Alvis Reference Manual v. 0.06

Marcin Szpyrka

February 9, 2010

Contents

Chapter 1

Alvis Behaviour Description language

Alvis Behaviour Description language (or shortly ABD language) is used to describe behaviour of individual agents in an Alvis model. ABD language has its origin in CCS ([]) and XCCS ([]) process algebras. However, to make the language more convenient from practical (engineering) point of view, algebraic equations and operators have been replaced with instructions typical for high level programming languages.

An Alvis model consists of two layers – graphical and code ones. The *graphical layer* is used to design the communication channels among agents (parts of the considered model). The detailed description of the layer is presented in chapter **??**. The *code layer* is used to define:

- data types used in the model under consideration,
- functions for data manipulation
- behaviour of individual agents.

Alvis Behaviour Description language uses Haskell functional programming language to define and manipulate data types. Therefore, the Haskell syntax has an influence on ABD language syntax.
...

1.1 Identifiers

1.2 Case Sensitivity in Alvis

Haskell is case sensitive language. ABD language is based on Haskell, thus it is case sensitive too. Haskell requires type names to start with an uppercase letter, and variable names to start with a lowercase letter. We follow in Haskell footsteps. Moreover, ABD language requires agent names to start with uppercase letter, and port names to start with a lowercase letter.

1.3 Keywords

This section presents Alvis and Haskell keywords. They cannot be used as identifiers in Alvis models. Some other words e.g. *True*, *False* might seem like keywords, but they are actually literals and of course cannot be used as identifiers too.

Alvis keywords: agent, alt, cli, connections, delay, exec, far, in, jump, out, ports, sti (list not completed yet)

Haskell keywords: as, case, of, class, data, default, deriving, do, forall, foreign, hiding, if, then, else, import, infix, infixl, infixr, instance, let, in, mdo, module, newtype, qualified, rec, type, where.

1.4 General file structure

The code layer of an Alvis model is stored in a textual source file. The general structure of such a file is shown in listing 1.1.

```
-- Preamble:
--   types
--   constants
--   functions


-- Implementation:
--   agents
```

Listing 1.1: General structure of an ABD file

The *preamble* contains definition of types, constans and functions used to manipulate data in a model. The preamble is encoded in pure Haskell.

The *implementation* contains definitions of agents' behaviour. This part is encoded using native ABD language statements, but the preamble contents is used to represent parameters values and to manipulate them.

1.5 Data types

ABD language uses 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 contains 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.

Moreover, Haskell uses a type reference process. It means that compiler can automatically deduce the type of almost all expressions in a program.

1.5.1 Basic Types

Table 1.1 contains selected basic Haskell's types recommended to be used in ABD language.

Table 1.1: Selected basic Haskell's types recommended to be used in ABD language

Type name	*Description*
`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.

1.5.2 Composite data types

The most common composite data types in Haskell (and ABD) are *lists* and *tuples*.

A *list* is a sequence of elements of the same type, with the elements being enclosed in square parentheses and separated by commas:

```
[1,2,3,4]           -- list of integers, type [Int]
['a','b','c']       -- list of characters, type [Char] (String)
[True,False]        -- list of Boolean values, type [Bool]
```

Haskell represents a text string as a list of `Char` values. The list shown in line 2 can be shortly written as `"abc"`. The empty list is denoted by `[]`.

A *tuple* is a sequence of elements of possibly different types, with the elements being enclosed in round parentheses and separated by commas:

```
(1,2)               -- type (Int,Int)
('a',True)          -- type (Char,Bool)
("abc",1,True)      -- type (String,Int,Bool)
```

Tuples containing different number of types of elements have disting types, as do tuple whose types appear in different orders.

1.5.3 Type synonyms

To make the source code more readable, we can introduce a synonym for an existing type:

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

1.5.4 Defining a new data type

A new data type is defined using the `data` keyword:

```
data AgentDescription = AgentDesc Int String [String]
```

The identifier after the `data` keyword is the name of the new type, while the identifier after the = sign is called *value constructor* (data constructor). A value constructor is a special function, which name, as the name of type, must start with an uppercase letter. Types placed after a value constructor name are called *components* of the type. A value constructor is used to create a new value of the corresponding type, e.g.:

```
myAgent = AgentDesc 1 "Buffer" ["put","get"]
```

A value constructor name can be the same as the type one.

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

An algebraic data type can be used to define an enumeration type, e.g.:

```
data Move = East | South | West | North
```

Moreover, an algebraic data type can be used to define a type with different variants of data, e.g.:

```
type Point = (Double,Double)
data Shape = Circle Point Double
           | Rectangle Point Point
```

1.5.5 Structures

1.6 Constants

Constants are defined using parameterless Haskell functions:

```
e = 2,718281828
size = 10
name = "Agent"
```

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

1.7 Implementation part

The implementation part contains definitions of agents' behaviour. It contains at least one *agent block* of the following form:

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

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 spaces:

```
agent AgentName1 AgentName2 AgentName3
-- declaration of parameters
-- agent body
```

In Haskell and ABD language indentation is important, it continues an existing definition, instead of starting a new one. It is however recommended to finish an agent body with an empty line.

1.8 Parameters

Parameters are defined using Haskell syntax. Each parameter is places in a separate line. The line starts with a parameter name, then the `::` symbol is places followed by the parameter type:

```
size :: Int
inputData :: (Int, Char)
queue :: [Double]
signal :: TrafficSignal
```

1.9 Communication with outside world

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

1) connections to the port in the corresponding communication diagram;

2) statements used in the code layer.

Let's focus on the code layer. Communication diagrams are described in chapter **??**.

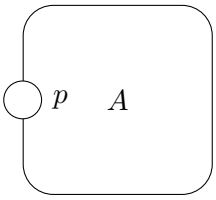

Figure 1.1: Agent A with port p

Figure 1.1 presents an agent A with a single port p. Suppose, the graphical layer (communication diagram) does not restrict the role of port p. Thus, the port can be used as an input or output one. Moreover, any communication through the port can be a pure synchronisation or a single value (probably of an composed type) can be sent.

1.9.1 Pure synchronisation

A *pure synchronisation* is a communication without sending values of parameters. Let's consider the following example:

```
agent A
  in p
  delay 1000
  out p
```

Agent A collects a synchronisation signal through port p, waits 1 second and sends a synchronisation signal through the same port.

1.9.2 Value passing communication

A value passing communication not only synchronises two agents but also a parameter value is send through the corresponding ports. Let's consider the following example:

```
agent A
  i :: Int
  in p i
  delay 1000
  out p i
```

Agent A collects an integer value through port p and assigns it to parameter i. Then, it waits 1 second and sends the value of parameter i through the same port.

Instead of a parameter name, an expression of the suitable type or a constant can be used in the *out* statement. In the following example, the doubled value of i is sent through port p:

```
agent A
  i :: Int
  in p i
  delay 1000
  out p (2 * i)
```

An expression must be placed inside round brackets.

1.9.3 Guards

A *guard* is an additional constraint which must be fulfilled before the statement is executed. Guards are logical expressions, written in Haskell, placed inside round brackets after the statement name. In the following example, the *out* statement is executed only if the value of i is less than 5:

```
agent A
  i :: Int
  in p i
  delay 1000
  out (i < 5) p i
```

When a guard is used in the *in* statement, then it can use only already stored values. In other words, in the following example the current (old) value of i is used. It means that the new value will be accepted only if the current value is less than 5:

```
in (i < 5) p i
```

1.10 Recursion

Recursion is the mechanism for looping in ABD language. Two language concept are use for this purpose *labels* and *jump statement*. Labels in Alvis are identifiers followed by a colon. The jump statement is composed of the *jump* key word and a label name (without colon). If necessary, the `jump` key word may be followed by a guard placed inside round brackets.

Let's consider agent `Adder` shown in Fig. 1.2. Its behaviour is defined as follows:

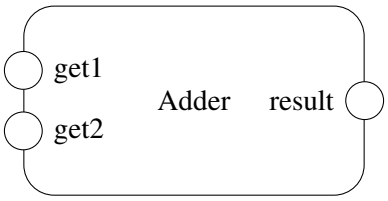

Figure 1.2: Agent `Adder`

```
agent Adder
  x :: Int
  y :: Int
start:
  in get1 x
  in get2 y
  out result (x + y)
  jump start
```

Agent `Adder` takes values for parameters `x` and `y` through ports `get1` and `get2` respectively, sends the sum of those parameters through port `result` and repeats the sequence of statements.

1.11 Delays

To postpone an agent for some time the *delay* statement is used. The statement is composed of the *delay* key word, a guard (if necessary) and a time period in milliseconds, e.g.

```
delay 500
```

1.12 Parameters manipulation

As it was said before, in the implementation part, the = symbol stands for the assignment operator. The operator is used as a part of the *exec* statement. Thus, to assing a literal value 7 to an integer parameter `x` the following statement can be used:

```
exec x = 7
```

If necessary, the `exec` key word may be followed by a guard placed inside round brackets.

```
exec (x == 0) x = 7
```

The *exec* statement is the default one in ABD language. Therefore, the `exec` keyword can be omitted if no guard is used, and the first assignment can be simply written as:

```
x = 7
```

The assignment operator can be also followed by an expression. ABD language uses Haskell to define and manipulate data types. Thus, such an expression takes the form of Haskell function call, e.g.:

```
exec x = x + 1
exec x = rem x 3
exec (y >= 0) x = sqrt y
```

Of course, the `exec` key word can be omitted in the first and the second example.

1.13 Alternatives

In order to allow for description of agents whose behaviour may follow different alternative paths, ABD language offer the *alt* statement. The simplest form of the *alt* statement allows selecting from, one or more alternatives.

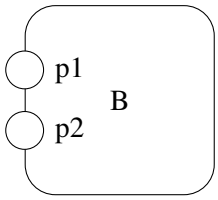

Figure 1.3: Agent `B`

Let's consider agent `B` show in Fig. 1.3 with the following definition:

```
agent B
  x :: Int
  y :: Int
start:
  alt in p1
      x = x + 1
      jump start
  alt in p2
      y = y + 1
      jump start
```

Agent `B` offers a choice between two alternatives. At the beginning of the *main loop* the agent is ready to collect a signal through port `p1` or port `p2`. If port `p1` is selected, variable `x` is increased and the loop is repeated. In similar way the second alternative if performed.

Each of alternatives of the *alt* statement is called a *branch*. All statements that constitutes a branch must have the same indentation, e.g. the `in`, `exec` and `jump` statements in the first branch. The considered code can be also written as follows:

```
agent B
  x :: Int
  y :: Int
start:
  alt
    in p1
    x = x + 1
    jump start
```

```
  alt
    in p2
    y = y + 1
    jump start
```

The selection between branches can depend on guards associated with each branch, e.g. statements in the first branch. The considered code can be also written as follows:

```
agent B
  x :: Int
  y :: Int
start:
  alt (x < 10)
    in p1
    x = x + 1
    jump start
  alt (y < 20)
    in p2
    y = y + 1
    jump start
```

When an *alt* statement is to be executed, all guards are evaluated to determinw which branches are *open*. An branch is called *open*, if it does not have a guard attached or its guard evaluates to *true*. Otherwise, a branch is called *closed*. If more than one branch is open, the choice between them is indeterministic.