Marcin Szpyrka, Piotr Matyasik,
Michał Wypych, Jerzy Biernacki
Łukasz Podolski

Alvis modelling language

Manual, version 0.13 (draft)

January 11, 2017

Contents

1 Introduction 1
1.1 White papers 2
1.2 Manual contents 3

2 Communication diagrams 5
2.1 Nonhierarchical Communication Diagrams 5
2.2 Hierarchical Communication Diagrams 8
2.3 Hierarchical communication diagrams in practice 10
2.3.1 Modules 12
2.3.2 Removing multiple connections 13
2.3.3 Replacing multiple agent's instances with a single representation 14
2.3.4 Grouping repeating fragments of a model 15
2.3.5 Components reusability 16

3 Code layer 19
3.1 Code structure 19
3.2 Types and parameters 19
3.3 Communication statements 21
3.4 Loop statements and recursion 23
3.5 Alternatives 25
3.6 Procedures 26
3.7 Null statement 27
3.8 Delay statement 27
3.9 Start statement 27

4 Basic Haskell model 29
4.1 Haskell files used by Alvis Compiler 29
4.2 Representation of model structure in Haskell 30
4.3 Representation of agents' states 32
4.4 Transitions 36

4.5 Functions *enable* and *fire* 40

5 LTS graphs for non-time models 53
5.1 LTS graphs 53
5.2 Generation of LTS graphs using Haskell model representation 55

6 LTS graphs for time models 59
6.1 Time in Alvis models 59

References 63

1

Introduction

Alvis is a formal modelling language being developed at AGH University of Science and Technology in Kraków, Department of Applied Computer Science. The main aim of the project is to provide a flexible modelling language for concurrent and real-time systems with possibilities of formal models verification. Alvis combines advantages of high level programming languages with a graphical language for modelling interconnections between subsystems (called agents) of a concurrent system.

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 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.

The key concept of Alvis is *agent*. The name has been taken from the CCS process algebra [1] 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 instead of algebraic equations. Moreover, it combines hierarchical graphical modelling with high level programming language statements.

An Alvis model is a system of agents that usually run concurrently, communicate one with another, compete for shared resources etc. Agents are divided into three groups: active agents can be treated as processing nodes, passive agents represent shared resources and hierarchical agents represent submodels.

An Alvis model is composed of two 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 called communication diagram 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 Alvis statements and the Haskell functional programming language [2].

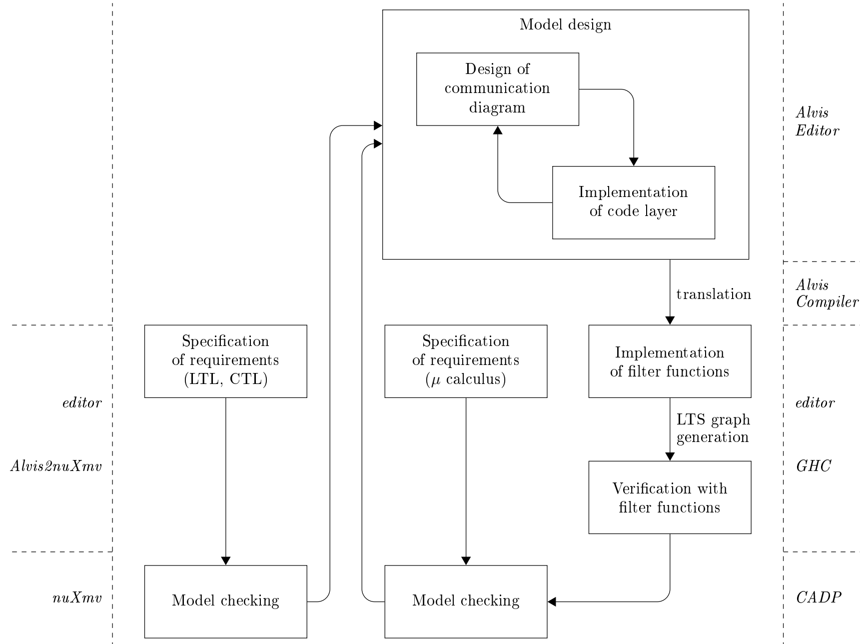

Fig. 1.1. Modelling and verification process with Alvis

The scheme of the modelling and verification process with Alvis is shown in Fig. 1.1. From a user's perspective, it starts from designing a model using prototype modelling environment called *Alvis Editor*. The designed model is stored using XML file format. Then *Alvis Compiler* is used to translate it into Haskell source code and its Haskell representation is used to generate the LTS graph (*labelled transition system*). We use Haskell as a middle-stage representation of an Alvis model in similar way as CPN Tools uses SML to generate reachability graphs for coloured Petri nets [3]. The main difference between these approaches is that Alvis users have access to the generated Haskell source files and may include some extra Haskell code into them.

The manual is addressed to people who want to use Alvis for systems modelling, both to beginner and advanced users. Beginner users will find here information about the syntax and semantics of the language and description of the software supporting the use of Alvis. Advanced users will find here detail information about the Haskell representation that will allow them to adapt to their needs the model or to develop their own verification methods.

1.1 White papers

This section provides information about the most important white papers on the Alvis language.

[4] Szpyrka M., Matyasik P., Biernacki J., Biernacka A., Wypych M., Kotulski, L.: *Hierarchical communication diagrams*. Computing and Informatics, vol. 35, no. 1, 2016, pp. 55–83
The paper contains formal description of communication diagrams. Both non-hierarchical and hierarchical communication diagram have been defined in the paper. Moreover, it contains detail information about transformation of hierarchical diagrams to equivalent forms.
[5] Szpyrka M., Matyasik P., Mrówka R., Kotulski L.: *Formal description of Alvis language with* α^0 *system layer*. Fundamenta Informaticae, vol. 129, no. 1–2, 2014, pp. 161–176
The paper contains formal description of non-time multi-processor version of the Alvis language. The presented description is slightly different version than the currently implemented one, but the paper provides a formal description of agent and model states, the transitions idea etc.
[6] Szpyrka M., Matyasik P., Mrówka R.: *Alvis – modelling language for concurrent systems*. Studies in Computational Intelligence, vol. 362, Springer-Verlag, 2011, pp. 315–341
The chapter contains the first complete description of the Alvis language syntax. The currently implemented version of Alvis is slightly different than the version presented in this chapter.

1.2 Manual contents

- Chapter 2 presents communication diagrams – the visual part of the Alvis language.
- Chapter 3 concerns the code layer. It contains description of all statements currently supported by the *Alvis Compiler*.
- Chapter 4 provides detail description of the basic Haskell representation of Alvis models. The representation is common for all system layers and is the basis for implementation of algorithms for computing LTS graphs.

2

Communication diagrams

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 [7], [8]. 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 [4] 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, Haskell keywords and identifiers used in the Haskell model representation (see Chapter 4) cannot be used as identifiers. Names of agents that are initially activated (represent running processes) are underlined. Hierarchical agents are indicated by black triangles. A survey of Alvis graphical items is given 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 next to the port. It must fulfill the same requirements as agents' identifiers but its first character must be a lower-case letter.

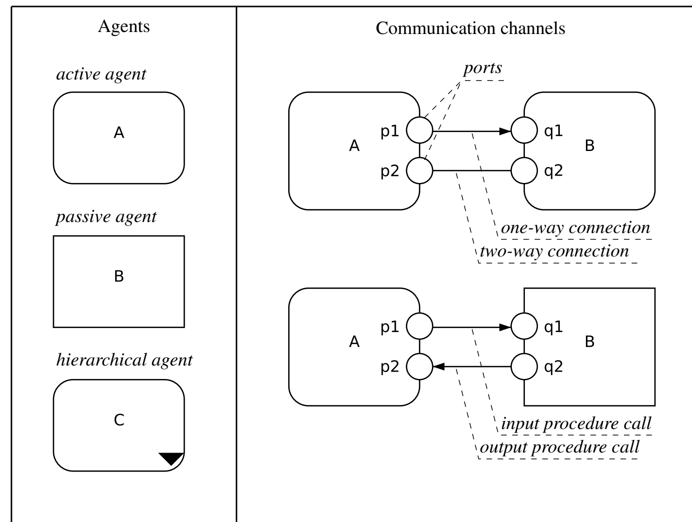

Fig. 2.1. Graphical elements of Alvis modelling language

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.

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. A 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, a 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*. An input port can be used as argument of the *in* statement. Similarly, an output port can be used as argument of the *out* statement (see Chapter 3).

Let us consider the communication diagram shown in Fig. 2.2. Port $B.b$ is 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 $C.c$ is both 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 $C.c$ 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. 2.2 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 $A.a$, the value can be collected (if suitable statements are used)

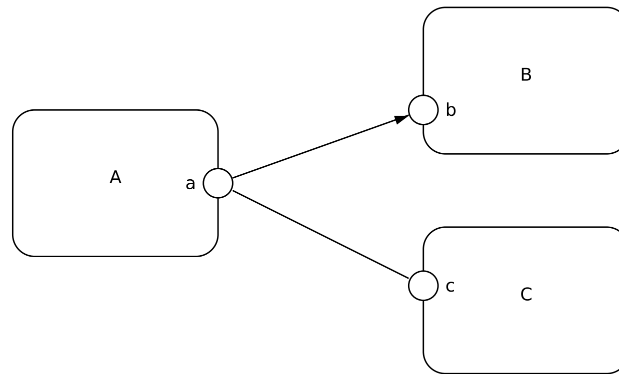

Fig. 2.2. Example of connections among active agents

by agent B or C. If agent A initialises a communication demanding a signal/value on port $A.a$, such a signal/value can be provided only by agent B.

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

1. A connection between an active agent port and a passive agent procedure port (a port with a *proc* statement) – 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 or output ones.

To summarize, a non-hierarchical communication diagram must fulfil the following restrictions:

1. A connection cannot be defined between ports of the same agent.
2. Procedural ports are either input or output ones.
3. A connection between an active and a passive agent must be a procedure call.
4. Any connection with a passive agent must be a one-way connection.
5. A connection between two passive agents must be a procedure call from a non-procedure port.

An example of a communication diagram for a dining philosophers problem is shown in Fig. 2.3. 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

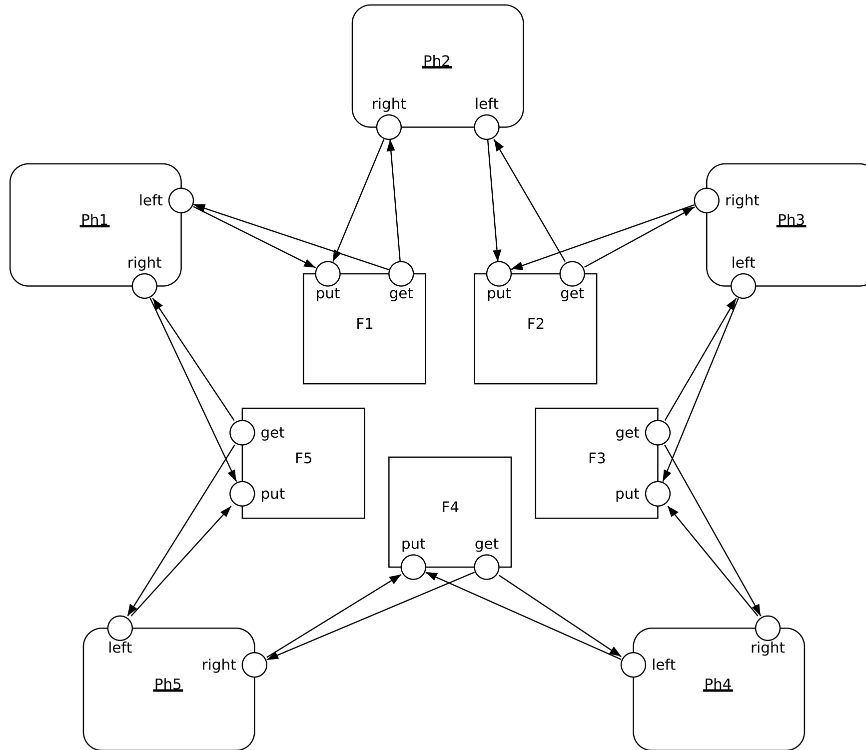

Fig. 2.3. Communication diagram of the dining philosophers problem

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 two ports used to get and put right and left forks respectively. Forks are represented by passive agents with two 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.

A communication diagram can be treated as a module and represented by a single hierarchical agent at the higher level. Hierarchical agents are not defined in the code layer. Non-hierarchical part of a communication diagram is called *page*. A non-hierarchical communication diagram is composed of a single page. Hierarchical communication diagrams are composed of a set of pages. The main difference between a non-hierarchical communication diagram and a page is the possibility of using hierarchical agents in case of pages.

A page D^i must satisfy the following restrictions:

1. A connection cannot be defined between ports of the same agent.
2. Procedural ports are either input or output ones.
3. A connection between an active and a passive agent must be a procedure call.
4. Any connection with a passive agent must be a one-way connection.
5. A connection between two passive agents must be a procedure call from a non-procedure port.
6. Hierarchical agents cannot have procedure ports.
7. A connection between a hierarchical and a passive agent must be a one-way connection.

The above description treats hierarchical agents almost like active ones. However, connections with ports of hierarchical agents can make some substitutions illegal, i.e. after the transformation of a hierarchical diagram into the equivalent flat one, all connections must satisfy the conditions defined for non-hierarchical diagrams.

The substitution mechanism is based on assigning ports of the consider hierarchical agent with the so-called *join ports* on the corresponding page (subpage). The set of join ports contains all ports from the subpage with names the same as those of the hierarchical agent. It should be underlined that the join ports may be distributed among a few agents, but they cannot be connected with any ports.

If each port of a hierarchical agent has assigned one join port the substitution is called a *simple* one. If one port of a hierarchical agent may have assigned more than one join port on the subpage the substitution is called an *extended* one.

The idea of simple substitution is illustrated by Fig. 2.4, while the idea of extended substitution is illustrated by Fig. 2.5.

Formally, a hierarchical communication diagram is defined as a pair $H = (\mathcal{D}, \gamma)$, where D is set of pages with pairwise disjoint sets of agents and γ is the *substitution function* that maps each hierarchical agent from those pages to its direct subpage. The function must fulfil the following requirements:

1. γ is an injection.
2. For any hierarchical agent X, agent X ang page $\gamma(X)$ satisfy the requirements of the simple or extended substitution.
3. The directed graph $\mathcal{G}$ with the set of nodes D and the set of edges E, where $(D^i, X^i, D^j) \in E$ iff $\gamma(X^i) = D^j$ is a tree or a forest.

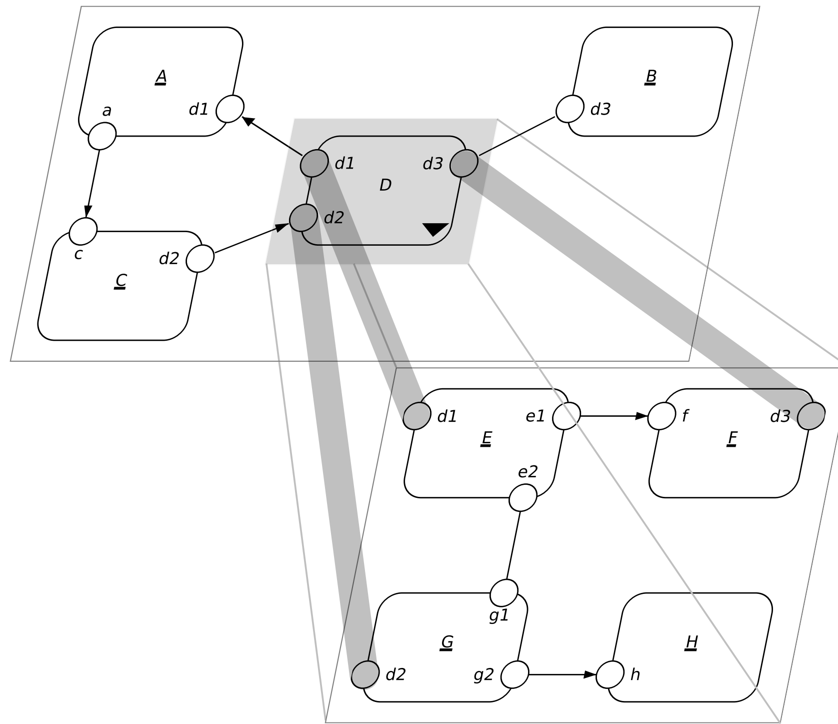

Fig. 2.4. Simple substitution

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

Formally pages that are not assigned to any hierarchical agent are called *primary pages*, They are roots of trees that constitute the page hierarchy graph.

User can manage the structure of a hierarchical communication diagram replacing a hierarchical agent with its subpage (*analysis operation*) or moving a part of a page to a subpage and introducing a hierarchical agent (*synthesis operation*). The results of analysis operations for models from Fig. 2.4 and Fig. 2.5 are shown on Fig. 2.6 and Fig. 2.7 respectively.

2.3 Hierarchical communication diagrams in practice

One of the main motivations behind formulating the Alvis language was to make the formal verification more intelligible and easy to use for the average engineer. The graphical layer of the Alvis model was expected to be both easy to model and to understand. Even complex systems can be modelled using relatively simple com-

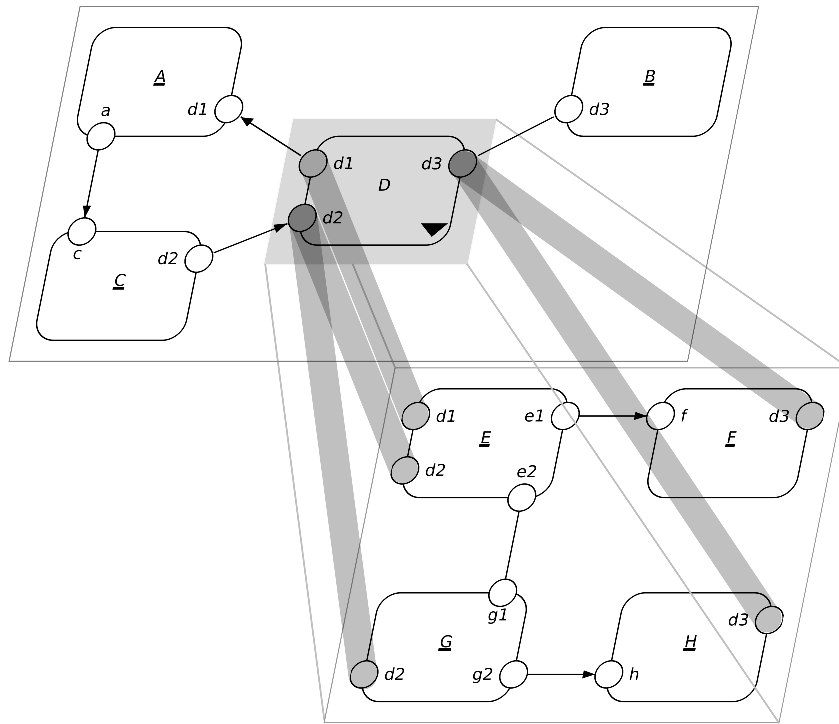

Fig. 2.5. Extended substitution

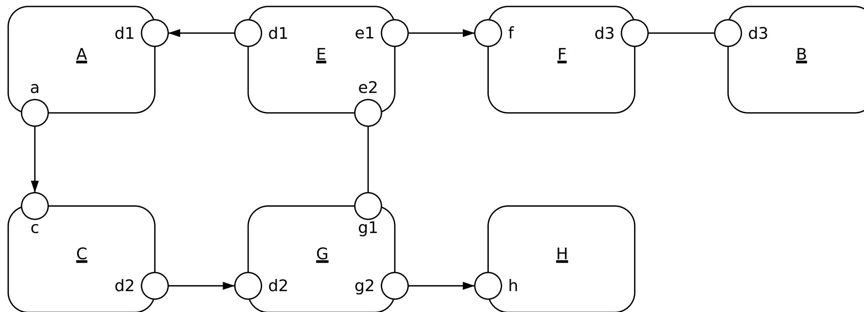

Fig. 2.6. Result of analysis operation for model from Fig. 2.4

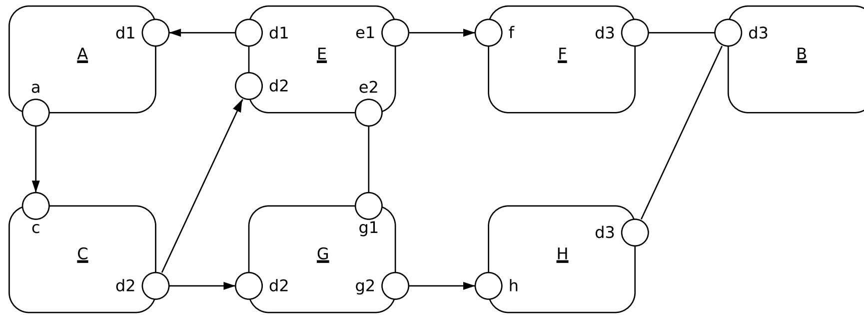

Fig. 2.7. Result of analysis operation for model from Fig. 2.5

munication diagrams, and, among other factors, this is possible due to the usage of hierarchy. Synthesis and analysis operations are actually crucial for making communication diagrams intuitive. This section is introducing standard situations in which hierarchy usage is greatly improving the readability of the model. This section is an extended version of the rules presented in [4]

2.3.1 Modules

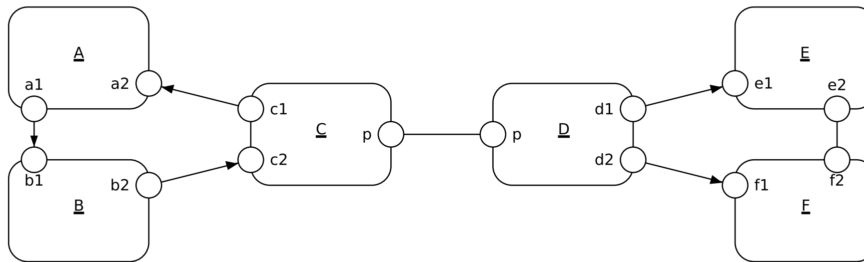

Fig. 2.8. Original communication diagram

Splitting the system into smaller parts is one of the most basic concepts in software engineering. Its fundamental purpose it to avoid situations in which having everything in one module, class or file one have to worry about everything at once when expanding the existing solution. Although this practice may work for small systems, for big ones it quickly becomes next to impossible. To solve this problem, fragments of functionality are split into their own modules which encapsulate the separated pieces of logic. Then, when working on a particular module, one does not have to directly consider the implications of the work on other parts of the system. This is invaluable for working efficiently. There are many other benefits to break-

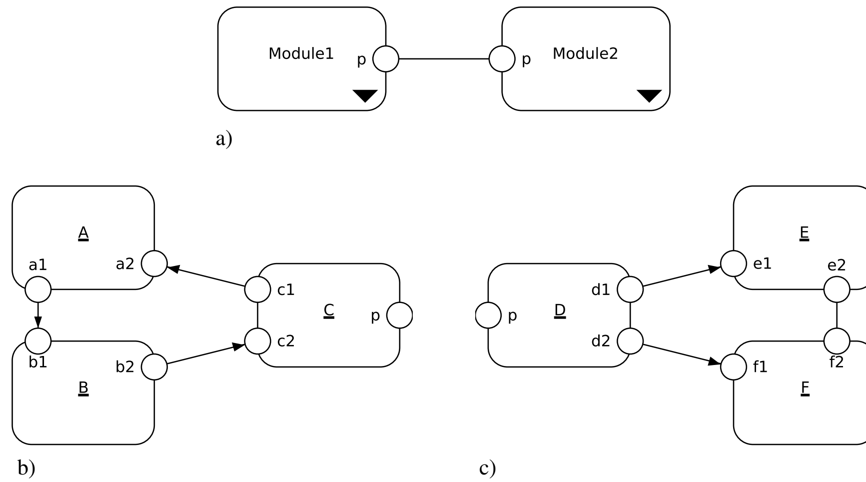

Fig. 2.9. a) Hierarchical agents for two modules; b) subpage for agent Module1; c) subpage for agent Module2

ing system into modules, e.g. the model is more maintainable, testable and reusable. Moreover, such a module can be used as a reusable component.

Breaking the system into modules is the most common and general application of hierarchy. Modelling in Alvis language incorporates this practice. Figure 2.8 presents an example of a communication diagram. Supposing its logic can be divided and encapsulated into two separate modules, the synthesis operation may be used. Its result is presented in Fig. 2.9. There is a hierarchical diagram containing two hierarchical agents on the primary page (a) and two subpages with agents belonging to the corresponding modules (b and c).

2.3.2 Removing multiple connections

Creating even a moderately complex model can lead to situations in which the communication diagram becomes difficult to read due to the increasing amount of connections between the agents. The warning signal is definitely the moment in which some of the connections are intersecting each other. A simple model with 4 agents and 7 connections between their ports is shown in Fig. 2.10. Although this diagram is readable, it is mostly so because of its small size. It is not difficult to notice that agent `D` is connected to every other agent in the diagram. Adding more agents connected to it will result in progressing obscuration of the model. However, a smart use of a synthesis operation can reduce the amount of visible connections. Fig. 2.11 presents the same model with all the agents communicating with agent `D` grouped and brought to a subpage. The total amount of connections between the agents is reduced to 5 and the legibility of the model is definitely increased.

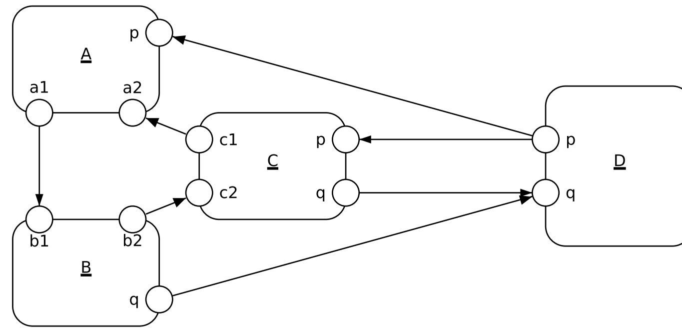

Fig. 2.10. Original communication diagram

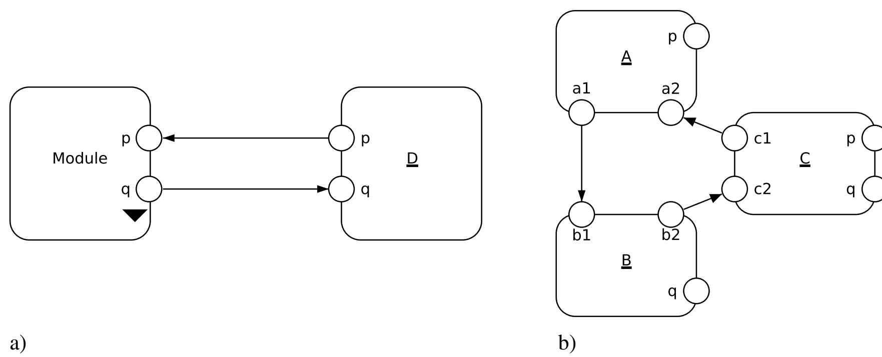

Fig. 2.11. a) Inserted hierarchical agent Module; b) subpage for agent Module

2.3.3 Replacing multiple agent's instances with a single representation

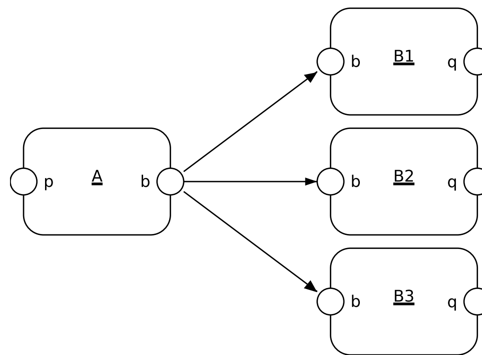

Fig. 2.12. Original communication diagram

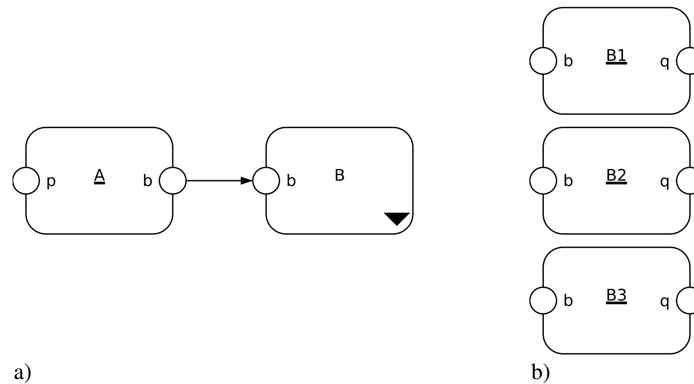

Fig. 2.13. a) Inserted hierarchical agent B; b) subpage for agent B

Another condition for using hierarchy is when the model contains multiple instances of the same agent. This is actually a very common scenario in real-time systems. Elements such as sensors, indicators, displays and other subdevices are often repeated in the scope of a single system. Each of them can be on some level of abstraction represented by a separate agent. Therefore the communication diagram can quickly become crowded with the great amount of reoccurring instances of specific agents. Utilising hierarchy in such case enables the possibility of replacing all these instances with a single hierarchical agent. An example is shown in Fig. 2.12. The original diagram contains 3 instances of the same agent ($B1 – B3$). These instances can be easily replaced by a single hierarchical agent B and placed on one subpage (Fig. 2.13). This operation has one more great advantage. When new instances are added, it is enough to add them to the defined subpage. One does not have to worry about drawing the connections with other agents. This also applies to the removal of unwanted instances.

2.3.4 Grouping repeating fragments of a model

In extensive systems, repeating fragments of the model can often occur. Each of these excerpts represents similar functionality but is placed in different part of the system and is connected to different agents. Such a model is therefore potentially easy to disrupt. Assuming that these fragments need some kind of a change, one would have to find all of these fragments and update them one by one. There is a great chance that the engineer updating the model will make a mistake and the model will not be consistent. And it would be very hard to determine where the mistake has been made. The solution to this problem is in hierarchy once again. Fig. 2.14 contains a simple example of a model with repeating fragments. In the example these fragments are placed in the same place and are easy to identify. In model of a real system it may not be so. Therefore, placing them on a single subpage may be very useful. In the Fig. 2.15 these excerpts are represented by a single hierarchical agent and moved to

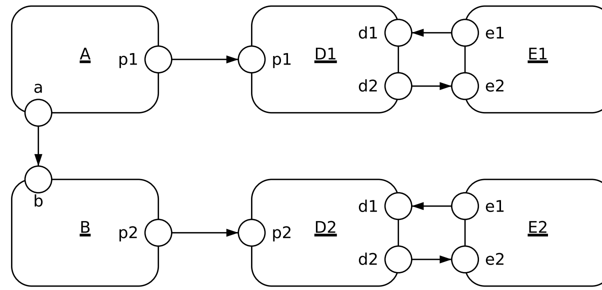

Fig. 2.14. Original communication diagram

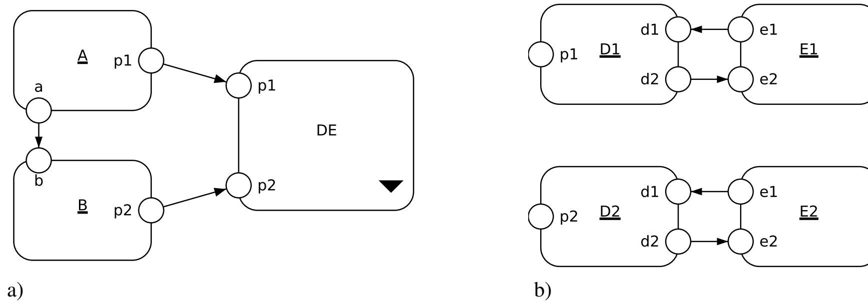

Fig. 2.15. a) Inserted hierarchical agents DE; b) subpage for agent DE

the lower level. This way, any changes to be made are less prone to cause mistakes because all repeated fragments are gathered in one place.

2.3.5 Components reusability

Grouping some functionalities into reusable components with defined interface is another common practice used in software engineering. Systems often consist of smaller subsystems. Each one of these subsystems is an independent entity and can be modelled and verified all by itself. Having such a subsystem modelled one may want to use it in many different systems. With hierarchy this is an easy task. It is enough to add a defined component as a subpage represented by a single hierarchical agent. Ports of this new hierarchical agent are the interface of the component and can be connected to the proper ports of the surrounding system. An example of a model of reusable component is presented in Fig. 2.16. Fig. 2.17 shows two systems in which this component is incorporated.

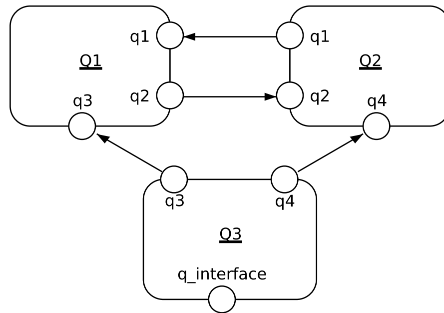

Fig. 2.16. Reusable component Q communication diagram

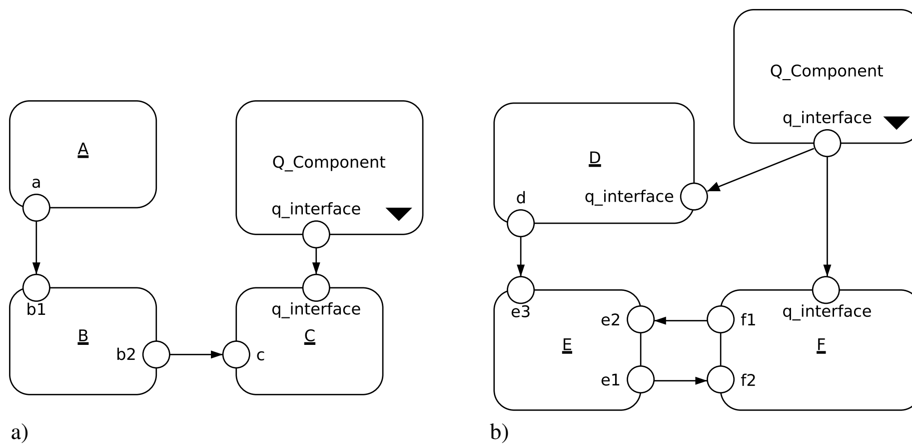

Fig. 2.17. a) Component Q used in model ABC; b) Component Q used in model DEF

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 statements and some elements of the Haskell functional programming language [2].

3.1 Code structure

The code layer is a sequence of agents' blocks. The structure of an agent block is shown in Listing 3.1.

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

Listing 3.1. Structure of an agent block

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.

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 3.2). 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 3.2. Examples of Haskell composite data types

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 3.3.

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

Listing 3.3. 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 3.4 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 3.4).

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

Listing 3.4. 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 [9].

Communication modes in Alvis can be grouped according to various criteria. First of all, we can distinguish communication between two active agents and communication with a passive agent (the second side of such a communication may be both active or passive agent). When two active agents communicate with each other, they synchronise their execution. It is required for both agents to reach an appropriate state to be able to execute the communication protocol. In the case of a communication with a passive agent, the communication is treated as a procedure call. The passive agent performs a service for the second agent using its context.

On the other hand, any communication may be a *pure communication* or a *value passing communication*. In case of a pure communication, a signal (without specified value) is sent between agents, while in case of a value passing communication, a value is sent. The value can be a composed type. The only difference between these communication modes is the new state of the agent that collects the value/signal. In the case of a value passing communication, value of one agent's parameter is updated.

Finally, any communication may be a *blocking* or *non-blocking* one. In the case of blocking communication, the agent that initiates the communication waits until another agent finalises it. On the other hand, when a non-blocking communication is used, the agent that initiates the communication may abandon it when the second side is not ready to finalise it.

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. The *in* statement assigns the collected value to its parameter, while the *out* statement sends the value of its parameter. The syntax for the *in*/*out* statements is given in Listing 3.5.

```
in p;                    -- 1

in p x;                  -- 2

in (t) p;                -- 3
```

```
in (t) p x;                  -- 4

in (t) p {                   -- 5
  success { ... }
  fail    { ... }
}

in (t) p x {                 -- 6
  success { ... }
  fail    { ... }
}

out p;                       -- 7

out p x;                     -- 8

out (t) p;                   -- 9

out (t) p x;                 -- 10

out (t) p {                  -- 11
  success { ... }
  fail    { ... }
}

out (t) p x {                -- 12
  success { ... }
  fail    { ... }
}
```

Listing 3.5. Syntax of the *in* and out *out* statements

Listing 3.5 provides syntax for the following versions of the communication statements:

1. Blocking pure *in* statement;
2. Blocking value passing *in* statement;
3. Non-blocking pure *in* statement;
4. Non-blocking value passing *in* statement;
5. Non-blocking pure *in* statement with *success* and *fail* clauses – the clauses are optional and only one of them may be used;
6. Non-blocking value passing *in* statement with *success* and *fail* clauses – the clauses are optional and only one of them may be used;
7. Blocking pure *out* statement;
8. Blocking value passing *out* statement;
9. Non-blocking pure *out* statement;
10. Non-blocking value passing *out* statement;

11. Non-blocking pure *out* statement with *success* and *fail* clauses – the clauses are optional and only one of them may be used;
12. Non-blocking value passing *out* statement with *success* and *fail* clauses – the clauses are optional and only one of them may be used;

A communication between two active agents can be initialised by any of them. In case of blocking communication, 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. In case of a non-blocking communication, the communication may be abandoned if the second agent is not ready to finalise it. The *in* and *out* statements for non-blocking communication contain additional parameter that represents how long the corresponding agent may wait for finalisation of the communication. In case of non-timed Alvis models, the time parameter is always treated as 0. Thus, for a non-timed model, a (successful) non-blocking communication may be used to finalise the communication only.

The *in* and *out* non-blocking statements may be equipped with *success* and *fail* clauses (both are optional). The *success* clause is executed upon successful communication, while the *fail* clause is executed, if the communication has been abandoned.

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. Moreover, a passive agent may also contain internal non-procedure ports. Such ports are connected with another passive agents and are used to call other procedures inside procedures of the considered agent.

In case of an input procedure, an 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. A procedure is always executed using an active agent context. If a procedure is called by an active agent, the passive agent uses the active agent context. If a procedure is called by a passive agent, the passive agent uses the same context as the agent that called the procedure. For more details see Sections 3.6.

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 3.6. It repeats its contents infinitely.

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

Listing 3.6. General *loop* statement

The second loop repeats its contents while the guard (g) is satisfied (see Listing 3.7). 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 3.7. While *loop* statement

The last loop statement is the so-called *loop every* statement as shown in Listing 3.8. The loop repeats its contents every t time-units. For non-time models the loop is treated as the general loop. The contents of a loop every statement must end with the *null* statement.

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

Listing 3.8. The *loop every* statement

Let us consider the implementation of agent D shown in Listing 3.9. The agent collects 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;
  }
}
```

Listing 3.9. Agent D implementation

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. For example, the behaviour of the agent D from Listing 3.9 can be also defined as shown in Listing 3.10.

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

Listing 3.10. Definition of agent D infinite behaviour with the *jump* statement

3.5 Alternatives

In order to allow for the description of agents whose behaviour may follow different alternative paths, Alvis offers the *select* statement (see Listing 3.11). 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 omits them all and executes the statements below the *select* statement.

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

Listing 3.11. Syntax of the *select* statement

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;
- *non-procedure 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 the *exit* statement has been performed. The *exit* statement must be used only after the *in/out* statement that corresponds to the procedure call.

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

Listing 3.12. Syntax of the *proc* statement

The general syntax of the *proc* statement is shown in Listing 3.12 – 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.

Let us consider the model of dining philosophers (see Section 2.1 and Fig. 2.3). The code layer for the model is shown in Listing 3.13. 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.

```
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 3.13. Dinign philosophers model – code layer

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.

3.7 Null statement

Alvis provides a null statement that does not represent any particular operation. The statement must be placed as the last statement inside the *loop every* statement (see Section 3.4).

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

3.8 Delay statement

The *delay* postpones an agent for a given time. The syntax of the statement is given in Listing 3.14 (t stays for an integer).

```
delay (t);
```

Listing 3.14. Syntax of the *delay* statement

3.9 Start statement

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. The syntax of the statement is given in Listing 3.15.

```
start Agent_name;
```

Listing 3.15. Syntax of the *start* statement

4

Basic Haskell model

Alvis is a formal modelling language. The verification process is based on LTS graphs (Labelled Transition Systems) and outer verification tools like CADP [10] and nuXmv [11]. To generate an LTS graph for a given Alvis model the Haskell [2] middle-stage representation is used (see Fig. 1.1, page 2). This chapter deals with the basic Haskell representation that is common for all Alvis models regardless of chosen system layer.

4.1 Haskell files used by Alvis Compiler

The *Alvis Compiler* installation package contains a subdirectory called *compiler-files*. This subdirectory contains a set of Haskell source files used by the compiler to complete the Haskell source code for a given model. The collection of files used during a model compilation depends on the used compiler options. The list of all Haskell files is given in Listing 4.1.

```
aldebaran.hs
alvis.hs
dot.hs
lts.hs
priority_default.hs
priority_p1.hs
priority_p2.hs
program.hs
program_ald.hs
time_aldebaran.hs
time_dot.hs
time_lts.hs
```

Listing 4.1. Haskell files attached to compiler

Some compiler options accept files defined by the user e.g. algorithms to manage priorities. Moreover, advanced users may redefine some parts of the Haskell files provided by the compiler.

In addition to the functions listed in Fig. 4.1 the Haskell representation contains source code generated individually for each model, e.g. enumerated data types for agents and ports, data types used to represent states of individual agents and the whole model etc.

4.2 Representation of model structure in Haskell

The simple Alvis model presented in Fig. 4.1 will be used to illustrate pieces of Haskell code generated for Alvis. To simplify referring to the code layer, the comments contain the numbers of Alvis statements.

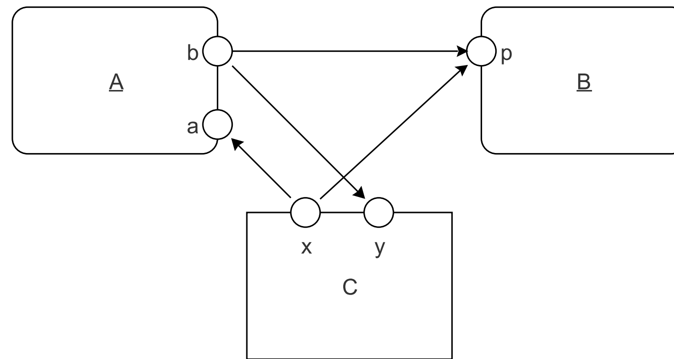

Fig. 4.1. Example of Alvis model (communication diagram)

```
agent A {
  loop {                                  -- 1
    in  (0) a {                           -- 2
      success { jump off; } }             -- 3
    out (0) b {                           -- 4
      success { jump off; } }             -- 5
  }
  off:
  null;                                   -- 6
}

agent B {
  loop {                                  -- 1
    in p;                                 -- 2
  }
```

```
}

agent C {
  m :: Bool = False;
  proc (m == True) x {
    out x;                                    -- 1
    m = False;                                -- 2
    exit; }                                   -- 3
  proc (m == False) y {
    in y;                                     -- 4
    m = True;                                 -- 5
    exit; }                                   -- 6
}
```

Listing 4.2. Example of Alvis model (code layer)

The Haskell representation of an Alvis model behaviour is based on the so-called *enable-fire* approach which takes inspiration from Petri nets. The *enable* function takes a state and an agent name and provides the list of steps (transitions) the agent may execute in the state. The *fire* function for a state and a transition enabled in the state provides a list (usually with one element) of new states. The main goal of the Haskell middle-stage representation is to provide these two functions. The source code contains only those informations (data types and function) that are necessary to use the *enable* and *fire* functions to generate the LTS graph.

```
data Agent
  = A
  | B
  | C
  deriving (Eq,Ord,Show)

agents :: [Agent]
agents   = [A, B, C]

modelSize :: Int
modelSize  = 3
```

Listing 4.3. *Agent* data type for model from Fig. 4.1

The Haskell code generated individually for the given model starts with data type *Agent*. For the model from Fig. 4.1 it is defined as shown in Listing 4.3. Moreover, the list *agents* provides the list of all model agents and *modelSize* provides the size of the list. The Haskell representation provides also the function *agentNumber*:

```
agentNumber :: Agent -> Int
```

that returns the position of the agent in the *agents* list (starting from 1).

The Alvis compiler works with flat models, thus hierarchical agents are not taken into consideration.

```
data Port
  = A_a
  | A_b
  | B_p
  | C_x
  | C_y
  | None
  deriving (Eq,Ord)
```

Listing 4.4. *Port* data type for model from Fig. 4.1

The second data type that concerns the model structure directly is *Port*. Due to the Haskell syntax the underscore sign is used instead of dot to separate the agent name from the port name. The dots are restored while conversion to *String*. The *None* constructor represents the *empty* port. This special value is used by some functions to point out the lack of port to return.

The Haskell representation provides also the function *agentName*:

```
agentName :: Port -> Agent
```

that returns the agent name for the given port.

Communication channels are not represented directly in the model. They are hidden in the *fire* function (see Section 4.5).

4.3 Representation of agents' states

A *state of an agent* X[1] is a tuple $S(X) = (am(X), pc(X), ci(X), pv(X))$, where $am(X)$, $pc(X)$, $ci(X)$ and $pv(X)$ denote *agent mode*, *program counter*, *context information list* and *parameters values* of the agent X respectively. A state of an Alvis model is represented as a sequence of agents' states as shown in Fig. 4.2.

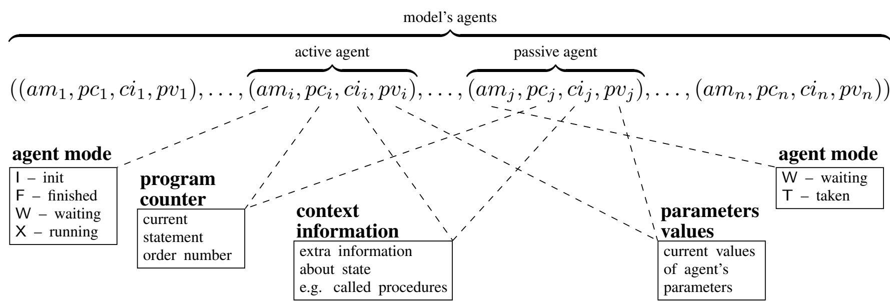

Fig. 4.2. Representation of an Alvis model state

For active agents the agent's mode takes one of the following values:

[1] X is not valid agent name. The letter is used only for theoretical considerations.

- *init* (I) – the default mode for agents that are inactive in the initial state;
- *running* (X) – an agent is performing one of its statements;
- *waiting* (W) – an agent is waiting for an event, e.g. for a communication with another active agent, or for a currently inaccessible procedure of a passive agent;
- *finished* (F) – an agent has finished its work.
- *ready* (R) – an agent is ready to perform one of its steps (statements), but it has no access to the processor (This mode is used for single-processor environment only).

For passive agents the mode takes one of the two values:

- *waiting* (W) – an agent is inactive and waits for another agent to call one of its accessible procedures;
- *taken* (T) – one of the passive agent's procedures has been called and the agent is executing it.

The Haskell model provides an enumerated data type *Mode* to denote the possible modes (see Listing 4.5).[2]

```
data Mode
  = F      -- ^ finished
  | I      -- ^ init
  | R      -- ^ ready
  | T      -- ^ taken
  | W      -- ^ waiting
  | X      -- ^ running (executing)
  deriving (Eq,Show,Ord)
```

Listing 4.5. *Mode* data type definition

The program counter points at the next statement to be executed or the statement that has been already executed but to be completed, it needs a feedback from another agent (e.g. a communication between agents). For active agents in the *init* or *finished* mode and for passive agents in the *waiting* mode, the program counter is equal to 0.

We say that:

- $pc(X)$ *points out* an *delay* (*exec, exit, jump, null, start*) statement iff the next statement to be executed is an *delay* (*exec, exit, jump, null, start*) statement;
- $pc(X)$ *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;
- $pc(X)$ *points out* a *select* statement iff the next statement to be executed is entering the *select* statement and possibly one of its branches;
- $pc(X)$ *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).

[2] The letters used to denote agents' modes cannot be used as names for agents.

The statements are numbered according to their position in the source code. However, only statements that are represented by transitions (see Section 4.4) are assigned ordinal numbers. For example, key words like *alt, fail* or *success* represents only clauses of other statements and are not treated as independent ones.

Possible values of the program counter for agents from Listing 4.2 are given in the comments. Values of program counters are modified by the *fire* function (see Section 4.5).

The context information list contains additional information about the current state of an agent e.g. if an agent is in the *waiting* mode, *ci* contains information about events the agent is waiting for. Possible values put into *ci* lists are given in Listing 4.6.

```
data ContextInfo
  = CProc   Port
  | CIn     Port
  | COut    Port
  | CTimer Int Int
  | CTimeout Int
  | CSFT    Int
  | CNSFT   Int
  | CLock
  | CNone
  | COTimer Int Int
  deriving (Eq,Ord)
```

Listing 4.6. *ContextInfo* data type definition

The order of placement of constructor functions is important and cannot be modified by a programmer. The entries have the following meanings:

- `CProc Port` – The entry represents the name of called procedure.
- `CIn Port` – For passive agents (in the *waiting* mode), the entry denotes the name of accessible input procedure. For active agents (in the *waiting* mode), the entry denotes the name of port used in the last *in* statement i.e. the entry indicates where the other agent should provide a signal (value) to finish the communication.
- `COut Port` – For passive agents (in the *waiting* mode), the entry denotes the name of accessible output procedure. For active agents (in the *waiting* mode), the entry denotes the name of port used in the last *out* statement i.e. the entry indicates from where the other agent should collect a signal (value) to finish the communication.
- `CTimer Int Int` – The entry denotes the number of timer time-units (second parameter) to go by for the given statement (first parameter).
- `CTimeout Int` – The entry represents a timeout signal generated for the given statement (the parameter points out the number of the statement).
- `CSFT Int` – The entry denotes the number of timer time-units necessary to finish current statement.

- `CNSFT Int` – The entry denotes the number of timer time-units necessary to finish current statement. This entry is used as a temporal one, while computation the result for a multi-step. It is never used in states presented in an LTS graph.
- `CLock` – The entry denotes that the agent is blocked by an ongoing communication transition.
- `CNone` – This is an *empty entry* used by some searching functions to point out the lack of result.
- `COTimer Int Int` – The entry denotes the number of timer time-units (second parameter) to go by for the given statement (first parameter). This entry is used as a temporal one, while computation the result for a multi-step. It is never used in states presented in an LTS graph.

If an agent is in the *init* or *finished* mode, its context information list is empty.

For performance reasons the context information lists are implemented as sets of values of the *ContextInfo* data type. The data type is equipped with the *show* function presented in Listing 4.7.

```
instance Show ContextInfo where
  show (CProc a)       = "proc(" ++ (show a) ++ ")"
  show (CIn a)         = "in(" ++ (show a) ++ ")"
  show (COut a)        = "out(" ++ (show a) ++ ")"
  show (CTimer n d)    = "timer(" ++ (show n) ++ ","
                         ++ (show d) ++ ")"
  show (CTimeout n)    = "timeout(" ++ (show n) ++ ")"
  show (CSFT d)        = "sft(" ++ (show d) ++ ")"
  show CLock           = "lock"
  show _               = ""
```

Listing 4.7. The *show* function for the *ContextInfo* data type

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

The Alvis Compiler generates individual data types for each agent (compiler works with flat models). Data type generated for agents from Listing 4.2 are given in Listing 4.8.

```
type AState = (Mode, Int, Set ContextInfo, ())
type BState = (Mode, Int, Set ContextInfo, ())
type CState = (Mode, Int, Set ContextInfo, (Bool))
```

Listing 4.8. Data types used to represent states of agents from Listing 4.2

The whole model state is represented by the *State* data type. The data type definition for the model from Fig. 4.1 and Listing 4.2 is given in Listing 4.9.

```
type State = (AState, BState, CState)
```

Listing 4.9. Data type used to represent states of the model from Fig. 4.1 and Listing 4.2

The Haskell model representation provides a list of functions used to access elements of the model states (the first parameter is the number of agent):

```
takeam :: Int -> State -> Mode
takepc :: Int -> State -> Int
takeci :: Int -> State -> Set ContextInfo
```

The modes, program counters, context information lists and values of parameters are modified by the *fire* function (see Section 4.5). When calculating a new state for the number of transitions executed in parallel, it may be useful to modify a context information list "for a moment". The Haskell model representation provides two function to manage such changes:

```
updateContextInfo :: Int -> (Set ContextInfo -> Set ContextInfo)
                      -> State -> State
updateAllContextInfo :: (Set ContextInfo -> Set ContextInfo)
                         -> State -> State
```

The former function is used to modify the context information list for one agent pointed out by the first parameter. The latter function is used to apply the same changes to all agents. For example to add a context information to the context list of the n-th agent the following function may be used:

```
addContextInfo :: Int -> ContextInfo -> State -> State
addContextInfo n info = updateContextInfo n (insert info)
```

The Haskell model provides some auxiliary functions used to obtain some selected information about the *CProc* entry, which is very important while searching for enable transitions. The function *isProc* is used to check, whether the given entry is a *CProc* entry:

```
isProc :: ContextInfo -> Bool
```

For the given *CProc* entry, the *procAgent* function provides the name of agent, which port is the CProc entry argument

```
procAgent :: ContextInfo -> Agent
```

The *findProc* function searches the given context information list for a *CProc* entry related to the agent (first parameter) procedure. The function returns `CProc None`, if such an entry does not exist.

```
findProc :: Agent -> Set ContextInfo -> ContextInfo
```

The *procFree* checks whether the given context information list does not contain a *CProc* entry.

```
procFree :: Set ContextInfo -> Bool
```

4.4 Transitions

Execution of an agent statement can be treated as a *step* performed by the corresponding model. If we ignore the time of executing of statements (option `--no-time`) the

LTS graph represents changes of states that are results of performing single steps. Steps performed by a model are described using the *transition* idea. The set of all possible transitions for the considered Alvis models is represented by the *TTransition* data type shown in Listing 4.10.

```
data TTransition
  = TDelay      Agent Int
  | TExec       Agent Int
  | TExit       Agent Int
  | TIn         Port Int
  | TInAP       Port Port Int
  | TInPP       Port Port Int
  | TInF        Port Port Int
  | TJump       Agent Int
  | TLoop       Agent Int
  | TLoopEvery  Agent Int
  | TNull       Agent Int
  | TOut        Port Int
  | TOutAP      Port Port Int
  | TOutPP      Port Port Int
  | TOutF       Port Port Int
  | TSelect     Agent Int
  | TStart      Agent Int
  | STInAP      Port Port Int
  | STInPP      Port Port Int
  | STOutAP     Port Port Int
  | STOutPP     Port Port Int
  | STDelayEnd  Agent Int
  | STLoopEnd   Agent Int
  | STInEnd     Port Int
  | STOutEnd    Port Int
  | STTime      Int
  deriving (Eq,Ord)
```

Listing 4.10. *TTransition* data type definition

For all transitions except *STTime* the first parameter points out the agent (sometimes using its port) and the integer parameter denotes the number of the corresponding statement.

The transitions *TDelay, TExec, TExit, TJump, TNull* and *TStart* represent corresponding statements.

The *TLoop* transition represents evaluating the guard (if any) and possibly entering the *loop* statement. Similarly, the *TLoopEvery* transition represents entering the *loop every* statement.

The *TSelect* transition represents entering the *select* statement and possibly one of its clauses – if all guards are closed control is transferred outside the statement.

The transitions *TInAP, TInPP TInF* and *TIn* represent the *in* statement. The use of four transitions for single statement is associated with a variety of situations in which the statement can be used.

- *TInAP* – An active agent calls an output procedure.
- *TInPP* – A passive agent calls an output procedure.
- *TInF* – An active agent finishes a communication with another active agent.
- *TIn* – An active agent initialises a communication and will wait for finalisation.

The transitions *TInAP, TInPP TInF* contains two parameters of the *Port* type. The second parameter represents the port of the second agent, e.g. for the *TInAP* transition the parameters denote the port of active agent used in the *in* statement, the procedure port of the passive agent and the number of the *in* statement respectively.

Similarly, the transitions *TOutAP, TOutPP TOutF* and *TOut* represent the *out* statement.

- *TOutAP* – An active agent calls an input procedure.
- *TOutPP* – A passive agent calls an input procedure.
- *TOutF* – An active agent finishes a communication with another active agent.
- *TOut* – An active agent initialises a communication and will wait for finalisation.

Moreover, the Haskell model provides four system transitions *STInAP, STInPP, STOutAP, STOutPP*. They represent situations when after waiting for an inaccessible procedure the agent that already initialised a communication is *waked up* when the procedure is already accessible.

The statements *delay, loop every, non-blocking in* and *out* moves an agent into *waiting* mode for some time. The waiting time is measured by a timer (put into the corresponding context information list). After the waiting time expires a *time-out* entry is put into the context information list, which activates one of the system transitions *STDelayEnd, STLoopEnd, STInEnd* or *STOutEnd*.

The last transition *STTime* is a system transition used to represent the passage of time. It is used to represent a situation when none agent can execute a statement (e.g. all are in the *waiting* mode) but after some passage of time at least one of them will be able to execute its next step.

The TTransition data type contains the set of all transitions used both with time and non-time models. If the `--no-time` compiler option is used only a subset of transitions is used (all transitions referring to time are omitted).

The *TTransition* data type is equipped with the *show* function presented in Listing 4.11. This is very important function, because its results are used to label arcs in LTS graphs. To change the default labelling system user may redefine the function.

```
instance Show TTransition where
  show (TDelay a _)       = "delay(" ++ show(a) ++ ")"
  show (TExec a _)        = "exec(" ++ show(a) ++ ")"
  show (TExit a _)        = "exit(" ++ show(a) ++ ")"
  show (TIn a _)          = "in(" ++ show(a) ++ ")"
  show (TInAP a _ _)      = "in(" ++ show(a) ++ ")"
  show (TInPP a _ _)      = "in(" ++ show(a) ++ ")"
```

```
show (TInF a _ _)        = "in(" ++ show(a) ++ ")"
show (TJump a _)         = "jump(" ++ show(a) ++ ")"
show (TLoop a _)         = "loop(" ++ show(a) ++ ")"
show (TLoopEvery a _)    = "loop_every(" ++ show(a) ++ ")"
show (TNull a _)         = "null(" ++ show(a) ++ ")"
show (TOut a _)          = "out(" ++ show(a) ++ ")"
show (TOutAP a _ _)      = "out(" ++ show(a) ++ ")"
show (TOutPP a _ _)      = "out(" ++ show(a) ++ ")"
show (TOutF a _ _)       = "out(" ++ show(a) ++ ")"
show (TSelect a _)       = "select(" ++ show(a) ++ ")"
show (TStart a _)        = "start(" ++ show(a) ++ ")"
show (STInAP a _ _)      = "wakeup(" ++ show(a) ++ ")"
show (STInPP a _ _)      = "wakeup(" ++ show(a) ++ ")"
show (STOutAP a _ _)     = "wakeup(" ++ show(a) ++ ")"
show (STOutPP a _ _)     = "wakeup(" ++ show(a) ++ ")"
show (STDelayEnd a _)    = "timeout(" ++ show(a) ++ ")"
show (STLoopEnd a _)     = "timeout(" ++ show(a) ++ ")"
show (STInEnd a _)       = "timeout(" ++ show(a) ++ ")"
show (STOutEnd a _)      = "timeout(" ++ show(a) ++ ")"
show (STTime n)          = "time"
```

Listing 4.11. The *show* function for the *TTransition* data type

We can distinguish some subsets of transitions:

- *Communication transitions*: *TIn, TOut, TInAP, TInPP, TInF, TOutAP, TOutPP, TOutF, STInAP, STInPP, STOutAP, STOutPP*.
- *Communication in-out transitions*: *TInAP, TInPP, TInF, TOutAP, TOutPP, TOutF, STInAP, STInPP, STOutAP, STOutPP*.
- *Agents' communication in-out transitions*: *TInAP, TInPP, TInF, TOutAP, TOutPP, TOutF*.
- *System dropping communication transitions*: *STInEnd, STOutEnd*.

To check whether a transition belongs to one of the indicated subsets we can use the following Haskell model functions:

```
isCommTransition :: TTransition -> Bool
isInOutTransition :: TTransition -> Bool
isAgentInOutTransition :: TTransition -> Bool
isInOutEndTransition :: TTransition -> Bool
```

The Haskell model provides some auxiliary functions used to obtain some selected information about transitions.

Functions *transAgent* and *transNumber* returns the name of the agent that executes the transition (step) and the corresponding statement number respectively. Both functions provide results for all transitions except *STTime*.

```
transAgent :: TTransition -> Agent
transNumber :: TTransition -> Int
```

Functions *fstPort* and *sndPort* return the first and the second port of the communication transition respectively. The former function provides result for all communication transition, while the latter only for communication in-out transitions.

```
fstPort :: TTransition -> Port
sndPort :: TTransition -> Port
```

Moreover, function *sndAgent* returns the name of the second agent of a communication in-out transition:

```
sndAgent :: TTransition -> Agent
```

4.5 Functions *enable* and *fire*

The Haskell representation of an Alvis model behaviour is based on the so-called *enable-fire* approach which takes inspiration from Petri nets [12]. The *enable* function takes a model state and an agent name and provides a list of transitions of the agent that are enabled in the state. The *fire* function takes an enabled transition and a state and provides the list of states that are result of firing the transition in the given state.

```
enable :: State -> Agent -> [TTransition]
fire :: TTransition -> State -> [State]
```

Let us discuss in detail the activity of individual transitions. The following assumptions are valid for all **E** rules:

- A and B are active agents with ports $A.a$ and $B.b$ respectively.
- C and D are passive agents with ports $C.c$ and $D.d$ respectively.
- n denotes the number of the statement the considered transition refers to.
- $context(C)$ denotes the active agent in which context C works.
- The underscore mark as a wildcard.
- For a communication in-out transition the corresponding communication channel is defined in the communication diagram.

Rule E1. `TDelay A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out a *delay* statement.

Rule E2. `TDelay C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out a *delay* statement, $am(context(C)) = \mathsf{X}$.

Rule E3. `TExec A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out an *exec* statement.

Rule E4. `TExec C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out an *exec* statement, $am(context(C)) = \mathsf{X}$.

Rule E5. `TExit A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out an *exit* statement.

Rule E6. `TExit C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out an *exit* statement, $am(context(C)) = \mathsf{X}$.

Rule E7. `TIn A_a n` is enable iff: $am(A) = \mathsf{X}$, $pc(A)$ points out an *in* statement, $ci(A)$ does not contain a `CProc _` entry, and transitions `TInAP A_a _ n` and `TInF A_a _ n` are not enable (see rules E9 and E11).

Remark: The compiler optimizes code and certain conditions may be omitted e.g. it is not necessary to check whether $ci(A)$ contains a `CProc _` entry if the model does not contain passive agents. This remark applies to most communication transitions.

Rule E8. `TIn C_c n` is enable iff:

1. $am(C) = \mathsf{T}$ and $pc(C)$ points out an *in* statement, $am(context(C)) = \mathsf{X}$ if $C.c$ is a procedure port.
2. $am(C) = \mathsf{T}$, $pc(C)$ points out an *in* statement, $ci(C)$ does not contain a `CProc _` entry, $am(context(C)) = \mathsf{X}$ and transition `TInPP C_c _ n` is not enable if $C.c$ is a non-procedure port.

Rule E9. `TInAP A_a C_c n` is enable iff: $am(A) = \mathsf{X}$, $pc(A)$ points out an *in* statement, $ci(A)$ does not contain a `CProc _` entry, $am(C) = \mathsf{W}$, $ci(C)$ contains `COut C_c` entry.

Rule E10. `TInPP C_c D_d n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out an *in* statement, $am(context(C)) = \mathsf{X}$, $ci(C)$ does not contain a `CProc _` entry, $am(D) = \mathsf{W}$, $ci(D)$ contains `COut D_d` entry.

Remark: $C.c$ is non-procedure port.

Rule E11. `TInF A_a B_b n` is enable iff: $am(A) = \mathsf{X}$, $pc(A)$ points out an *in* statement, $ci(A)$ does not contain a `CProc _` entry, $am(B) = \mathsf{W}$, $ci(B)$ contains `COut B_b` entry.

Rule E12. `TJump A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out a *jump* statement.

Rule E13. `TJump C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out a *jump* statement, $am(context(C)) = \mathsf{X}$.

Rule E14. `TLoop A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out a *loop* statement.

Rule E15. `TLoop C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out a *loop* statement, $am(context(C)) = \mathsf{X}$.

Rule E16. `TLoopEvery A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out a *loop every* statement.

Rule E17. `TLoopEvery C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out a *loop every* statement, $am(context(C)) = \mathsf{X}$.

Rule E18. `TNull A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out a *null* statement.

Rule E19. `TNull C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out a *null* statement, $am(context(C)) = \mathsf{X}$.

Rule E20. `TOut A_a n` is enable iff: $am(A) = \mathsf{X}$, $pc(A)$ points out an *out* statement, $ci(A)$ does not contain a `CProc _` entry, and transitions `TOutAP A_a _ n` and `TOutF A_a _ n` are not enable (see rules E22 and E24).

Rule E21. `TOut C_c n` is enable iff:

1. $am(C) = \mathsf{T}$ and $pc(C)$ points out an *out* statement, $am(context(C)) = \mathsf{X}$ if $C.c$ is a procedure port.
2. $am(C) = \mathsf{T}$, $pc(C)$ points out an *out* statement, $ci(C)$ does not contain a `CProc _` entry, $am(context(C)) = \mathsf{X}$ and transition `TOutPP C_c _ n` is not enable if $C.c$ is a non-procedure port.

Rule E22. `TOutAP A_a C_c n` is enable iff: $am(A) = \mathsf{X}$, $pc(A)$ points out an *out* statement, $ci(A)$ does not contain a `CProc _` entry, $am(C) = \mathsf{W}$, $ci(C)$ contains `CIn C_c` entry.

Rule E23. `TOutPP C_c D_d n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out an *out* statement, $am(context(C)) = \mathsf{X}$, $ci(C)$ does not contain a `CProc _` entry, $am(D) = \mathsf{W}$, $ci(D)$ contains `CIn D_d` entry.

Remark: $C.c$ is non-procedure port.

Rule E24. `TOutF A_a B_b n` is enable iff: $am(A) = \mathsf{X}$, $pc(A)$ points out an *out* statement, $ci(A)$ does not contain a `CProc _` entry, $am(B) = \mathsf{W}$, $ci(B)$ contains `CIn B_b` entry.

Rule E25. `TSelect A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out a *select* statement.

Rule E26. `TSelect C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out a *select* statement, $am(context(C)) = \mathsf{X}$.

Rule E27. `TStart A n` is enable iff: $am(A) = \mathsf{X}$ and $pc(A)$ points out a *start* statement.

Rule E28. `TStart C n` is enable iff: $am(C) = \mathsf{T}$ and $pc(C)$ points out a *start* statement, $am(context(C)) = \mathsf{X}$.

Rule E29. `STInAP A_a C_c n` is enable iff: $am(A) = \mathsf{W}$, $pc(A)$ points out an *in* statement, $ci(A)$ contains `CIn A_a` entry, $am(C) = \mathsf{W}$, $ci(C)$ contains `COut C_c` entry.

Rule E30. `STInPP C_c D_d n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out an *in* statement, $am(context(C)) = \mathsf{W}$, $ci(C)$ contains `CIn C_c` entry, $am(D) = \mathsf{W}$, $ci(D)$ contains `COut D_d` entry.

Remark: $C.c$ is non-procedure port.

Rule E31. `STOutAP A_a C_c n` is enable iff: $am(A) = \mathsf{W}$, $pc(A)$ points out an *out* statement, $ci(A)$ contains `COut A_a` entry, $am(C) = \mathsf{W}$, $ci(C)$ contains `CIn C_c` entry.

Rule E32. `STOutPP C_c D_d n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out an *out* statement, $am(context(C)) = \mathsf{W}$, $ci(C)$ contains `COut C_c` entry, $am(D) = \mathsf{W}$, $ci(D)$ contains `COut D_d` entry.

Remark: $C.c$ is non-procedure port.

Rule E33. `STDelayEnd A n` is enable iff: $am(A) = \mathsf{W}$, $pc(A)$ points out a *delay* statement, $ci(A)$ contains `CTimeout n` entry.

Rule E34. `STDelayEnd C n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out a *delay* statement, $am(context(C)) = \mathsf{W}$, $ci(C)$ contains `CTimeout n` entry.

Rule E35. `STLoopEnd A n` is enable iff: $am(A) = \mathsf{W}$, $pc(A)$ points out a *loop every* statement, $ci(A)$ contains `CTimeout n` entry.

Rule E36. `STLoopEnd C n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out a *loop every* statement, $am(context(C)) = \mathsf{W}$, $ci(C)$ contains `CTimeout n` entry.

Rule E37. `STInEnd A n` is enable iff: $am(A) = \mathsf{W}$, $pc(A)$ points out an *in* statement, $ci(A)$ contains `CTimeout n` entry.

Rule E38. `STInEnd C n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out an *in* statement, $am(context(C)) = \mathsf{W}$, $ci(C)$ contains `CTimeout n` entry.

Rule E39. `STOutEnd A n` is enable iff: $am(A) = \mathsf{W}$, $pc(A)$ points out an *out* statement, $ci(A)$ contains `CTimeout n` entry.

Rule E40. `STOutEnd C n` is enable iff: $am(C) = \mathsf{T}$, $pc(C)$ points out an *out* statement, $am(context(C)) = \mathsf{W}$, $ci(C)$ contains `CTimeout n` entry.

Rule E41. `STTime d` The transition is used to move the clock if none transition is enable, but at least one transition may be enable in future. The transition is not generated by the *enable* function, but is a part of the algorithm used to generate time LTS graphs (see Chapter **??**).

The *enable* function for model from Fig. 4.1 and Listing 4.2 is presented in Listing 4.12.

```
enable :: State -> Agent -> [TTransition]

enable s@(state1@(am1,pc1,ci1,pv1@()),
          state2@(am2,pc2,ci2,pv2@()),
          state3@(am3,pc3,ci3,pv3@(pv3_1))) A
  | am1 == X && pc1 == 1 = [TLoop A 1]
  | am1 == X && pc1 == 2 && (procFree ci1) && am3 == W
    && member (COut C_x) ci3 = [TInAP A_a C_x 2]
  | am1 == X && pc1 == 2 && (procFree ci1) = [TIn A_a 2]
  | am1 == W && pc1 == 2 && member (CIn A_a) ci1 && am3 == W
    && member (COut C_x) ci3 = [STInAP A_a C_x 2]
  | am1 == X && pc1 == 3 = [TJump A 3]
  | am1 == X && pc1 == 4 && (procFree ci1) && am2 == W
    && member (CIn B_p) ci2 && am3 == W && member (CIn C_y) ci3
    = [TOutF A_b B_p 4, TOutAP A_b C_y 4]
```

```
  | am1 == X && pc1 == 4 && (procFree ci1) && am2 == W
    && member (CIn B_p) ci2 = [TOutF A_b B_p 4]
  | am1 == X && pc1 == 4 && (procFree ci1) && am3 == W
    && member (CIn C_y) ci3 = [TOutAP A_b C_y 4]
  | am1 == X && pc1 == 4 && (procFree ci1) = [TOut A_b 4]
  | am1 == W && pc1 == 4 && member (COut A_b) ci1 && am3 == W
    && member (CIn C_y) ci3 = [STOutAP A_b C_y 4]
  | am1 == X && pc1 == 5 = [TJump A 5]
  | am1 == X && pc1 == 6 = [TNull A 6]
  | otherwise = []

enable s@(state1@(am1,pc1,ci1,pv1@()),
          state2@(am2,pc2,ci2,pv2@()),
          state3@(am3,pc3,ci3,pv3@(pv3_1))) B
  | am2 == X && pc2 == 1 = [TLoop B 1]
  | am2 == X && pc2 == 2 && (procFree ci2) && am1 == W
    && member (COut A_b) ci1 && am3 == W &&
    member (COut C_x) ci3 = [TInF B_p A_b 2, TInAP B_p C_x 2]
  | am2 == X && pc2 == 2 && (procFree ci2) && am1 == W
    && member (COut A_b) ci1 = [TInF B_p A_b 2]
  | am2 == X && pc2 == 2 && (procFree ci2) && am3 == W
    && member (COut C_x) ci3 = [TInAP B_p C_x 2]
  | am2 == X && pc2 == 2 && (procFree ci2) = [TIn B_p 2]
  | am2 == W && pc2 == 2 && member (CIn B_p) ci2
    && am3 == W && member (COut C_x) ci3 = [STInAP B_p C_x 2]
  | otherwise = []

enable s@(state1@(am1,pc1,ci1,pv1@()),
          state2@(am2,pc2,ci2,pv2@()),
          state3@(am3,pc3,ci3,pv3@(pv3_1))) C
  | am3 == T && pc3 == 1 && (xContext 3 s) = [TOut C_x 1]
  | am3 == T && pc3 == 2 && (xContext 3 s) = [TExec C 2]
  | am3 == T && pc3 == 3 && (xContext 3 s) = [TExit C 3]
  | am3 == T && pc3 == 4 && (xContext 3 s) = [TIn C_y 4]
  | am3 == T && pc3 == 5 && (xContext 3 s) = [TExec C 5]
  | am3 == T && pc3 == 6 && (xContext 3 s) = [TExit C 6]
  | otherwise = []
```

Listing 4.12. The *enable* function for model from Fig. 4.1 and Listing 4.2

The *enable* function uses some auxiliary Haskell model functions:

```
xContext :: Int -> State -> Bool
wContext :: Int -> State -> Bool
```

The *xContext* function checks whether the context agent of the k-th agent (k is the first parameter) is in *running* mode in the given state. Similarly, *wContext* checks whether the context agent is in *waiting* mode.

The *fire* function is used to compute results of a transition execution. The function takes as its first argument a transition and focuses on results of executing of this simple transition in the given state.

Let us discuss in detail the activity of individual transitions. The following assumptions are valid for all **F** rules:

- A and B are active agents with ports $A.a$ and $B.b$ respectively.
- C and D are passive agents with ports $C.c$ and $D.d$ respectively.
- n denotes the number of the statement the considered transition refers to.
- $context(C)$ denotes the active agent in which context C works.
- d denotes the number of time-units i.e. the time argument of *delay*, *loop every*, non-blocking *in* and non-blocking *out* statements.
- $nextpc(n)$ denotes the next program counter determined on the basis of the code structure for the considered agent and current program counter n.
- For passive agents k denotes the current program counter of their context agent.

The *fire* function uses the current state written as:

```
s@(state1@(am1,pc1,ci1,pv1@(pv1_1,pv1_2)),
   state2@(am2,pc2,ci2,pv2@()),
   state3@(am3,pc3,ci3,pv3@(pv3_1, pv3_2, pv3_3)),
   state4@(am4,pc4,ci4,pv4@()))
```

We will focus on the analysis of these elements that are changed when a transition fires. The results of executing individual transitions in the given state s are as follows:

Rule F1. `TDelay A n` sets $am(A)$ to W and inserts `CTimer n d` entry into $ci(A)$.

Special case: If the `--non-time` option is used, *delay* statements are treated as *null* ones.

Rule F2. `TDelay C n` sets $am(context(C))$ to W and inserts `CTimer n d` entry into $ci(C)$.

Special case: If the `--non-time` option is used, *delay* statements are treated as *null* ones.

Rule F3. `TExec A n` updates value of the corresponding parameter and sets $pc(A)$ to $nextpc(n)$

Special case: If the considered *exec* statement is the agent last statement (i.e. the agent finishes its work after the statement), the transition updates value of the corresponding parameter, sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F4. `TExec C n` updates value of the corresponding parameter and sets $pc(C)$ to $nextpc(n)$.

Rule F5. `TExit A n` sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F6. `TExit C n` sets $am(C)$ to W, $pc(C)$ to 0, $ci(C)$ to the set of C procedures accessible in the new state, and

- if the $C.c$ procedure has been called by an active agent A then the `CProc C_c` entry is removed from $ci(A)$ and $pc(A)$ is set to $nextpc(k)$;

- if the $C.c$ procedure has been called by a passive agent D then the `CProc C_c` entry is removed from $ci(D)$ and $pc(D)$ is set to $nextpc(m)$ (where m is the current value of $pc(D)$).

Special case: If the procedure calling was the agent A last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F7. `TIn A_a n`
Cases:

1. *Blocking in*: sets $am(A)$ to W and inserts `CIn A_a` entry into $ci(A)$;
2. *Non-blocking in,* $d = 0$: sets $pc(A)$ to $nextpc(n)$;
3. *Non-blocking in,* $d > 0$: sets $am(A)$ to W and inserts `CIn A_a`, `CTimer n d` entries into $ci(A)$

Special cases:

1. If the `--non-time` option is used, all time arguments of a non-blocking *in* statement are treated as zero.
2. If a non-blocking *in* statement with $d = 0$ is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F8. `TIn C_c n`
Cases:

1. $C.c$ is the current procedure port: updates value of the corresponding parameter (if a value has been sent), and sets $pc(C)$ to $nextpc(n)$;
2. $C.c$ is non-procedure port, *blocking in*: sets $am(context(C))$ to W and inserts `CIn C_c` entry into $ci(C)$.
3. $C.c$ is non-procedure port, *non-blocking in,* $d = 0$: sets $pc(C)$ to $nextpc(n)$;
4. $C.c$ is non-procedure port, *non-blocking in,* $d > 0$: sets $am(context(C))$ to W and inserts `CIn C_c`, `CTimer n d` entries into $ci(C)$.

Special case: If the `--non-time` option is used, all time arguments of a non-blocking *in* statement are treated as zero.

Rule F9. `TInAP A_a C_c n` inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the number of the first statement in `C_c` procedure, and sets $ci(C)$ to [ ].

Rule F10. `TInPP C_c D_d n` inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the number of the first statement in `D_d` procedure, and sets $ci(D)$ to [ ].

Rule F11. `TInF A_a B_b n` updates value of the corresponding parameter of agent A (if a value has been sent), sets $pc(A)$ to $nextpc(n)$, sets $am(B)$ to X, $pc(B)$ to $nextpc(m)$ (where m is the current value of $pc(B)$), and removes `COut B_b`, `CTimer n _` entries from $ci(B)$.

Remark: The $ci(B)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *out* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Special cases:

1. If the *in* statement is the agent A last statement, the transition updates value of the corresponding parameter (if a value has been sent), sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].
2. If the *out* statement is the agent B last statement, the transition sets $pc(B)$ to 0, $am(B)$ to F, and $ci(B)$ to [ ].

Rule F12. `TJump A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement after the *jump* label).

Rule F13. `TJump C n` sets $pc(C)$ to $nextpc(n)$ (the number of the first statement after the *jump* label).

Rule F14. `TLoop A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement in the loop if the corresponding guard is satisfied or the number of the first statement after the loop otherwise).

Remark: The lack of a guard is treated as the default guard equal to *True*.

Special case: If the guard is not satisfied and the *loop* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F15. `TLoop C n` sets $pc(C)$ to $nextpc(n)$.

Rule F16. `TLoopEvery A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement in the loop), inserts `CTimer n t` into $ci(A)$, where $t = d -$ *duration of the considered transition*.

Special case: If the `--non-time` option is used, *loop every* statements are treated as general loops with guards equal to *True* (see Rule F14).

Rule F17. `TLoopEvery C n` sets $pc(C)$ to $nextpc(n)$ (the number of the first statement in the loop), inserts `CTimer n t` into $ci(C)$, where $t = d -$ *duration of the considered transition*.

Special case: If the `--non-time` option is used, *loop every* statements are treated as general loops with guards equal to *True* (see Rule F15).

Rule F18. `TNull A n` sets $pc(A)$ to $nextpc(n)$.

Special cases:

1. If the *null* statement is the agent A last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].
2. If the *null* statement is the last one in a *loop every* statement, the transition sets $am(A)$ to W, and $pc(A)$ to the number of the corresponding *loop every* statement.

Rule F19. `TNull C n` sets $pc(C)$ to $nextpc(n)$

Special case: If the *null* statement is the last one in a *loop every* statement, the transition sets $am(context(C))$ to W, and $pc(C)$ to the number of the corresponding *loop every* statement.

Rule F20. `TOut A_a n`
Cases:

1. *Blocking out*: sets $am(A)$ to W and inserts `COut A_a` entry into $ci(A)$;
2. *Non-blocking out,* $d = 0$: sets $pc(A)$ to $nextpc(n)$;
3. *Non-blocking out,* $d > 0$: sets $am(A)$ to W and inserts `COut A_a` , `CTimer n d` entries into $ci(A)$

Special cases:

1. If the `--non-time` option is used, all time arguments of a non-blocking *out* statement are treated as zero.
2. If a non-blocking *out* statement with $d = 0$ is the agent last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F21. `TOut C_c n`
Cases:

1. $C.c$ is the current procedure port: sets $pc(C)$ to $nextpc(n)$;
2. $C.c$ is non-procedure port, *blocking out*: sets $am(context(C))$ to W and inserts `COut C_c` entry into $ci(C)$.
3. $C.c$ is non-procedure port, *non-blocking out,* $d = 0$: sets $pc(C)$ to $nextpc(n)$;
4. $C.c$ is non-procedure port, *non-blocking out,* $d > 0$: sets $am(context(C))$ to W and inserts `COut C_c` , `CTimer n d` entries into $ci(C)$.

Special case: If the `--non-time` option is used, all time arguments of a non-blocking *out* statement are treated as zero.

Rule F22. `TOutAP A_a C_c n` inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the number of the first statement in `C_c` procedure, and sets $ci(C)$ to [ ].

Rule F23. `TOutPP C_c D_d n` inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the number of the first statement in `D_d` procedure, and sets $ci(D)$ to [ ].

Rule F24. `TOutF A_a B_b n` sets $pc(A)$ to $nextpc(n)$, updates value of the corresponding parameter of agent B (if a value has been sent), sets $am(B)$ to X, sets $pc(B)$ to $nextpc(m)$ (where m is the current value of $pc(B)$), and removes the `CIn B_b` , `CTimer n _` entries from $ci(B)$.

Remark: The $ci(B)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *in* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Special cases:

1. If the *out* statement is the agent A last statement, the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].
2. If the *in* statement is the agent B last statement, the transition updates value of the corresponding parameter (if a value has been sent), sets $pc(B)$ to 0, $am(B)$ to F, and $ci(B)$ to [ ].

Rule F25. `TSelect A n` sets $pc(A)$ to $nextpc(n)$ (the number of the first statement in the first open branch or the number of the first statement after the *select* statement if all branches are closed).

Special case: If all branches are closed and the *select* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F26. `TSelect C n` sets $pc(C)$ to $nextpc(n)$ (the number of the first statement in the first open branch or the number of the first statement after the *select* statement if all branches are closed).

Rule F27. `TStart A n` sets $pc(A)$ to $nextpc(n)$, and if the agent B (parameter of the *start* statement) is in the *init* mode sets $am(B)$ to X and $pc(B)$ to 1.

Special case: If the considered *start* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F28. `TStart C n` sets $pc(C)$ to $nextpc(n)$, and if the agent B (parameter of the *start* statement) is in the *init* mode sets $am(B)$ to X and $pc(B)$ to 1.

Rule F29. `STInAP A_a C_c n` sets $am(A)$ to X, removes `CIn A_a`, `CTimer n _` entries from $ci(A)$, inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the number of the first statement in the `C_c` procedure, and sets $ci(C)$ to [ ].

Remark: The $ci(A)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *in* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F30. `STInPP C_c D_d n` sets $am(context(C))$ to X, removes `CIn C_c`, `CTimer n _` entries from $ci(C)$, inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the number of the first statement in the `D_d` procedure, and sets $ci(D)$ to [ ].

Remark: The $ci(C)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *in* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F31. `STOutAP A_a C_c n` sets $am(A)$ to X, removes `COut A_a`, `CTimer n _` entries from $ci(A)$, inserts `CProc C_c` into $ci(A)$, sets $am(C)$ to T, sets $pc(C)$ to the number of the first statement in the `C_c` procedure, and sets $ci(C)$ to [ ].

Remark: The $ci(A)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *out* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F32. `STOutPP C_c D_d n` sets $am(context(C))$ to X, removes `COut C_c`, `CTimer n _` entries from $ci(C)$, inserts `CProc D_d` into $ci(C)$, sets $am(D)$ to T, sets $pc(D)$ to the number of the first statement in the `D_d` procedure, and sets $ci(D)$ to [ ].

Remark: The $ci(C)$ contains a `CTimer n _` entry only if the communication has been initialised with a non-blocking *out* statement ($d > 0$) and time models are considered (the `--non-time` option is not used).

Rule F33. `STDelayEnd A n` sets $am(A)$ to X, sets $pc(A)$ to $nextpc(n)$, removes `CTimeout n` from $ci(A)$.

Special case: If the considered *delay* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F34. `STDelayEnd C n` sets $am(context(C))$ to X, sets $pc(C)$ to $nextpc(n)$, removes `CTimeout n` from $ci(C)$.

Rule F35. `STLoopEnd A n` sets $am(A)$ to X, removes `CTimeout n` from $ci(A)$.

Rule F36. `STLoopEnd C n` sets $am(context(C))$ to X, removes `CTimeout n` from $ci(C)$.

Rule F37. `STInEnd A n` sets $am(A)$ to X, sets $pc(A)$ to $nextpc(n)$, removes `CTimeout n` and `CIn A_a` entries from $ci(A)$.

Special case: If the considered *in* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F38. `STInEnd C n` sets $am(context(C))$ to X, sets $pc(C)$ to $nextpc(n)$, removes `CTimeout n` and `CIn C_c` entries from $ci(C)$.

Rule F39. `STOutEnd A n` sets $am(A)$ to X, sets $pc(A)$ to $nextpc(n)$, removes `CTimeout n` and `COut A_a` entries from $ci(A)$.

Special case: If the considered *out* statement is the agent last statement the transition sets $pc(A)$ to 0, $am(A)$ to F, and $ci(A)$ to [ ].

Rule F40. `STOutEnd C n` sets $am(context(C))$ to X, sets $pc(C)$ to $nextpc(n)$, removes `CTimeout n` and `COut C_c` entries from $ci(C)$.

Rule F41. `STTime d` The transition represent a passage of time. It updates time entries of all agents. For more details see Chapter **??**.

The *fire* function for model from Fig. 4.1 and Listing 4.2 is presented in Listing 4.12.

```
fire :: TTransition -> State -> [State]

fire (TInF A_a B_b 1) s@(state1@(am1,pc1,ci1,pv1@()),
                         state2@(am2,pc2,ci2,pv2@()),
                         state3@(am3,pc3,ci3,pv3@()))
  | pc2 == 2  = [((F,0,empty,pv1),(F,0,empty,pv2),state3)]

fire (TIn A_a 1) s@(state1@(am1,pc1,ci1,pv1@()),...
  = [((W,pc1,insert (CIn A_a) ci1,pv1),state2,state3)]

fire (TNull B 1) s@(state1@(am1,pc1,ci1,pv1@()),...
  = [(state1,(am2,2,ci2,pv2),state3)]

fire (TOutF B_b A_a 2) s@(state1@(am1,pc1,ci1,pv1@()),...
  | pc1 == 1  = [((F,0,empty,pv1),(F,0,empty,pv2),state3)]
```

```
fire (TOutF B_b C_a 2) s@(state1@(am1,pc1,ci1,pv1@()),...
  | pc3 == 1  = [(state1,(F,0,empty,pv2),(F,0,empty,pv3))]

fire (TOut B_b 2) s@(state1@(am1,pc1,ci1,pv1@()),...
  = [(state1,(W,pc2,insert (COut B_b) ci2,pv2),state3)]

fire (TInF C_a B_b 1) s@(state1@(am1,pc1,ci1,pv1@()),...
  | pc2 == 2  = [(state1,(F,0,empty,pv2),(F,0,empty,pv3))]

fire (TIn C_a 1) s@(state1@(am1,pc1,ci1,pv1@()),...
  = [(state1,state2,(W,pc3,insert (CIn C_a) ci3,pv3))]
```

Listing 4.13. The *fire* function for model from Fig. 4.1 and Listing 4.2

5

LTS graphs for non-time models

This chapter provides description of the labelled transition systems (LTS graphs) generation algorithm for non-time models. This is the simplest approach to LTS graphs generation because we consider changes of states that are results of firing single transitions as they are provided by the *fire* function.

5.1 LTS graphs

For a pair of states S, S' we say that S' is *directly reachable* from S iff there exists a transition t enable in S such that firing t leads to state S' ($S{-}t{\rightarrow}S'$).

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 $t^1, \ldots, t^k$ such that

$$S = S^1{-}t^1{\rightarrow}S^2{-}t^2{\rightarrow}\ldots{-}t^k{\rightarrow}S^{k+1} = S'. \tag{5.1}$$

The set of all states that are reachable from the initial state S_0 will be denoted by $\mathcal{R}(S_0)$.

The set of all transitions available for a particular model will be denoted by $\mathcal{T}$. For non-time models only transitions from Listing 5.1 are used.

```
TExec      Agent Int
TExit      Agent Int
TIn        Port Int
TInAP      Port Port Int
TInPP      Port Port Int
TInF       Port Port Int
TJump      Agent Int
TLoop      Agent Int
TNull      Agent Int
TOut       Port Int
TOutAP     Port Port Int
TOutPP     Port Port Int
TOutF      Port Port Int
```

```
TSelect     Agent Int
TStart      Agent Int
STInAP      Port Port Int
STInPP      Port Port Int
STOutAP     Port Port Int
STOutPP     Port Port Int
```

Listing 5.1. *Transitions* permissible in non-time models

States of an Alvis non-time model and transitions among them are represented using a labelled transition system (LTS graph for short). An LTS *graph* is directed graph LTS $= (\mathcal{R}(S_0), E, \mathcal{T}, S_0)$, where $\mathcal{R}(S_0)$ is the set of nodes, $\mathcal{T}$ is the set of labels, $E = \{(S, t, S') \colon S{-}t{\rightarrow}S' \land S, S' \in \mathcal{R}(S_0)\}$ is the set of edges and S_0 is the initial state.

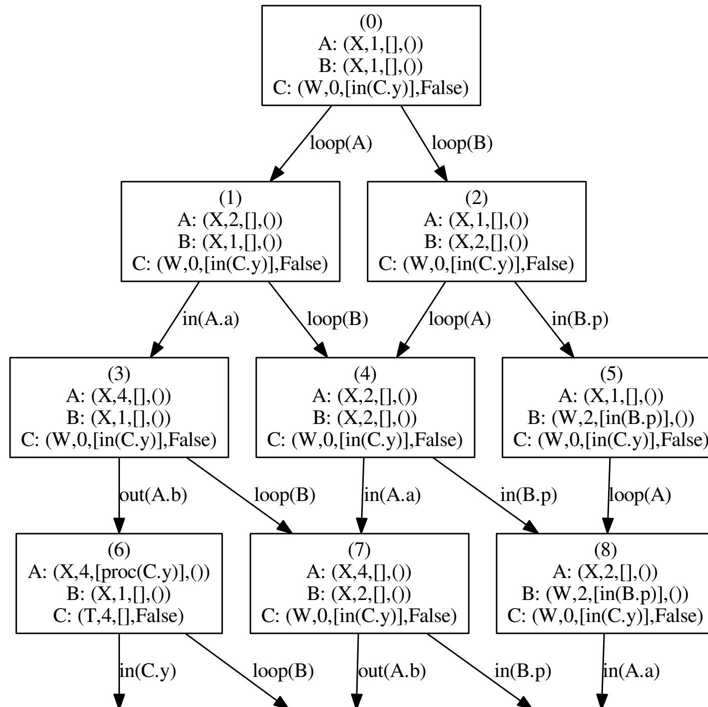

Fig. 5.1. Initial part of LTS graph for model from Fig. 4.1

The initial part of the LTS graph for the model from Fig. 4.1 is shown in Fig. 5.1. The figure has been obtained out of the LTS *dot* format automatically. The *dot* format is one of the Alvis export formats for LTS graphs. For more details see Chapter **??**.

5.2 Generation of LTS graphs using Haskell model representation

The Haskell model representation uses *Node* data type to represent nodes of an LTS graph. The type for non-time models is defined as follows:

```
type Node = (Int, State, [(TTransition, Int)])
```

A node is a triple: (node number, model state, list of output arcs). Each arc is described using its transition label and the number of destination node.

The LTS graphs generation algorithm is placed in *lts.hs* file (version for non-time models). An LTS graph is provided by *lts* function (see Listing 5.2) that is only a wrapper for *ltsgen* recursive function. The *ltsgen* starts the LTS graph generation using the initial state as the LTS graph node number 0 with initially empty set of output arcs.

```
lts = ltsgen [(0,s0,[])] (enableTransitions s0) [(0,s0,[])]

ltsgen :: [Node] -> [TTransition] -> [Node] -> [Node]
```

Listing 5.2. The *lts* and *ltsgen* functions

The *ltsgen* function takes a list of nodes to process, list of transitions enable in the first state of the list of unprocessed nodes and auxiliary list of already generated nodes. The function starts LTS graph generation using the *initial node* and the list of transitions enable in the *initial state*. For the current state and a transition enable in the state it computes the new states using *fire* function. Using the auxiliary list the algorithm checks whether such a node already exists. If so only a new arc is added for the current state. Otherwise a new state is appended to the both lists and the corresponding arc is added to the current state. The algorithm finishes graph generation if the list of nodes to process is empty.

The *ltsgen* function uses two auxiliary functions:

```
findState :: State -> [Node] -> Int
update :: ([Node], [Node]) -> TTransition
       -> [State]-> ([Node], [Node])
```

The former function takes a state and a list of nodes and provides the number of the LTS node that contains the given state or -1 if such a node does not exist. The latter function udpates the list of unprocessed nodes and the list of auxiliary nodes. For each state from the list of new states, the function checks whether the auxiliary list already contains a node with such a state. If so, only new arc is added to the first node of unprocessed nodes. Otherwise, a new node is added to both lists (and the corresponding arc to the first node of unprocessed nodes).

The generated LTS graph takes the form of list of nodes and the standard main function just writes it to the output file (see Listing 5.3).

```
main = do
  putStrLn (printDOT lts)
```

Listing 5.3. Example of a model *main* function

In case of user defined main function the *lst* list can be explored in any way e.g. we can filter states that fulfil the given condition, search for some paths in the graph etc. [13].

The Haskell approach to Alvis model verification requires Haskell programming skills, because the so-called *filtering functions* must be user-defined and included into the generated source file. Some of the functions are universal and can be included into any model, so it is possible to import them from an external Haskell module. However, most of these functions are based on the considered model *State* type and must be defined for a model individually.

Examples of universal filtering functions are given in Listing 5.4. The *deadState* function searches for states without outgoing arcs (dead states), while *singleOutState* function searches for states with single outgoing arc. Included comments illustrate the usage of these functions.

```
deadState :: Node -> Bool
deadState (n,s,ls) = ls == []
-- Prelude.filter deadState lts

singleOutState :: Node -> Bool
singleOutState (n,s,ls) = (length ls) == 1
-- Prelude.filter singleOutState lts
```

Listing 5.4. Examples of universal filtering functions

To print selected nodes the auxiliary functions show in Listing 5.5 can be used.

```
node2text :: Node -> String
node2text (n,((am1,pc1,ci1,pv1),(am2,pc2,ci2,pv2),
               (am3,pc3,ci3,pv3)),_)
  = (show n) ++ " A: (" ++ (show am1) ++ "," ++ (show pc1)
  ++ "," ++ (show (toList ci1)) ++ "," ++ (show pv1) ++")"
  ++ " B: (" ++ (show am1) ++ "," ++ (show pc1) ++ ","
  ++ (show (toList ci1)) ++ "," ++ (show pv1) ++")"
  ++ " C: (" ++ (show am1) ++ "," ++ (show pc1) ++ ","
  ++ (show (toList ci1)) ++ "," ++ (show pv1) ++")"

listOfNodes2text :: [Node] -> String
listOfNodes2text nodes = unlines (Prelude.map node2text nodes)
```

Listing 5.5. Auxiliary functions for printing selected nodes

The user defined *main* function using the filtering functions from Listing 5.4 is presented in Listing 5.6.

```
main = do
  putStrLn ("Dead states:")
  putStrLn (listOfNodes2text (Prelude.filter deadState lts))
  putStrLn ("States with single output:")
  putStrLn (listOfNodes2text (
```

```
                        Prelude.filter singleOutState lts))
```

Listing 5.6. Example of user defined *main* function

Knowledge of the `State` type details is fundamental for implementing more sophisticated filtering functions. The main disadvantage of such functions is their adaptation to the given model. Examples of special filtering functions implemented for the model from Fig. 4.1 are shown in Listing 5.7. The *isARunning* function searches for states with agent A in the *running* mode. Presented functions use the Haskell pattern matching mechanism. The underscore sign is a wild-card and its role changes depending on the place e.g. the first one replaces the number of a node, the second one – the program counter of agent A and the fifth – the state of agent B. The *twoWaiting* function searches for states with two agents in the *waiting* mode.

```
isARunning :: Node -> Bool
isARunning (_,((X,_,_,_),_,_),_)  = True
isARunning _ = False

twoWaiting :: Node -> Bool
twoWaiting (_,((W,_,_,_),(W,_,_,_),_),_)  = True
twoWaiting (_,((W,_,_,_),_,(W,_,_,_)),_)  = True
twoWaiting (_,(_,(W,_,_,_),(W,_,_,_)),_)  = True
twoWaiting _ = False
```

Listing 5.7. Examples of special filtering functions

The functions presented so far are used to search for states which fulfil given filter condition. As shown in Listing 5.6, they are used together with the standard *filter* function. More elaborated functions may search an LTS graph oneself. Example of such a function is given in Listing 5.8. The *node2node* function returns pairs of nodes connected with an arc with the given transition. It uses two auxiliary functions: *ithNode* searches for a node with the given number and *endNodeNo* searches for the number of the end node for the given arc.

```
ithNode :: Int -> [Node] -> Node
ithNode i ((number, state, labels):nodes)
  | i == number = (number, state, labels)
  | otherwise = ithNode i nodes

endNodeNo :: TTransition -> [(TTransition, Int)] -> Int
endNodeNo _ [] = -1
endNodeNo tr ((transition, number):labels)
  | tr == transition = number
  | otherwise = endNodeNo tr labels

node2node :: TTransition -> [Node] -> [Node] -> [(Node,Node)]
node2node _ _ [] = []
node2node label ltscopy ((number, state, labels):nodes) =
  if k /= -1
```

```
  then ((number, state, labels),(ithNode k ltscopy))
    : (node2node label ltscopy nodes)
  else node2node label ltscopy nodes
  where k = endNodeNo label labels
```

Listing 5.8. Examples of special filtering functions

Example of a main function using the functions from Listing 5.8 is shown in Listing 5.9.

```
main = do
  let pairs = node2node (TJump A 5) lts lts
  let text = unlines (Prelude.map pairOfNodes2text pairs)
  putStrLn (text)
```

Listing 5.9. Example of user defined *main* function

6

LTS graphs for time models

This chapter provides description of the labelled transition systems (LTS graphs) generation algorithm for time models. This is the default Alvis approach. The Alvis time model is based on the idea of a *global clock* used to measure the duration of model steps. In contrast to non-time models, an arc in an LTs graph represents not a single transition, but a set of transitions executed in parallel. As previously, the LTS graphs generation algorithm is based on the *enable* and *fire* functions.

6.1 Time in Alvis models

A user may assign an integer to each code layer statement represented by a transition. In the Haskell model representation this assignment is represented by the *duration* function:

```
duration :: Agent -> Int -> Int
```

The function takes an agent name and a statement number and provides the duration of the given statement. By default the function generated by Alvis Compiler provides 1 for each statement. A user is expected to redefine the function in the generated Haskell source file.

The duration function is user-friendly, but it is not used in the Haskell model representation directly. We use the *transDuration* function to get the duration for the given model transition:

```
transDuration :: TTransition -> Int
```

By default, the duration of system transition is equal to 0. To change this assumption, the *systemTransitionsDuration* function must be redefined (see Listing 6.1).

```
systemTransitionsDuration :: Int
systemTransitionsDuration = 0
```

Listing 6.1. Default definition of *systemTransitionsDuration* function

Moreover, the current version of Alvis uses four code statements that use time explicitly:

- `delay t` – postpones an agent for the given time;
- `loop (every t) {...}` – repeats loop contents every specified number of time-units;
- `in (t) ...` – non-blocking *in* statement – waits at most t time-units for finalisation of communication;
- `out (t) ...` – non-blocking *out* statement.

Each of these statements has assigned a local timer used to measure the elapsed time associated with the statement. The timer is represented by `CTimer` context entry. After the time for which the timer was set the entry is changed into `CTimeout` that usually enables a system transition that wakes up the agent.

The simple Alvis model presented in Fig. 6.1 will be used to illustrate time aspects in Alvis models. To simplify referring to the code layer, the comments contain the numbers of Alvis statements and their durations. The definition of *duration* function is given in Listing 6.3. The model contains all aforementioned statements and it allows us to illustrate many of the problems typical for LTS graphs for time Alvis models.

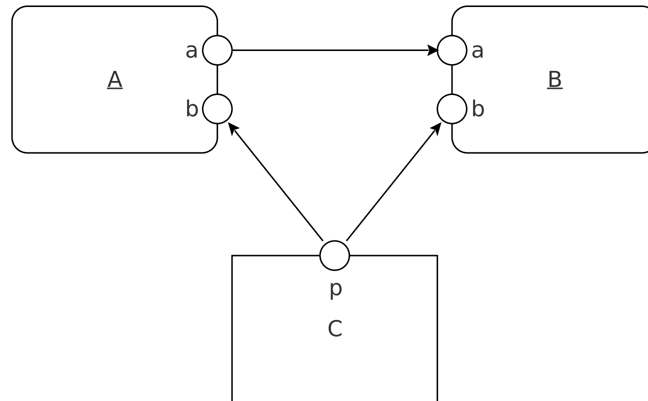

Fig. 6.1. Example of Alvis model (communication diagram)

```
agent A {
  loop (every 12) {    -- 1 / 1
    out (3) a;         -- 2 / 1
    delay 1;           -- 3 / 1
    in (2) b;          -- 4 / 2
  }
}
```

```
agent B {
  loop {                    -- 1 / 1
    in a;                   -- 2 / 2
    delay 4;                -- 3 / 1
    in b;                   -- 4 / 2
  }
}

agent C {
  proc p {
    out p;                  -- 1 / 2
    exit;                   -- 2 / 1
  }
}
```

Listing 6.2. Example of Alvis model (code layer)

```
duration :: Agent -> Int -> Int
duration A 1 = 1
duration A 2 = 1
duration A 3 = 1
duration A 4 = 2
duration A 5 = 0
duration B 1 = 1
duration B 2 = 2
duration B 3 = 1
duration B 4 = 2
duration C 1 = 2
duration C 2 = 1
```

Listing 6.3. *Duration* function for model from Fig. 6.1

```
CTimer  Int Int
COTimer Int Int
CSFT    Int
CNSFT   Int
```

Listing 6.4. Context information entries that use time explicitly (the last argument)

TO DO: to be continue

References

1. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
2. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. O'Reilly Media, Sebastopol, CA, USA (2008)
3. Jensen, K., Kristensen, L.: Coloured Petri nets. Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
4. Szpyrka, M., Matyasik, P., Biernacki, J., Biernacka, A., Wypych, M., Kotulski, L.: Hierarchical communication diagrams. Computing and Informatics **35**(1) (2016) 55–83
5. Szpyrka, M., Matyasik, P., Mrówka, R., Kotulski, L.: Formal description of Alvis language with α^0 system layer. Fundamenta Informaticae **129**(1-2) (2014) 161–176
6. 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 Studies in Computational Intelligence. Springer-Verlag (2011) 315–341
7. Barnes, J.: Programming in Ada 2005. Addison Wesley (2006)
8. Burns, A., Wellings, A.: Concurrent and real-time programming in Ada 2005. Cambridge University Press (2007)
9. Matyasik, P., Szpyrka, M., Wypych, M., Biernacki, J.: Communication between agents in Alvis language. In: Proc. of Mixdes 2016, the 23nd International Conference Mixed Design of Integrated Circuits and Systems, Łódź, Poland (June 23–25 2016)
10. 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
11. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Computer Aided Verification. Volume 8559 of Lecture Notes in Computer Science., Springer (2014) 334–342
12. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
13. Szpyrka, M., Matyasik, P., Wypych, M.: Generation of labelled transition systems for alvis models using haskell model representation. In: Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming (CS&P 2013). Volume 1032., Warsaw, Poland, CEUR Workshop Proceedings (2013) 409–420