AGH University of Science and Technology
Computer Science Laboratory
Department of Automatics
Al. Mickiewicza 30
30-059 Kraków, POLAND

Introduction to Alvis internal language syntax

Marcin Szpyrka, Piotr Matyasik, Rafał Mrówka
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND
`{mszpyrka,ptm,Rafal.Mrowka}@agh.edu.pl`

Published online: 05.05.2010

CSL Technical Report No. 1/2010

© by AGH University of Science and Technology, Kraków, Poland, 2008-9.

Computer Science Laboratory Technical Reports (CSL TR) series serves as an accesible platform for publishing research results of the CS Lab staff members at Department of Automatics of AGH University of Science and Technology.

See the `http://cslab.ia.agh.edu.pl/csltr:start` for the series homepage.

The editorial board:

- *Main Editor*: Marcin Szpyrka, PhD, DSc
- Prof. Antoni Ligęza, PhD, DSc
- Prof. Tomasz Szmuc, PhD, DSc
- Grzegorz J. Nalepa, PhD

Cover design: Marcin Szpyrka

LaTeX class design: Marcin Szpyrka

Contact us at: mszpyrka@agh.edu.pl

CSL Technical Report No. 1/2010

Introduction to Alvis internal language syntax*

Marcin Szpyrka, Piotr Matyasik, Rafał Mrówka
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND
`{mszpyrka,ptm,Rafal.Mrowka}@agh.edu.pl`

Abstract. Alvis modelling language is a novel approach to the design and formal verification of embedded systems. Alvis has its origins in CCS and XCCS process algebras, but to describe individual agents it uses a high level programming language instead of algebraic equations. An Alvis model consists of two layers, a graphical and a code one. The report presents a step by step introduction to the syntax of the code layer. To describe the behaviour of individual agents Alvis uses its *behaviour description* Haskell based language. The report contains a series of small examples used to present the most important of the language statements.

Keywords: Alvis modelling language, Haskell, agent behaviour, syntax

1 Introduction

The aim of the report is to present an introduction to the syntax of Alvis internal language used to describe the behaviour of individual agents. Alvis is a successor of the XCCS modelling language [9], [3], [6], which was an extension of the CCS process algebra [7], [5], [1]. Alvis models, as well as XCCS, consist of two layers, a graphical and a code (textual) one. The graphical layer takes the form of a hierarchical directed graph called *communication diagram* and represents all agents and communication channels among them. The code layer is defined using an internal Alvis language called *Alvis Behaviour Description* language (or shortly ABD language). 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. The *code layer* is used to define:

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

The ABD language is based on the Haskell programming language. Haskell is a general purpose, purely functional programming language [8]. It provides, among other things, lazy evaluation, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, higher-order functions, a rich set of primitive datatypes etc. Haskell uses a system of monads to isolate all impure computations from the rest of the program. Haskell functions are pure functions i.e. they are functions in mathematical sense and they have no side effects. This functional core of Haskell has been used in the Alvis Behaviour Description language. Haskell is used to define data types for parameters and to define (pure) functions for data manipulation. Therefore, the Haskell syntax has a significant influence on the ABD language syntax.

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

The report is organised as follows. Section 2 presents some information about the general structure of the code layer. Section 3 deals with data types and parameters. Language statements used for the communication description are presented in Section 4. Methods of parameters manipulation are presented in Section 5. Section 6 deals with recursion and alternatives. Examples of single agents are presented in Section 7. A short summary is given in the final section.

2 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.

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

-- Implementation:
--   agents
```

Listing 1: General structure of an ABD file

The *preamble* contains definitions of types, constants and functions used to manipulate data in a model. The preamble is encoded in pure Haskell. The *implementation* contains definitions of the 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.

The implementation part contains definitions of the agents' behaviour. It contains at least one *agent block* as shown in Listing 2.

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

Listing 2: Structure of an agent block

It is possible to share one definition among a few agents. In such a case, a few comma separated agents' names are placed after the keyword *agent*. If necessary, an agent name is followed by its priority put inside round brackets.

Haskell and Alvis are case sensitive. The Alvis language requires agent names to start with an uppercase letter.

3 Data types and parameters

The ABD language 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.

Table 1 contains selected basic Haskell's types recommended to be used in the ABD language. The most common composite data types in Haskell (and ABD) are *lists* and *tuples*. A *list* is a sequence

Table 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.

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. Examples of lists and tuples are presented in Listing 3. 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: Examples of lists and tuples with their types

To make the source code more readable, one can introduce a synonym for an existing type as shown in Listing 4.

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

Listing 4: Synonyms for composite data types

A new data type is defined using the `data` keyword (see Listing 5). The identifier after the `data` keyword is the name of the new type, while the identifier after the = sign is called *value constructor* (data constructor). A value constructor is a special function, which name, as the name of type, must start with an uppercase letter. Types placed after a value constructor name are called *components* of the type. A value constructor is used to create a new value of the corresponding type as shown in the second line of Listing 5. A value constructor name can be the same as the type one.

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

Listing 5: New composite data type

We can used more than one value constructor for one type. Such types in Haskell are called *algebraic data types*. Each value constructor is separated in the definition by the `|` sign. Each of an algebraic data type's value constructor can take zero or more arguments. An algebraic data type can be used to define an enumeration type or a type with different variants of data (see Listing 6).

Constants are defined using parameterless Haskell functions as shown in Listing 7.

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

Listing 6: Examples of Haskell algebraic data types

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

Listing 7: Examples of constants

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

Parameters are defined using Haskell syntax. The line starts with a parameter name, then the `::` symbol is placed followed by the parameter type. Examples of parameters definitions are shown in Listing 8. ABD language requires constants and parameters names to start with a lowercase letter.

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

Listing 8: Examples of parameters definitions

4 Communication with outside world

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

1) Connections to the port in the corresponding communication diagram (i.e. one-way or two-way connections);
2) Statements used in the code layer.

Moreover, any communication through the port can be a pure synchronisation or a single value (probably of a composed type) can be sent/collected. A *pure synchronisation* is a communication without sending values of parameters. A value passing communication not only synchronises two agents but also a parameter value is sent through the corresponding ports. ABD language requires port names to start with a lowercase letter.

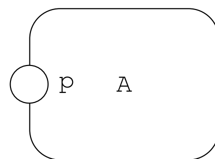

Figure 1: Agent `A` with port `p`

Figure 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.

Let us consider the code shown in Listing 9. The agent `A` collects a synchronisation signal through the port `p`, waits 1 second and sends a synchronisation signal through the same port. On the other

```
agent A {
  in p;
  delay 1000;
  out p;
}
```

Listing 9: Pure synchronisation

hand, the agent `A` presented in Listing 10, both synchronises with another agent and collects an interger value through the port `p`.

```
agent A {
  i :: Int;

  in p i;
  delay 1000;
  out p i;
}
```

Listing 10: Value passing communication

The `in` statement assigns the collected value to its parameter, while the `out` statement sends the value of its parameter. Instead of a parameter name, a constant can be used in the `out` statement.

A *guard* is an additional constraint that must be fulfilled before the corresponding statement is executed. Guards are logical expressions, written in Haskell, placed inside round brackets after the statement name. Both `in` and `out` statements may use guards, e.g.:

```
out (i > 1) p i;
```

When a guard is used in the `in` statement, then it can use only previously 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;
```

5 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 assign 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. The *exec* statement is the default one in the 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 also be followed by an expression. The ABD language uses Haskell to define and manipulate data types. Thus, such an expression takes the form of Haskell function call as shown below. Of course, the `exec` key word can be omitted in the first and the second example.

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

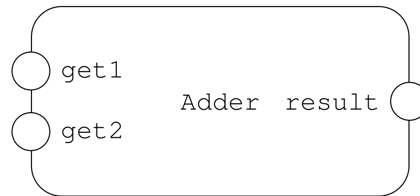

Figure 2: Agent *Adder*

```
agent Adder {
  x :: Int;
  y :: Int;

  in get1 x;
  in get2 y;
  y = x + y;
  out result y;
}
```

Listing 11: Agent `Adder`

Let us consider the agent `Adder` presented in Fig. 2 and Listing 11. The agent collects two integers through ports `get1` and `get2` and assigns them to parameters `x` and `y` respectively. Then, it sends the sum of those parameters through the port `result`. The agent performs those four statement and stops.

6 Recursion, loops and alternatives

Recursion is one of two mechanisms used for looping in the ABD 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 lowercase letter. The jump statement is composed of the `jump` key word and a label name (without a colon). If necessary, the `jump` key word may be followed by a guard placed inside round brackets. The `jump` statement is the key statement for translation algorithms from CCS to Alvis.

Let us consider the agent `Adder` shown in Fig. 2, but with the behaviour presented in Listing 12. This time the agent repeats its behaviour infinitely.

```
agent Adder {
  x :: Int;
  y :: Int;

go:
  in get1 x;
  in get2 y;
  y = x + y;
  out result y;
  jump go;
}
```

Listing 12: Agent `Adder` with infinite behaviour

The same behaviour can be described using the `loop` statement as shown in Listing 13.

Alvis also provide an `if else` statement (see Listing 14).

```
agent Adder {
  x :: Int;
  y :: Int;

  loop {
    in get1 x;
    in get2 y;
    y = x + y;
    out result y;
  }
}
```

Listing 13: Agent `Adder` with infinite behaviour – `loop` version

```
agent Adder {
  x :: Int;
  y :: Int;

  loop {
    in get1 x;
    in get2 y;
    if (x > y) { y = x - y; }
    else { y = x + y; }
    out result y;
  }
}
```

Listing 14: Agent `Adder` – `if else` statement

In order to allow for the description of agents whose behaviour may follow different alternative paths, the ABD language offers the `select` statement. The statement is similar to the basic `select` statement from Ada programming language, but there is no distinction between a server and a client.

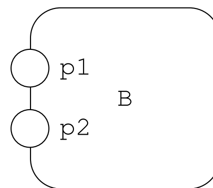

Figure 3: Agent `B`

```
agent B {
  x :: Int;
  y :: Int;

  loop {
    select {
      alt { in p1; x = x + 1; }
      alt { in p2; y = y + 1; }
    }
  }
}
```

Listing 15: Agent `B`

Let us consider the agent *B* presented in Fig. 3 and Listing 15. 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 the similar way the second alternative is performed. Each of the alternatives described with the `alt` statement is called a *branch*.

The selection between branches can depend on guards associated with each branch as shown in Listing 16. When an `alt` statement is to be executed, the first *open* branch is chosen. 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*.

```
agent B {
  x :: Int;
  y :: Int;

  loop {
    select {
      alt (x < 10) { in p1; x = x + 1; }
      alt (y < 20) { in p2; y = y + 1; }
    }
  }
}
```

Listing 16: Agent `B` – branches with guards

The `ready` function can be used as a guard (or its part). The function takes a list of ports names as its argument and returns `true` if all ports are ready to perform a communication (see Listing 17).

```
agent B {
  x :: Int;
  y :: Int;

  loop {
    select {
      alt (ready [p1]) { in p1; x = x + 1; }
      alt (ready [p2]) { in p2; y = y + 1; }
    }
  }
}
```

Listing 17: Agent `B` – branches with `ready` statement

```
agent B {
  x :: Int;
  y :: Int;

  loop {
    select {
      alt { in p1; x = x + 1; }
      alt { delay 2000; in p2; y = y + 1; }
    }
  }
}
```

Listing 18: Agent `B` – branche with time-out

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. The `delay` is also used to define time-outs. It is possible for one or more of the branches to start with a delay statement as shown in Listing 18. Such described agent `B` tries to communicate via port `p1`, but if the other agent is not ready for the communication, then it waits 2 seconds and tries to communicate via port `p2`. However, if the communication via port `p1` is established before the delay goes by, then the delay is cancelled. A delay branch can be guarded if necessary.

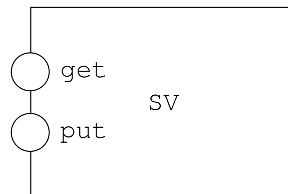

Figure 4: Passive agent `SV`

```
agent SV {
  sharedVariable :: Int;

  sharedVariable = 0;
  select {
    alt { in put sharedVariable; }
    alt { out get sharedVariable; }
  }
}
```

Listing 19: Passive agent `SV`

The `select` statement is also used for description services provided by passive agents. *Passive agents* do not perform any individual activity, and are similar to the Ada language protected objects (shared variables, see [2], [4]). Passive agents provide mechanism for the mutual exclusion and data synchronisation.

Let's consider a passive agent used to store a shared variable with two ports `get` and `put`, as shown in Fig. 4 and Listing 19. Each branch defines one service of the agent. All statements put before the `select` statement are treated as initial statements and are executed once at the beginning of the corresponding model executing.

7 Examples

This section presents a few simple examples of Alvis agents (two active and two passive ones) defined with the statements presented in this report. All those agents can be used as parts of Alvis models composed of a set of agents.

7.1 Clock

The agent `Clock` presented in Fig. 5 and Listing 20 sends a synchronisation signal via the port `clk` every 1 second. However, if the agent is connected with another one, it waits every time a signal is ready for the second agent readiness. It means that there is at least 1 second delay between sending two consecutive signals, but the agent can also wait for the synchronisation infinitely.

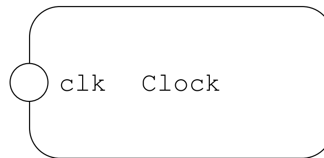

Figure 5: Agent `Clock`

```
agent Clock {
  loop {
    out clk;
    delay 1000;
  }
}
```

Listing 20: Agent `Clock`

7.2 Logic gate

The agent `And` presented in Fig. 6 and Listing 21 represents the *and* logic gate. The agent collects two Boolean argument through ports `arg1` and `arg2` respectively. Then, the conjunction of collected values is sent through port `and`. Another logic gates can be described in the similar way. Haskell built-in *and* operator is used this time to determine the result. Sometime, it is necessary to define an operator (e.g. xor) ourselves.

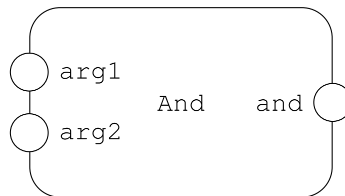

Figure 6: Agent `And` – logic gate

```
agent And {
  a :: Bool;
  b :: Bool;

  loop {
    in arg1 a;
    in arg2 b;
    a = a && b;
    out and a;
  }
}
```

Listing 21: Agent `And` – logic gate

7.3 Counter

The passive agent `Counter` presented in Fig. 7 and Listing 22 count the collected signals. Everytime the `in` statement is executed, the `n` parameter is increased. The second port (`get`) is used to read the current value of `n`.

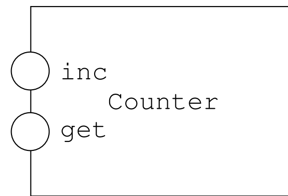

Figure 7: Passive agent `Counter`

```
agent Counter {
  n :: Int;

  n = 0;
  select {
    alt {
      critical { in inc; n = n + 1; }
    }
    alt { out get n; }
}
```

Listing 22: Passive agent `Counter`

The `critical` statement is used to guarantee that the agent cannot be postponed between executing the corresponding two statements.

7.4 Queue

The passive agent `Queue` presented in Fig. 8 and Listing 23 provides a FIFO queue of integers. The port `put` is used to include a new element at the end of the queue. The standard Haskell ++ operator is used to concatenate two lists. The second branch is open only if the queue is not empty. Haskell standard functions `head` and `tail` are used to determine the *head* and *tail* of the list (queue). Then, the head is sent through port `get`.

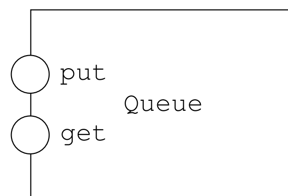

Figure 8: Passive agent `Queue`

8 Summary

Alvis is a new modelling language defined for embedded systems. Alvis combines a graphical modelling language (hierarchical communication diagrams) with Haskell based high level programming language. A survey of the basic statements of the Alvis internal language (ABD) has been presented in the report. The ABD language is used to describe behaviour of individual agents in Alvis models. The ABD language is based on Haskell, and uses Haskell for parameters manipulation. The report is not a complete description of the ABD language. Advanced statements used to rule agents behaviour (statements for scheduler or interrupt handling) have not been presented here. The report is just a starting point to learn Alvis.

```
agent Queue {
  q :: [Int];
  n :: Int;

  q = [];
  select {
    alt {
      critical {
        in put n;
        q = q ++ [n];
      }
    }
    alt (q /= []) {
      critical {
        n = head q;
        q = tail q;
        out get n;
      }
    }
  }
}
```

Listing 23: Passive agent `Queue`

References

[1] L. Aceto, A. Ingófsdóttir, K. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, Cambridge, UK, 2007.

[2] Ada Europe. *Ada Reference Manual ISO/IEC 8652:2007(E) Ed. 3*, 2007.

[3] K. Balicki and M. Szpyrka. Formal definition of XCCS modelling language. *Fundamenta Informaticae*, 93(1-3):1–15, 2009.

[4] J. Barnes. *Programming in Ada 2005*. Addison Wesley, 2006.

[5] C. Fencott. *Formal Methods for Concurrency*. International Thomson Computer Press, Boston, MA, USA, 1995.

[6] P. Matyasik. *Design and analysis of embedded systems with XCCS process algebra*. PhD thesis, AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Kraków, Poland, 2009.

[7] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[8] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Sebastopol, CA, USA, 2008.

[9] M. Szpyrka and P. Matyasik. Formal modelling and verification of concurrent systems with XCCS. In *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008)*, pages 454–458, Krakow, Poland, July 1-5 2008.