Movement* agent definition is given in Listing 15.19. The main statement in the code is also an infinite loop. The agent collects a decision through the *motors_obstacle* port (if any) and sends control decisions to wheels motors.

15.8 Agent and Model State

An Alvis model is a triple that contains a communication diagram, a code layer and a system layer. A state of a model is represented as a sequence of agents states. To describe the current state of an agent, we need a tuple with four pieces of information:

- agent mode (*am*);
- program counter (*pc*);
- context information list (*ci*);
- parameters values tuple (*pv*).

Let us focus on passive agents firstly. A passive agent is always in one of two modes: *waiting* or *taken*. The former one means that the agent is inactive and waits for another agent to call one of its accessible procedures. In such a situation the *program counter* is equal to zero and the *context information list* contains names of accessible procedures. In any state, the *parameters values list* contains the current values of the agent parameters. The *taken* mode means that one of the passive agent procedures has been called and the agent executes it. In such a case, *ci* contains the name of the called procedure (i.e. the name of the port used for current communication). The *pc* points out the index of the next statement to be executed or the current statement if the corresponding active agent is *waiting*.

An active agent can be in one of the following modes: *finished, init, ready, running, waiting*. (The *ready* is not used with the α^0 system layer.) 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. If an agent is in the *init* mode, its *pc* is equal to zero and *ci* is empty.

The *finished* mode means that an agent has finished its work or it has been terminated using the *exit* statement. The statement is argumentless and an agent can terminate its work itself only. If an agent is in the *finished* mode, its *pc* is equal to zero and *ci* is empty.

The *waiting* mode means that an active agent is waiting either for a synchronous communication with another active agent, or for a currently inaccessible procedure of a passive agent. In such a case, the *pc* points out the index of the current statement and *ci* contains names of the agent ports that can be used for the desired communication.

The last mode *running* used here means that an agent is performing one of its statements. If it is a synchronous communication with another active agent or a procedure call, then the used port's name and the other agent's name (for procedures) are placed into *ci*. The *pc* points out the index of the current (e.g. for procedure call)

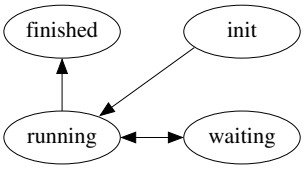

Fig. 15.10 Possible transitions among modes (without the *ready* mode).

or next agent statement. All possible transitions among modes of an active agent are shown in Fig. 15.10.

It is very important to explain the way Alvis statements are marked with numbers.

- We say that *pc points out* an *exec* (*exit, in, jump, null, out, start*) statement iff the next statement to be executed is an *exec* (*exit, in, jump, null, out, start*) statement.
- We say that *pc points out* an *if* statement iff the next statement to be executed is the evaluating of the guard and entering one of the *if* statement alternatives.
- We say that *pc points out* a *loop* statement iff the next statement to be executed is the evaluating of the guard (if any) and entering the *loop* statement.
- We say that *pc points out* a *select* statement iff the next statement to be executed is entering the *select* statement and possibly one of its branches.

It is worth emphasizing the difference between 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.

```
0:
Obstacle: (running,1,[],((0,0),255,255,255))
Movement: (running,1,[],(0,0,(0,0),Stop))
```

Listing 15.20 WMC system – initial state

```
4:  Obstacle: (running,3,[],((0,0),50,255,255))
    Movement: (running,3,[],(0,0,(0,0),Stop))
        in(Obstacle.f(90)) -> 5
5:  Obstacle: (running,4,[],((0,0),50,90,255))
    Movement: (running,3,[],(0,0,(0,0),Stop))
        in(Obstacle.r(100)) -> 6
6:  Obstacle: (running,5,[],((0,0),50,90,100))
    Movement: (running,3,[],(0,0,(0,0),Stop))
        in(Movement/Obstacle) -> 7
7:  Obstacle: (running,5,[],((0,0),50,90,100))
    Movement: (running,4,[],(0,0,(0,0),Obstacle))
        exec(Obstacle/rbs) -> 8
8:  Obstacle: (running,6,[],((1,1),50,90,100))
    Movement: (running,4,[],(0,0,(0,0),Obstacle))
        if(Movement) -> 9
9:  Obstacle: (running,6,[],((1,1),50,90,100))
    Movement: (running,7,[],(0,0,(0,0),Obstacle))
        if(Movement) -> 10
10: Obstacle: (running,6,[],((1,1),50,90,100))
    Movement: (running,13,[],(0,0,(0,0),Obstacle))
        if(Movement) -> 11
11: Obstacle: (running,6,[],((1,1),50,90,100))
    Movement: (running,14,[],(0,0,(0,0),Obstacle))
        in(Movement.motorsObstacle) -> 12
12: Obstacle: (running,6,[],((1,1),50,90,100))
    Movement: (waiting,14,[out(Obstacle.motors)],
                          (0,0,(0,0),Obstacle))
        if(Obstacle) -> 13
13: Obstacle: (running,7,[],((1,1),50,90,100))
    Movement: (waiting,14,[out(Obstacle.motors)],
                          (0,0,(0,0),Obstacle))
        out(Obstacle.motors) -> 14
14: Obstacle: (running,8,[],((1,1),50,90,100))
    Movement: (waiting,14,[out(Obstacle.motors)],
                          (0,0,(0,0),Obstacle))
        delay(Obstacle) -> 15
15: Obstacle: (waiting,1,[timer(20)],((1,1),50,90,100))
    Movement: (waiting,14,[out(Obstacle.motors)],
                          (0,0,(0,0),Obstacle))
        in(Movement.motorsObstacle) -> 16
16: Obstacle: (waiting,1,[timer(20)],((1,1),50,90,100))
    Movement: (running,14,[],(0,0,(0,0),Obstacle))
        in(Movement.motorsObstacle) -> 17
17: Obstacle: (waiting,1,[timer(20)],((1,1),50,90,100))
    Movement: (running,15,[],(0,0,(1,1),Obstacle))
```

Listing 15.21 WMC system – part of the LTS graph

The initial state for the considered WMC system is presented in Listing 15.20. We consider behaviour of Alvis models at the level of detail of single steps. Each Alvis statement is treated as a single step transition. Thus, we consider, for example, t_{exec} transition (executing the *exec* statement), t_{loop} transition (entering a loop) etc. Each step is realised in the context of one active agent. Also procedures of passive agents are realised in the context of active agents that called them.

States of an Alvis model and transitions among them are represented using a labelled transition system (LST graph for short). A LTS graph is an ordered graph with nodes representing states of the considered system and edges representing transitions among states. A small part (the textual representation) of the LTS graph for the WMC system is presented in Listing 15.21.

A LTS graph is generated automatically for a model and 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 (*Construction and Analysis of Distributed Processes*) toolbox [11]. CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking.

15.9 Summary

An informal description of the Alvis modelling language has been given in the chapter. Defined for the concurrent systems design, Alvis seems to be more useful for engineers than classical formal methods. The main differences between Alvis and formal methods, especially process algebras, are: the syntax that is more user-friendly from engineers point of view, and the visual modelling language (communication diagrams) that is used to define communication among agents.

The Alvis language can be used for modelling systems that are composed of distributed elements that work concurrently. Thus, it is suitable both to design embedded systems with concurrent processes and distributed systems with many communicating nodes. One of the main advantages of the language is the possibility of models formal verification using proven model checking techniques. An LTS graph is a formal representation of the considered concurrent system. The properties of the LTS graph can be formally verified with utilization of the CADP toolbox.

Due to the fact that many embedded systems or agents in multiagent systems use rule-based systems to support the decision process, Alvis has been equipped with a possibility of including decision tables into the model. As shown in the chapter, encoding a decision table as a Haskell function allows a designer to include a rule-based system into the code layer of an Alvis model.

Acknowledgements. The paper is supported by the Alvis Project funded from 2009-2010 resources for science as a research project.

References

1. Aceto, L., Ingófsdóttir, A., Larsen, K., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007)
2. Ada Europe: Ada Reference Manual ISO/IEC 8652:2007(E), 3 edn. (2007)
3. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, London (2008)
5. Balicki, K., Szpyrka, M.: Formal definition of XCCS modelling language. Fundamenta Informaticae 93(1-3), 1–15 (2009)
6. Barnes, J.: Programming in Ada 2005. Addison-Wesley, Reading (2006)
7. Bengtsson, J.E., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
8. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra. Elsevier Science, Upper Saddle River (2001)
9. Burns, A., Wellings, A.: Concurrent and real-time programming in Ada 2005. Cambridge University Press, Cambridge (2007)
10. Fencott, C.: Formal Methods for Concurrency. International Thomson Computer Press, Boston (1995)
11. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
12. van Harmelen, F., Lifschitz, V., Porter, B. (eds.): Handbook of Knowledge Representation. Elsevier Science, Amsterdam (2007)
13. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River (1985)
14. ISO: Information processing systems, open systems interconnection LOTOS. Tech. Rep. ISO 8807 (1989)
15. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, vol. 1-3. Springer, Berlin (1992-1997)
16. Matyasik, P.: Design and analysis of embedded systems with XCCS process algebra. Ph.D. thesis, AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Kraków, Poland (2009)
17. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
18. Murata, T.: Petri nets: Properties, analysis and applications. Proc. of the IEEE 77(4), 541–580 (1989)
19. Nalepa, G.J.: Languages and tools for rule modeling. In: Giurca, A., Dragan Gasevic, K.T. (eds.) Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, pp. 596–624. IGI Global, Hershey (2009)
20. Nalepa, G.J., Ligęza, A.: The hekate methodology. hybrid engineering of intelligent systems. Applied Mathematics and Computer Science 20(1), 35–53 (2010)
21. Object Management Group: OMG Systems Modeling Language (OMG SysML) (2008)
22. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. O'Reilly Media, Sebastopol (2008)

23. Samolej, S., Rak, T.: Simulation and performance analysis of distributed internet systems using tcpns. Informatica 33(4), 405–415 (2009)
24. Szpyrka, M.: Analysis of RTCP-nets with reachability graphs. Fundamenta Informaticae 74(2-3), 375–390 (2006)
25. Szpyrka, M.: Design and analysis of rule-based systems with adder designer. In: Cotta, C., Reich, S., Schaefer, R., Ligęza, A. (eds.) Knowledge-Driven Computing: Knowledge Engineering and Intelligent Computations. Studies in Computational Intelligence, vol. 102, pp. 255–271. Springer, Heidelberg (2008)
26. Szpyrka, M.: Petri nets for modelling and analysis of concurrent systems. WNT, Warsaw (2008) (in Polish)