Introduction to Alvis modelling language

Abstract

Alvis is a novel modelling language designed for embedded systems. It combines both high level programming language used to define agents behaviour with hierarchical graphical modelling language used to define interconnections between agents. The paper presents a survey of the most important features of the language.

Keywords: Alvis language, formal verification, embedded systems.

Wprowadzenie do języka modelowania Alvis

Streszczenie

Alvis jest nowym językiem modelowania przeznaczonym do rozwiązywania systemów wbudowanych. Łączy w sobie cechy języków programowania wysokiego poziomu z hierarchicznym językiem modelowania połączeń między agentami. Podstawowym elementem języka Alvis są agenty, które mogą działać współbieżnie, komunikując się ze sobą, czy też współzawodnicząc o zasoby dzielone. Dynamika poszczególnych agentów jest opisywana w warstwie kodu używającej do tego celu języka programowania wysokiego poziomu (połączenia natywnych konstrukcji języka Alvis i języka funkcjonalnego Haskell). W warstwie graficznej definiowane są połączenia między agentami wskazujące, które agenty się ze sobą komunikują i jaki jest kierunek tej komunikacji. Warstwa ta ma postać grafu hierarchicznego, co pozwala rozwiązywać systema wbudowane metodą od ogółu do szczegółu lub odwrotnie. Formalną reprezentacją modelu w języku Alvis jest graf LTS (Labelled Transition System), który reprezentuje wszystkie osiągane stany i przejścia między nimi. Graf ten jest stosowany do formalnej weryfikacji modelu. Artykuł zawiera przegląd najistotniejszych cech języka Alvis.

Słowa kluczowe: język modelowania Alvis, formalna weryfikacja, systemy wbudowane.

1. Introduction

Alvis [7, 8] is a successor of the XCCS modelling language [2], which was an extension of the CCS process algebra [1, 4]. However, instead of algebraic equations, Alvis uses a high level programming language based on the Haskell syntax. Alvis combines hierarchical graphical modelling with high level programming language. A model consists of three layers. The graphical layer is used to define data and control flow among agents. The code layer is used to describe behaviour of individual agents. The third system layer is used for simulation and analysis purposes. Moreover, it allows a designer to choose a scheduling algorithm or to define the algorithm oneself.

Alvis, as well as other formal methods, like process algebras, Petri nets (e.g. Alvis is defined to be more flexible than RTCP-nets [6]) or time automata, and can be used for modelling concurrent systems, especially embedded ones.
An Alvis model may contain not only the application under consideration, but also selected elements of the operating system and the hardware that are essential from the considered system point of view.

The paper is organised as follows. Sections 2 and 3 present the graph and code layers, respectively. Section 4 deals with system layers. A short description of Alvis models and their states is presented in Section 6.

2. Communication diagrams

Communication diagrams are the visual part of the Alvis modelling language. They are used to describe an embedded system from the control and data flow point of view. Communication diagrams are the only way to point out agents that communicate with each other. A communication diagram is a hierarchical graph composed of a set of non-hierarchical parts called pages. Each page may contain active, passive and hierarchical agents. A hierarchical agent stands for a submodel that usually gives a more detailed description of the activity represented by the agent.

Active agents (drawn as rounded boxes) perform some activities and each of them can be treated as a thread of control in a concurrent system. Passive agents (drawn as rectangles) do not perform any individual activity. They are similar to shared variables and provide a mechanism for the mutual exclusion and data synchronisation. An agent can communicate with other agents through ports (drawn as circles). A connection (communication channel) is defined explicitly between two agents and connects two ports. Connections are drawn as lines. An arrowhead points out the input port for the particular connection. Connections without arrowheads represent pairs of connections with opposite directions. A connection between two active agents creates a synchronisation point between them. On the other hand, a connection between an active and a passive agent is similar to a procedure call.

Fig. 1. SBR - communication diagram
Rys. 1. SBR - diagram komunikacji

Let us consider a simple Sender-Buffer-Receiver system. The communication diagram for the system is shown in Fig. 1. The sender (agent S) places some information (integers in the considered example) into the buffer (agent B), while the receiver (agent R) collects the information.

3. Code layer

The code layer is used to describe the behaviour of individual agents in Alvis models. The layer uses Alvis behaviour description language and some elements of the Haskell functional programming language. In spite of the fact that Alvis has its origin in CCS [1, 4] and XCCS [2] process algebras, to make the language more convenient from the practical (engineering) point of view, algebraic equations and operators have been replaced with statements 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, and to specify some environment characteristics.

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.

The general structure of the code layer is as follows:

```haskell
-- Preamble:
- types
- constants
- functions
- environment specification
- Implementation:
agent AgentName;
declaration of parameters
agent body
```

The preamble contains definitions of types, constants and functions used to manipulate data in a model. This part of the preamble is encoded in pure Haskell. Moreover, the preamble may contain specification of some environment activities that may be useful e.g. for an Alvis model simulation. The implementation part contains definitions of the agents' behaviour. It is possible to share one definition among a few agents. In that case, a few agents' names are placed after the keyword agent separated by commas. If necessary, an agent name is followed by its priority put inside round brackets. Priorities range from 0 to 9. Zero is the higher system priority.

Alvis uses the Haskell's type system. Selected basic Haskell types recommended to be used in Alvis are: Char, Bool, Int, Double. The most common composite data types are lists and tuples. To make the source code more readable, one can introduce a synonym for an existing type using Haskell type keyword. Moreover, new data types can be defined using the data key word. Haskell supports also structures data type. For more details see for example [5].

Parameters are defined using 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:

```
Alvis provides a typical if else statement with optional elseif clauses and three types of loop statements. Moreover, the recursion mechanism can be used to define agents that repeat a sequence of statements. The recursion mechanism uses two language concepts: labels and the jump statement. It is useful for algorithms that translate CCS scripts into Alvis.

Some Alvis statements contain so-called guards. Guards are logical expressions, written in Haskell, placed inside round brackets. They are used for example, as conditions for the if else statement.

In order to allow for the description of agents whose behaviour may follow different alternative paths, Alvis offers the select statement. The statement may contain a series of alt clauses called branches. Each branch may be guarded. These guards divide branches into open and closed ones. A branch is called open, if it does not have a guard attached or its guard evaluates to True. Otherwise, a branch is called closed. To avoid indeterminism, if more than one branch is open the first of them is chosen to be
```
executed. If all branches are closed, the corresponding agent is postponed until at least one branch is open.

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.

Passive agents are used to store data shared among agents and to avoid the simultaneous use of such data by two or more agents. Each procedure has its own port attached and a communication with a passive agent via that port is treated as the corresponding procedure call. Depending on the communication direction, such a procedure may be used to send or collect some data from the passive agent. Each procedure is defined with the proc statement. The procedure is accessible only if its guard evaluates to True.

The set of all Alvis statements is given in Table 1. To simplify the syntax, the following symbols have been used. A stands for an agent name, p stands for a port name, x stands for a parameter, g, g1, g2,... stand for guards, and ms stands for milliseconds.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt (g) {...}</td>
<td>Defines a branch inside a select statement. The guard is optional.</td>
</tr>
<tr>
<td>cli</td>
<td>Turns off the interrupts handlers - interrupts will be ignored.</td>
</tr>
<tr>
<td>critical {...}</td>
<td>Define a set of statements that must be executed as a single one. It cannot be interrupted.</td>
</tr>
<tr>
<td>delay ms</td>
<td>Delays an agent execution for a given number of milliseconds.</td>
</tr>
<tr>
<td>exec x = expression</td>
<td>Evaluates the expression and assign the result to the parameter; the exec keyword can be omitted.</td>
</tr>
<tr>
<td>exit</td>
<td>Terminates the agent that performs the statement.</td>
</tr>
<tr>
<td>if (g) {...}</td>
<td>If the guard is satisfied the if part is executed, otherwise the else part is executed.</td>
</tr>
<tr>
<td>elseif (g2) {...}</td>
<td>Extended version of the conditional statement.</td>
</tr>
<tr>
<td>elseif (g3) {...}</td>
<td></td>
</tr>
<tr>
<td>else {...}</td>
<td></td>
</tr>
<tr>
<td>in p</td>
<td>Collects a signal (without value) via the port p.</td>
</tr>
<tr>
<td>in p x</td>
<td>Collects a value via the port p and assigns it to the parameter x.</td>
</tr>
<tr>
<td>jump label</td>
<td>Transfers the control to the line of code identified with the label.</td>
</tr>
<tr>
<td>jump far A</td>
<td>Transfers the control to the agent A.</td>
</tr>
<tr>
<td>loop (every ms) {...}</td>
<td>Repeats execution every ms milliseconds.</td>
</tr>
<tr>
<td>loop (g) {...}</td>
<td>Repeats execution of the contents while the guard if satisfied, the guard is checked every time before entering the loop contents. It is similar to the while loop in most languages.</td>
</tr>
<tr>
<td>null</td>
<td>Empty statement.</td>
</tr>
<tr>
<td>out p</td>
<td>Sends a signal (without value) via the port p.</td>
</tr>
<tr>
<td>out p x</td>
<td>Sends a value of the parameter x via the port p; a literal value can be used instead of a parameter.</td>
</tr>
<tr>
<td>proc (g) p {...}</td>
<td>Defines the procedure for the port p of a passive agent. The guard is optional.</td>
</tr>
<tr>
<td>select {}</td>
<td>Selects one of the alternative choices. Guards g1, g2, g3,... decide which alternatives can be chosen after entering the select statement.</td>
</tr>
<tr>
<td>=start A</td>
<td>Activates the agent A if it is in the Init state, otherwise do nothing.</td>
</tr>
<tr>
<td>sti</td>
<td>Turns on the interrupts handlers.</td>
</tr>
</tbody>
</table>

Let us consider the Sender-Buffer-Receiver system. The agents can be defined as follows:

agent S {
    loop {out push;}
}
tagent R {
    loop {in pop;}
}
tagent B {
    1 :: Int = 0;
    proc (i == 0) push {
        in push;
        i = 1;
    }
    proc (i == 1) pop {
        out pop;
        i = 0;
    }
}

4. System layer

The system layer is a predefined one. It is necessary for models simulation and analysis. From the users point of view, the layer works in the read-only mode. It gathers information about all agents in the model and their states.
The system layer provides some functions that are useful for the implementation scheduling algorithms or for retrieving information about other agents states.

A user can choose one of a few versions of the layer and it affects the model semantic. The system layer is strictly connected with the system architecture and the chosen operating system. System layers differ about the scheduling algorithm and system architecture mainly.

The most universal system layer is denoted by $\alpha^0$. It is based on the following assumptions: 1) Each active agent has access to its own processor and performs its statements as soon as possible; 2) The scheduler function is called after each statement automatically. 3) In case of conflicts, agents priorities are taken under consideration. If two or more agents with the same highest priority compete for the same resources, the system works indeterministically. A conflict is a state when two or more active agents try to call a procedure of the same passive agent or two or more active agents try to communicate with the same active agent.

5. Alvis model

An Alvis model is a triple that contains a communication diagram, a code layer and a system layer. A state of a model is represented as a sequence of agents' states, where an agent state contains the following pieces of information: agent mode (am) - finished, init, ready, running, waiting: program counter (pc); context information list (ci); parameters values tuple (pv).

A passive agent is always in one of two modes: waiting or taken. The former one means that the agent is inactive and waits for another agent to call one of its accessible procedures. In that situation the program counter is equal to zero and the context information list contains names of accessible procedures. In any state, the parameters values list contains the current values of the agent parameters. The taken mode means that one of the passive agent procedures has been called and the agent is executing it. In that case, $ci$ contains the name of the called procedure. The $pc$ points out the index of the current/next statement to be executed.

An active agent can be in any of the modes. If an active agent starts in the init mode, it is inactive until another agent activates it. Active agents that are initially activated are distinguished in the communication diagram - their names are underlined. If an agent is in the init mode, its $pc$ is equal to zero and $ci$ is empty. The finished mode means that an agent has finished its work or it has been terminated. In such a case ,its $pc$ is equal to zero and $ci$ is empty. The waiting mode means that an active agent is waiting either for a synchronous communication with another active agent, or for a currently inaccessible procedure of a passive agent. In such a case, the $pc$ points out the index of the current statement and $ci$ contains the names of the agent ports that can be used for the desired communication. The next mode ready means that an agent is ready to perform the next statement (pointed out by $pc$), but it is waiting for process switching. In this mode, $ci$ may contain the names of the agent ports, if the agent was waiting previously, or it may contain the names of called passive agents, if the agent has called a procedure that has been interrupted. The last mode running means that the agent is performing one of its statements. If it is a synchronous communication with another active agent or a procedure call, then the used port name and the other agent name are placed into $ci$. If within the procedure, the passive agent calls a procedure of another passive agent, the used port name and the second agent name are included into $ci$ of the passive agent.

For example, the initial state for the Sender-Buffer-Receiver system can be represented as follows:

0:  
S: {running,1,{},1}  
R: {running,1,{},0}  
B: {waiting,0,[{in(push)}],0}

Such a state is a node in an LTS graph generated for any Alvis model. An LTS graph contains a node for each reachable state and an edge for each transition between states. Such a graph not only provides a semantic for a model, but also is used for the verification purposes. To verify a model formally, an LTS is generated and encoded in the BCG format. Then, the CADP Evaluator [3] is used to check whether the generated LTS satisfies its requirements e.g. defined using $\mu$-calculus formulas.

6. Summary

A short introduction to the Alvis modelling language has been given in the paper. Defined for the embedded systems design, Alvis seems to be more accessible for engineers than classical formal methods. Alvis should be treated like other formal methods, especially process algebras. The main differences between Alvis and these languages are: the syntax that is more user-friendly from the programmers point of view, and the visual modelling language (communication diagrams) used to define connections between agents. The language is still under development. For more information about the current status of the project visit http://fm.ia.agh.edu.pl.

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

7. References