AUTOMATYKA • 2011 • Tom 15 • Zeszyt 2

Marcin Szpyrka*, Piotr Matyasik*, Rafał Mrówka*,
Wojciech Witalec*, Jarosław Baniewicz**, Leszek Kotulski*

Introduction to Modelling Embedded Systems with Alvis

1. Introduction

Alvis [18] is a novel modelling language being developed at AGH University of Science and Technology in Kraków, Department of Automatics. More information about the current status of the project can be found at the website http://fm.ia.agh.edu.pl. The main aim of the project is to provide a flexible modelling language for embedded systems with a possibility of models formal verification. Although Alvis has been designed for embedded systems, it can be used for modelling any system composed of concurrent subsystems.

An embedded system usually consists of a set of sensors cooperating with one or more decision units. The design of such a system complicates in respect of both a complex scheme of components interconnections and their concurrent execution. In practice, the latter one excludes testing as a way to guarantee an expected level of system quality. Thus, a formal verification of such systems is necessary. From this point of view, Alvis can be treated similarly to another formal methods like Petri nets [13, 9, 10, 5, 15], process algebras [6, 8, 12, 1] or time automata [2, 5]. However, due to specific mathematical syntax, these languages are very seldom used in real software projects. The aim of the Alvis language development is to provide a formal modelling language similar (from the syntax point of view) to the most popular languages used in industry. In contrast to formal methods mentioned above, Alvis provides a simple way to describe properties of a developed embedded system. Instead of designing the environment as a part of the model it is possible to specify signals generated or collected by the environment. This simplifies the model and reduces time need for a model development.

* AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, IT and Electronics, Department of Automatics
** Electronic Controls / E&S, Delphi Poland S.A., Kraków

The paper is organised as follows. Section 2 provides a short presentation of the Alvis modelling language. The presented concepts are illustrated with an example of an Automatic Train Protection system described in Section 3. Computer software supporting the design with Alvis is described in Section 4. The paper is summarised in the final section.

2. Alvis Language

Alvis is a successor of the XCCS language [3, 11], which is an extension of the CCS process algebra [12, 1]. In contrast to process algebras, Alvis uses a high level programming language based on the Haskell syntax, instead of algebraic equations. Moreover, it combines hierarchical graphical modelling with high level programming language statements. The key concept of Alvis is an *agent* that denotes any distinguished part of the system under consideration with a defined identity persisting in time. An Alvis model is a system of agents that usually run concurrently, communicate one with another, compete for shared resources etc. To describe all dependences among agents Alvis uses three model layers: graphical, code and system one.

2.1. Code layer

The *code layer* is used to define the behaviour of agents. Each agent is described with a piece of source code that may contain Alvis statements presented in Table 1. Moreover, Alvis uses the Haskell programming language [14] to define parameters, data types and data manipulation functions [16].

From the code layer point of view, agents are divided into *active* and *passive* ones. *Active agents* perform some activities and are similar to tasks in Ada programming language [4] (see Fig. 1 agents *ATS* and *Timer*). Each of them can be treated as a thread of control in a concurrent or distributed system. *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. They provide a set of procedures that may be called by other agents (see Fig. 1 agent *Console*).

2.2. Graphical Layer

The *graphical layer* takes the form of a communication diagram [18]. The layer is used to define interconnections (communication channels) among agents. A communication diagram is a hierarchical graph whose nodes may represent both kinds of agents (*active* or *passive*) and parts of the model from the lower level. The diagrams allow programmers to combine sets of agents into modules that are also represented as agents (called *hierarchical ones*). Active and hierarchical agents are drawn as rounded boxes while passive ones as rectangles. Hierarchical agents are indicated by black triangles. An agent can communicate with other agents through *ports* that are drawn as circles placed at the edges of the corresponding rounded box or rectangle. 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 communication diagram is shown in Figure 1.

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

Table 1

Alvis statements

Statement	Description
`cli`	Turns off the interrupts handlers.
`critical {...}`	Define a set of statements that must be executed as a single one.
`delay ms`	Delays an agent execution for a given number of milliseconds.
`exec x = e`	Evaluates the expression and assigns the result to the parameter; the *exec* keyword can be omitted.
`exit`	Terminates the agent that performs the statement.

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

`in p`
`in p x`
	Collects a signal via the port *p*.
	Collects a value via the port *p*
	and assigns it to the parameter *x*.

`jump label`	Transfers the control to the line of code identified with the *label*.
`jump far A`	Transfers the control to the agent *A*.

`loop(g){...}`
`loop(every ms)`
`{...}`
`loop{...}`
	Repeats execution of the contents while the guard if satisfied.
	Repeats execution every *ms* miliseconds.
	Infinite loop.

`null`	Empty statement.

`out p`
`out p x`
	Sends a signal via the port *p*.
	Sends a value of the parameter *x* via the port *p*; a literal value can be used instead of a parameter.

`proc (g) p {...}`	Defines the procedure for the port *p* of a passive agent (guard is optional).

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

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

A communication between an active and a passive agent can be initialised only by the former. If the active agent performs the suitable in (or out) statement it calls the corresponding procedure. Each procedure in Alvis uses only one either input or output parameter (or signal in case of parameterless communication).

2.3. System layer

From users point of view, the *system layer* is predefined and only graphical and code layers must be designed. It is strictly connected with the system architecture and the chosen operating system. Alvis provides a few different system layers. The most universal one is denote by α^0 and makes Alvis similar to other formal languages. If the layer is chosen, each active agent has access to its own processor and performs its statements as soon as possible. On the other hand the α^1 denotes, among other things, a single microprocessor environment that is characteristic for most of embedded systems. The system layer keeps information about the current state of the system and provides some functions that are useful for implementation of scheduling algorithms or for retrieving information about other agents states.

2.4. Environment specification

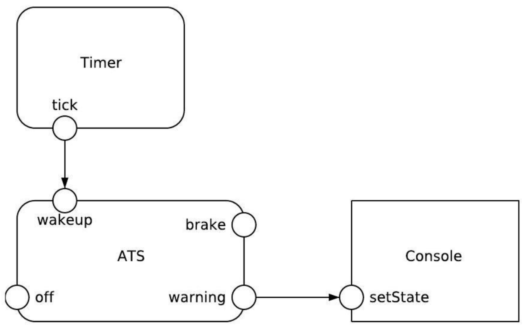
An embedded system is one that is a part of a larger one. It collects inputs that come from its environment (from sensors) and provides outputs that go to the environment (to controllers). To verify an embedded system formally we cannot separate it from its environment. If the Alvis language is chosen for an embedded system modelling, it is not necessary to design such an environment as a part of the model. It is sufficient to specify only signals/ values collected from the environment by the considered system or provided to it. Agents in the model may contain ports that are not used in any connection (so-called *border ports*) that are used for communication with the considered system environment. Properties of border ports are specified in the code layer preamble with the use of the environment statement. Each border port used as an input one is described with at least one *in* clause. Similarly, each border port used as an output one is described with at least one out clause. Each clause inside the *environment* statement contains the following pieces of information: *in* or *out* key word, the border port name, a type name or a list of permissible values to be sent through the port, a list of time points, when the port is accessible, optionally some modifiers: *durable*, *queue*, *signal*.

In the considered example (see Fig. 1) two border ports are used: *off* and *brake*. The former one is the input port that may provide a signal (without any specified value) every 1 s. The *signal* keyword means that this signal may but does not have to appear. In this model, it is used to simulate the possibility of turning the control signals off by the driver (see Section 3). The latter border port is the output one. A control signal (without any value – this is indicated by the first empty list) can be sent via the port at any time (this is indicated by the second empty list). In this model, the *brake* port is used to send a control signal to the brake that is treated here as a part of the environment. For more details about border ports specification see [17].

3. Automatic Train Stop System Model

Trains could not run safely without signalling devices. Some automatic systems are used to transfer signal directly to a driver cab. The driver must always obey the signal, but the possibility of human error can cause serious accidents.

a)



b)

```
environment {
  in off [] [1..] signal;
  out brake [] [];
}

agent Timer {
  loop (every 6000) {
    out tick;
  }
}
```

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

agent Console {
  state ::Int = 0;
  proc setState { in setState state ; }
}
```

Fig. 1. ATS model: a) communication diagram; b) code layer

Automatic Train Stop system (ATS system for short) is a kind of an Automatic Train Protection system used to guarantee safety of a train even if the driver is not capable of controlling the train. In the ATS system, a light signal is turned on every 60 seconds to check whether the driver controls the train. If the driver fails to acknowledge the signal within 6 seconds, a sound signal is turned on. Then, if the driver does not disactivate the signals within 3 seconds, using the acknowledge button, the emergency brakes are applied automatically to stop the train.

In the presented approach, the considered embedded system is composed of three agents:

1) *Timer* agent is used to wake up the system every 60 s. To meet the requirement, the *loop every* statement is used. In this example, 1 second is used as the basic time unit instead of the default 1 millisecond. As a result of this assumption, durations of steps execution can be disregarded.
2) *Console* agent represents the driver *console* used to display control signals. This is a passive agent with only one procedure used to set a new state of the console.

3) *ATS* agent represents the main control element of the system. Inside its main loop, the agent waits for a signal to be sent to its *wakeup* port. Then, it calls the *setState* procedure using the *warning* port and sends 1 (turn on light signal) as the parameter. The agent waits at most 6 s for a signal (interrupt) sent to the *off* port. After this time, it calls the procedure of the *Console* agent once again, but with another parameter, etc.

As it was already said, the model uses two border ports. The *off* port is used to collect signals that represent turning the system off by the driver. To reduce the state space (number of nodes in the corresponding LTS graph), we consider the signal at the level of details of 1 s.

Similarly to other formal methods, Alvis provides a possibility of formal verification of models. An Alvis model can be transformed into a *labelled transition system* (LTS). A LTS graph is an ordered graph with nodes representing states of the considered system and edges representing transitions among states. A state of a model is represented as a sequence of agents states. A state on an agent is four-tuple that contains: agent mode (e.g. running, waiting), its program counter (points out the current step – see comments in Fig. 1), context information list (contains additional information e.g. the name of called procedure) and a tuple with parameters values (for more details see [18]). For example, the initial state for the considered system is as follows:

```
ATS:(running,1,[],())
Timer:(running,1,[],())
Console:(waiting,0,[in(setState)],(0))
*:1/off
```

The agents *ATS* and *Timer* are running and are about to execute their first steps. The *Console* agent is waiting – the input *setState* procedure is accessible. The last line means that in 1 s the *off* signal may be received from the environment. The complete LTS-graph for this example can be found at http://fm.ia.agh.edu.pl/alvis:cases:ats.

4. Alvis Toolkit

As it was already said, Alvis is still under development. Nowadays, we focus on developing a computer software supporting the language. The current version of *Alvis Editor* – a tool to the design of Alvis models (see Fig. 2) can be downloaded from the project website (http://fm.ia.agh.edu.pl).

Alvis Editor is being developed in the Java language and is a free software covered by the GNU Library General Public License. It provides a full-featured graphical editor for the design of hierarchical communication diagrams and a code editor for the implementation of the code layer. Alvis Editor stores models in XML files.

Currently, we focus on a computer software for automatic LTS graphs generation for models. There are two approaches to this problem under implementation.

1) Alvis Editor XML file is read by *Alvis Translator* that builds a Java representation of the model. Then LTS graphs for individual agents are generated and joined into the final LTS graph.
2) Alvis Editor XML file is read by *Alvis Translator* and Haskell code for the model is generated. The generated source file contains code that represents the corresponding model and an algorithm for the LTS graph generation. Then, the Haskell file is compiled and executed. This approach enables users to extend the Haskell code with additional algorithms for the LTS analysis (e.g. selecting states that satisfy a given Boolean condition).

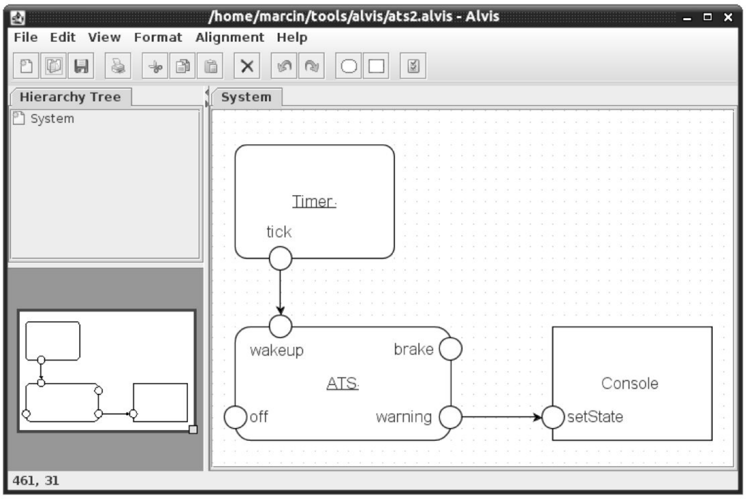


Fig. 2. Alvis Editor with the ATS system model

An LTS graph generated with any of the above approaches is used for the corresponding model verification. For example, after encoding such a graph using the *Binary Coded Graphs* (BCG) format, its properties are verified with the CADP toolbox [7]. CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking.

5. Conclusion

A short survey of the Alvis modelling language features has been presented in the paper. Alvis combines features of formal modelling languages with features of programming languages used in industry for embedded systems development. The main differences between Alvis and more classical formal methods, especially process algebras, are: the syntax that is more user-friendly from engineers point of view, and the visual modelling language (communication diagrams) that is used to define communication among agents. The main difference between Alvis and industry programming languages is a possibility of formal verification of Alvis models e.g. using model checking techniques.

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

References

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

[2] Alur R., Dill D., *A theory of timed automata*. Theoretical Computer Science, 126(2):183–235, 1994.

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

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

[5] Bengtsson J., Yi W., *Timed automata: Semantics, algorithms and tools*. Lecture Notes on Concurrency and Petri Nets, 3098, 2004.

[6] Bergstra J.A., Ponse A., Smolka S.A. (Eds). *Handbook of Process Algebra*. Elsevier Science, Upper Saddle River, NJ, USA, 2001.

[7] 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, pages 158–163, Berlin, Germany, 2007. Springer.

[8] Hoare C.A.R., *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[9] Jensen K., *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. volume 1–3. Springer-Verlag, Berlin, Germany, 1992–97.

[10] Jensen K., Kristensen L.M., *Coloured Petri nets. Modelling and Validation of Concurrent Systems*. Springer, Heidelberg, 2009.

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

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

[13] Murata T., *Petri nets: Properties, analysis and applications*. Proceedings of the IEEE, 77(4):541–580, 1989.

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

[15] Szpyrka M., *Petri nets for modelling and analysis of concurrent systems*. WNT, Warsaw 2008 (in Polish).

[16] Szpyrka M., *Alvis On-line Manual*. AGH University of Science and Technology, 2011.

[17] Szpyrka M., Kotulski L., Matyasik P., *Specification of embedded systems environment behaviour with Alvis modelling language*. In Proc. of the 2011 International Conference on Embedded Systems and Applica-tions ESA'11 (part of Worldcomp 2011), Las Vegas, Nevada, USA, 2011.

[18] Szpyrka M., Matyasik P., Mrówka R., *Alvis – modelling language for concurrent systems*. In P. Bouvry, Gonzalez-Velez H., Kołodziej J. (Eds), Intelligent Decision Systems in Large-Scale Distributed Environments, Studies in Computational Intelligence, vol. 362, Springer-Verlag, 2011, 315–342.