INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2012, VOL. 58, NO. 3, PP. 247–254
Manuscript received February 14, 2012; revised March, 2012. DOI: 10.2478/v10177-012-0034-2

Communication with Environment in Alvis Models

Marcin Szpyrka, Piotr Matyasik, Rafał Mrówka, and Leszek Kotulski

***Abstract*—Alvis is a modelling language defined for the design and a formal verification of embedded systems. An Alvis model is a system of agents that usually run concurrently, communicate one with another, compete for shared resources etc. Due to the fact that an embedded system usually collects inputs that come from its environment and provides outputs that go to the environment it is necessary to provide a mechanism to describe such a communication. In contrast to another formal languages used to model embedded systems it is not necessary, using Alvis, to design such an environment as a part of a model. The paper deals with the problem of modelling a communication with an embedded system environment with Alvis.**

***Keywords*—Alvis modelling language, embedded systems, formal verification, communication with environment.**

I. Introduction

ALVIS[1] [1], [2] is a novel modelling language designed for real-time systems, especially for embedded ones. The main goal of the Alvis project was to strike a happy medium between formal and practical modelling languages. Formal methods like Petri nets [3], [4], [5], time automata [6], process algebra [7] etc. are very seldom used in real IT projects due to their specific mathematical syntax. The Alvis syntax seems to be more user-friendly. From programmers point of view, it is necessary to design two layers of an Alvis model. The code layer uses Alvis statements supported by the Haskell functional programming language to define a behaviour of individual agents. The graphical layer (communication diagram) is used to define communication channels among agents. The layer takes the form of a hierarchical graph, that allows designers to combine sets of agents into modules that are also represented as agents (called *hierarchical ones*). The Alvis language is supported by Alvis Toolkit software that, among other things, provides Alvis Editor used for developing Alvis models and Alvis Translator used to generate LTS graphs (Labelled Transition System). Such an LTS graph is a formal representation of an Alvis model can be formally verified e.g. with the help of the CADP toolbox [8].

An embedded system is one that is a part of a larger one. It is surrounded by other parts of the larger system that constitute the embedded system environment. Such an embedded system collects inputs that come from its environment (from sensors) and provide outputs that go to the environment (to controllers). To verify an embedded system formally we cannot separate it from its environment. Thus, if a formal language is used e.g. Petri nets, time automata, process algebra etc., an embedded system model must include both the system and its environment. As a result of such a situation a model is often significantly more complex and the state explosion problem makes a formal verification difficult or even impossible. One of the main Alvis advantages is the possibility of a flexible specification of a behaviour of an embedded system's environment. Instead of describing an environment as a part of an Alvis model, it is possible to specify signals/values sent or received by the environment.

The paper focuses at the modelling of a communication with an embedded system environment in Alvis models. It is organised as follows. Section II provides a short presentation of the Alvis modelling language. Border ports used for a communication with an embedded system environment are described in Section III. A formal description of Alvis models is given in Section IV. Section V deals with methods of communication with environment in Alvis. The paper is summarised in the final section.

II. Alvis Language

Alvis is a successor of the XCCS modelling language [9], [10], which was an extension of the CCS process algebra [11], [7]. An Alvis model is composed of three 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 and supports both *top-down* and *bottom-up* approaches to systems development. Graphical items used in a communication diagram design are shown in Fig. 1.
- *Code layer* – is used to describe the behaviour of individual agents. It uses both Haskell functional programming language [12] and original Alvis statements [1], [2]. The set of Alvis statements is given in Table I.
- *System layer* – depends on the model running environment i.e. the hardware and/or operating system. The layer is the predefined one. The system layer is used for a simulation and verification purposes.

Agents in Alvis are divided into three groups: *active, passive* and *hierarhical agents*. *Active agents* (see Fig. 1 agent A) perform some activities and are similar to tasks in the Ada programming language [13]. Each of them can be treated as a thread of control in a concurrent system. *Passive agents* (agent B) do not perform any individual activity, and are similar to protected objects (shared variables). Hierarchical agents (agent C) represent submodels (modules).

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. A *communication channel* is defined explicitly between two agents and connects two ports. Communication channels are drawn as lines. An arrowhead points out the input port for the particular connection

M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski are with AGH University of Science and Technology, Department of Applied Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland (e-mails: {mszpyrka; ptm; Rafal.Mrowka; kotulski}@agh.edu.pl).

[1]Project web site: `http://fm.ia.agh.edu.pl`

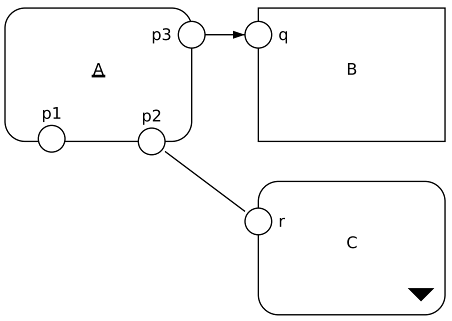


Fig. 1. Elements of Alvis communication diagrams.

TABLE I
ALVIS STATEMENTS

Statement	Description
`cli`	Turns off border ports.
`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 miliseconds.
`exec x = e`	Evaluates the expression and assigns the result to the parameter; the `exec` keyword can be omitted.
`exit`	Terminates an agent or a procedure.
`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 loop contents while the guard if satisfied.. Repeats execution of the loop contents 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. The 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 does nothing.
`sti`	Turns on border ports.

(connection $(A.p3, B.q)$). Communication channels without arrowheads represent pairs of connections with opposite directions (connection between ports $A.p2$ and $C.r$).

The code layer is used to define data types used in the model under consideration, functions for data manipulation and the behaviour of individual agents. The layer uses the Haskell functional language (e.g. the Haskell type system) and original Alvis statements. The set of Alvis statements is given in Table I. 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 (Boolean conditions), `e` stands for an expression and `ms` stands for milliseconds.

Each non-hierarchical agent placed in a communication diagram must be defined in the corresponding code layer and vice-versa. Besides the statements presented in Table I, Alvis provides the *environment* statement that can be used only in a code layer preamble. This statement is used to specify behaviour of border ports (see Section III).

The system layer considered in the paper is denoted by α^1. The α^1 system layer has been worked out to make Alvis suitable for the modelling of single-processor embedded systems. Frankly speaking, α^1 stands for a set of system layers that differ about the scheduling algorithm. The α^1 layers are based on the following assumptions:

- All active agents share the same processor.
- The predefined α^1 scheduler function is called after each statement automatically and makes agents running as soon as possible.

III. BORDER PORTS

Border ports can be used both for collecting or sending some information to the embedded system environment. They cannot be connected with any other ports. Properties of border ports are specified in the code layer preamble with the use of the *environment* statement. It is possible to specify (in the code layer) all details of signals that a system collects or sends through any border port. Such ports must have unique names. The same name of a border port used twice means that two agents use the same border port.

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 [14]:

- *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.*

If a border port is used both as an input and output one, then it must be described both with the *in* and *out* clauses. If different kinds of signals can be sent through a border port, then more than one *in* or *out* clause must be used. If a border port is used for a parameterless communication, then the first list is empty. Similarly, if a border port is always accessible, then the second list is empty. Lists are defined using the Haskell language. In particular, it is possible to use infinite lists [12].

It should be underlined that only the *signal* modifier should be used in the final model of an embedded system. Other modifiers are defined mainly for the verification purposes, if reduced models are considered.

Let us focus on the description of input border ports presented in Fig. 2. Signals' directions are considered from an embedded system point of view, thus all considered ports are used to send information from an environment to the

```
in a [1..4] [];
in b [1..4] (map (100*) [1..]);
in c [1..4] (map (100*) [1..]) signal;
in d [1..4] (map (100*) [1..]) durable;
in e [1..4] (map (100*) [1..]) queue;
in f [1..4] (map (100*) [1..]) signal durable;
in g [1..4] (map (100*) [1..]) signal queue;
```

Fig. 2. Examples of input border ports' specification.

corresponding embedded system. In each case, one of the values $1, 2, 3, 4$ (at random) can be collected through a port. However, the ports differ about the way the values are used.

- `a` A value from the port can be collected at any time point. An agent that performs the *in* statement receives the value immediately (never waits for it). Such border ports are useful for a modelling of input sensors whose values can be read at any time.
- `b` Every 100 time-unit (by default milliseconds) a value is provided by the environment via the port. If none agent waits for it (*waiting* mode), the value is lost.
- `c` The port behaves similarly to the `b` one, but the signal may not be provided.
- `d` Every 100 time-unit a value is provided by the environment via the port. The value is accessible for the corresponding embedded system until an agent collects it. If while waiting for a collecting the value, another one is sent via the port, the previous one is overwritten.
- `e` The port behaves similarly to the `d` one, but if while waiting for a collecting the value, another one is sent via the port, it is put into a FIFO queue.
- `f` The port behaves similarly to the `c` one, but the value is accessible for the corresponding embedded system until an agent collects it or it is overwritten.
- `g` The port behaves similarly to the `f` one, but the values are put into a FIFO queue.

The specifications of ports $b, \ldots, g$ use the Haskell `map` function and an infinite list. For more details (if necessary) see [12].

```
out a [1..4] [];
out b [1..4] (map (100*) [1..]);
out c [1..4] (map (100*) [1..]) signal;
out d [1..4] (map (100*) [1..]) durable;
out e [1..4] (map (100*) [1..]) queue;
out f [1..4] (map (100*) [1..]) signal durable;
out g [1..4] (map (100*) [1..]) signal queue;
```

Fig. 3. Examples of output border ports' specification.

Let us focus on the description of output border ports presented in Listing 3.

- `a` Any of the values $1, 2, 3, 4$ can be sent through the port at any time point. An agent that performs the *out* statement sends the value immediately (never waits for the port accessibility).
- `b` Any of the values $1, 2, 3, 4$ can be sent through the port every 100 time-units, but if the system is not ready to send a value then the opportunity is lost.
- `c` The port behaves similarly to the `b` one, but the accessibility of the port is not guaranteed.
- `d` Any of the values $1, 2, 3, 4$ can be sent through the port every 100 time-units, but if the system is not ready to send a value then the environment waits for it.
- `e` The port behaves similarly to the `d` one, but the opportunities are put into a FIFO queue.
- `f` The port behaves similarly to the `d` one, but the accessibility of the port is not guaranteed.
- `g` The port behaves similarly to the `e` one, but the accessibility of the port is not guaranteed.

If a border port is used both as an input and an output one, then it must be described both with the *in* and *out* clauses. If different kinds of signals can be sent through a border port, then more than one *in* or *out* clause can be used, but the time points lists must be pairwise disjoint.

IV. Formal Model Description

A hierarchical communication diagram can be transformed into a non-hierarhical one with the *analysis* operation [15]. Thus, from the formal description point of view it is necessary to consider models with non-hierarhical communication diagrams only.

Let $\mathcal{P}(X)$ denote the set of ports of an agent X. We can distinguish the following subsets of the set $\mathcal{P}(X)$:

- $\mathcal{P}_{border}(X)$ denotes the set of *border ports* of agent X i.e. ports that are specified in the *environment* statement.
- $\mathcal{P}_{internal}(X) = \mathcal{P}(X) - \mathcal{P}_{border}(X)$ denotes the set of *internal ports* of agent X.
- $\mathcal{P}_{in}(X)$ denotes the set of *input ports* of agent X. An input border port is a border port with at least one *in* specification. An input internal port is an internal port with at least one one-way connection leading to this port or with at least one two-way connection.
- $\mathcal{P}_{out}(X)$ denotes the set of *output ports* of agent X. An output border port is a border port with at least one *out* specification. An output internal port is an internal port with at least one one-way connection leading from this port or with at least one two-way connection.
- $\mathcal{P}_{unc}(X) = \mathcal{P}_{internal}(X) - (\mathcal{P}_{in}(X) \cup \mathcal{P}_{out}(X))$ denotes the set of *unconnected ports*.
- $\mathcal{P}_{proc}(X) \subseteq \mathcal{P}_{internal}(X)$ denotes the set of procedure ports of agent X (for passive agents only) i.e. ports with defined the *proc* statement (names of such ports are treated as names of procedures).

For a set of agents W we define sets: $\mathcal{P}(W) = \sum_{X \in W} \mathcal{P}(X)$, $\mathcal{P}_{in}(W) = \sum_{X \in W} \mathcal{P}_{in}(X)$, etc. Moreover, let $\mathcal{P}$ denote the set of all model ports, $\mathcal{P}_{in}$ denote the set of all model input ports, etc.

Definition 1: A *Non-hierarchical communication diagram* is a triple $D = (\mathcal{A}, \mathcal{C}, \sigma)$, where:

- $\mathcal{A} = \{X_1, \ldots, X_n\}$ is the set of *agents* consisting of two disjoint sets, $\mathcal{A}_A$, $\mathcal{A}_P$ such that $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$, containing *active* and *passive* agents respectively.

- $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ is the *communication relation*, such that:

$$\forall X \in \mathcal{A} \colon (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C} = \emptyset, \quad (1)$$
$$((\mathcal{P}_{border} \times \mathcal{P}) \cup (\mathcal{P} \times \mathcal{P}_{border})) \cap \mathcal{C} = \emptyset, \quad (2)$$
$$\mathcal{P}_{proc} \cap \mathcal{P}_{in} \cap \mathcal{P}_{out} = \emptyset, \quad (3)$$
$$(p,q) \in (\mathcal{P}(\mathcal{A}_A) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow q \in \mathcal{P}_{proc}, \quad (4)$$
$$(p,q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_A)) \cap \mathcal{C} \Rightarrow p \in \mathcal{P}_{proc}, \quad (5)$$
$$\begin{aligned}&(p,q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow \\ &\Rightarrow (p \in \mathcal{P}_{proc} \wedge q \notin \mathcal{P}_{proc}) \vee \\ &\vee (q \in \mathcal{P}_{proc} \wedge p \notin \mathcal{P}_{proc}).\end{aligned} \quad (6)$$

- $\sigma \colon \mathcal{A}_A \to \{False, True\}$ is the *start function* that points out initially activated agents.

Each element belonging to $\mathcal{C}$ is called a *connection* or a *communication channel*. The restrictions from Definition 1 have the following meaning. (1) – A connection cannot be defined between ports of the same agent. (2) – Border ports cannot be connected with any other ports. (3) – Procedure ports are either input or output ones. (4), (5) – A connection between an active and a passive agent must be a procedure call. From conditions (3)–(5) it follows that any connection with a passive agent must be one-way connection. (6) – A connection between two passive agents must be a procedure call from a non-procedure port.

The start function σ makes possible delaying activation of some agents. Names of agents that are initially activated are underlined in a communication diagram.

Definition 2: A flat Alvis *model* is a triple $\overline{\mathbf{A}} = (D, B, \varphi)$, where:

- $D = (\mathcal{A}, \mathcal{C}, \sigma)$ is a *non-hierarchical communication diagram*,
- B is a syntactically correct *code layer*,
- φ is a *system layer*.

Moreover, each agent X belonging to the diagram D must be defined in the code layer, and each agent defined in the code layer must belong to the diagram.

It should be underlined that a syntactically correct code layer means also that only input ports may be used as arguments of *in* statements, and only output ports may be used as arguments of *out* statements. From now on, we will consider only $\overline{\mathbf{A}} = (D, B, \alpha^1)$ models.

Definition 3: A *state of an agent* X is a tuple

$$S(X) = (am(X), pc(X), ci(X), pv(X)), \quad (7)$$

where $am(X)$, $pc(X)$, $ci(X)$ and $pv(X)$ denote mode, program counter, context information list and parameters values of the agent X respectively.

All possible modes and transitions among them are shown in Fig. 4. *Finished* means that an agent has finished its work. *Init* is the default mode for agents that are inactive in the initial state. An agent can be activated by another one with the *start* statement. *Running* means that an agent is performing one of its statements. *Ready* means that an agent is ready to execute the next statement, but it waits for an access the processor. *Taken* means that one of the passive agent procedures has been called and the agent is executing it. For passive agents, *waiting* means that the corresponding agent is inactive and waits for another agent to call one of its accessible procedures. For active agents, the mode means that the corresponding agent is waiting either for a communication with another active agent, or for a currently inaccessible procedure of a passive agent.

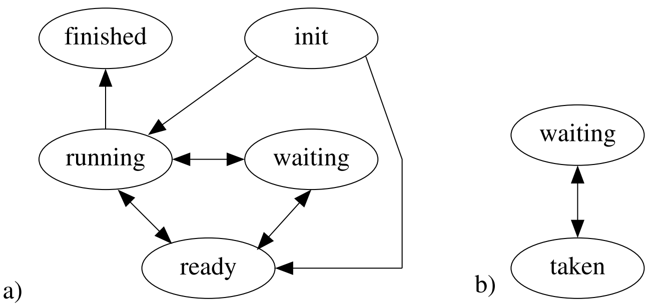


Fig. 4. Possible transitions among modes: a) active, b) passive agents.

TABLE II
RELATIONSHIPS BETWEEN THE MODE AND PROGRAM COUNTER

$am(X)$	$pc(X)$
finished	0
init	0
ready	current statement
running	current statement
taken	current statement of the called procedure
waiting (active agent)	current statement
waiting (passive agent)	0

The program counter points out the current statement of an agent i.e. the next statement to be executed or the statement that has been executed by an agent but needs a feedback from another agent to be completed (e.g. a communication between two active agents). Relationships between the mode and the program counter of an agent are shown in Table II.

The context information list contains additional information about the current state of an agent e.g. if an agent is the *waiting* mode, ci contains information about events the agent is waiting for. Possible entries put into ci lists are given in Table III. If an agent is in the *init* or *finished* mode, its context information list is empty.

If an Alvis model uses border ports then an extra agent denoted by $*$ and representing the system environment is considered. The state of the environment is described using its ci list only. The list contains information about time moments of the border ports accessibility.

Definition 4: A *state* of a model $\overline{\mathbf{A}} = (D, B, \alpha^1)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ and $\mathcal{A} = \{X_1, \dots, X_n\}$ is a tuple

$$S = (S(X_1), \dots, S(X_n), S(*)). \quad (8)$$

The $S(*)$ is omitted, if the considered model does not contain border ports.

Let us focus on the $ci(*)$ list. For each *in* and *out* clause from the *environment* statement, the list contains an entry with information about the next port accessibility with respect to

[1]We will use two notations to denote ports. A single lower-case letter e.g. p denotes a port p of some agent. If it is necessary to point out both a port name and agent name, the dot notation will be used e.g. $X.p$.

TABLE III
RELATIONSHIPS BETWEEN THE MODE AND THE CONTEXT INFORMATION LIST OF AN AGENT

agent X	$am(X)$	$ci(X)$ entry	description
active	running/waiting	$critical$	X is inside a critical section
active	ready/running/ waiting	$proc(Y.b, a)$	X has called the $Y.b$ procedure via port a and this procedure is being executed in the X agent context
		$timeout(s)$	a timer signal for the statement number s has been generated and waits for serving
		$timer(s, n)$	a timer signal for the statement number s will be generated in n time-units
		off	border ports handling turned off
		$in(*.a)$	a signal generated by the system environment has been provided via border port a
		$in(*.a \mid v)$	a value v generated by the system environment has been provided via border port a
		$out(*.a)$	the system environment is ready to collect a signal via border port a
		$out(*.a \mid i)$	the system environment is ready to collect via port a a value defined by i-th *out* clause for the port
active	ready/waiting	$in(a)$, $in(a \mid T)$	X waits for a communication via port $X.a$ ($X.a$ is the input port for this communication); T is the type of the expected value
		$out(a)$, $out(a \mid T)$	X waits for a communication via port $X.a$ ($X.a$ is the output port for this communication)
		$guard$	X waits for an open branch of a *select* statement
passive	taken	$proc(Y.b, a)$	X has called the $Y.b$ procedure via port a and this procedure is being executed in the same context as the X procedure
		$guard$	X waits for an open branch of a *select* statement
		$critical$	X is inside a critical section
		$timeout(s)$	a timer signal for the statement number s has been generated and waits for serving
		$timer(s, n)$	a timer signal for the statement number s will be generated in n time-units
passive	waiting	$in(a)$	input procedure $X.a$ is accessible
		$out(a)$	output procedure $X.a$ is accessible
passive	taken/waiting	*off*	border ports handling turned off
		$in(*.a)$	a signal generated by the system environment has been provided via border port a
		$in(*.a \mid v)$	a value v generated by the system environment has been provided via border port a
		$out(*.a)$	the system environment is ready to collect a signal via border port a
		$out(*.a \mid T)$	the system environment is ready to collect a value of type T via border port a
$*$		$in(a, n)$	a signal via input border port a will be accessible in n time units
		$in(a \mid i, n)$	a value defined by i-th *in* clause for port a will be accessible in n time units (if $i = 1$ then i is omitted)
		$out(a, n)$	output border port a will be accessible in n time units to collect a signal
		$out(a \mid i, n)$	output border port a will be accessible in n time units to collect a value defined by i-th *out* clause for the port

the clause. There are two exceptions, when such an entry is missing:

1) There is no the next time point when the corresponding port is accessible.
2) The time points list for the entry is empty – the corresponding port is always accessible with respect to the clause.

Suppose, the *environment* statement contains the following specification for port a:

```
in a [] [1,3,6]
```

Thus, in the initial state, $ci(*)$ contains the following entry $in(a, 1)$ i.e. a signal will be accessible via port a in one time-unit (one millisecond by default). One time-unit later, this entry will be replaced with $in(a, 2)$. Then, 1 time unit later the entry will be replaced with $in(a, 1)$, and next with $in(a, 3)$. After 6 time-units from the beginning, $ci(*)$ will not contain any entry assigned to the considered clause.

If a border port is specified with more than one *in* clause then the clauses distinguished by their indices. Let a border port a be specified as follows:

```
in a [] [1,3,6]
in a Bool [2,4,7,8,9]
```

Thus, in the initial state, $ci(*)$ contains two entries $in(a|1, 1)$, $in(a|2, 2)$. One time-unit later, these entries will be replaced with $in(a|1, 2)$, $in(a|2, 1)$, etc. Output border ports are represented in $ci(*)$ in similar way.

To determine the current mode of an agent that is ready to execute its next statement, the scheduler function α^1 is used. For any state S, the function is applied to a set of active agents $\mathcal{A}'_A$ and satisfies the following requirements:

- $\forall X \in \mathcal{A}'_A, \alpha^1_S(X) \in \{ready, running\}$,
- if $critical \in ci(X)$ then $\alpha^1_S(X) = running$,
- $|\{X \in \mathcal{A}'_A \colon \alpha^1_S(X) = running\}| \leq 1$.

Definition 5: The *initial state* of a model $\overline{\mathbf{A}} = (D, B, \alpha^1)$ is a tuple S_0 as given in (8), where:

- $am(X) = \alpha^1_{S_0}(X)$ for any active agent X such that $\sigma(X) = True$;
- $am(X) = init$ for any active agent X such that $\sigma(X) = False$;
- $am(X) = waiting$ for any passive agent X;
- $pc(X) = 1$ for any active agent X in the $ready$ or $running$ mode and $pc(X) = 0$ for other agents.
- $ci(X) = [\,]$ for any active agent X;
- For any passive agent X, $ci(X)$ contains names of all accessible ports of X (i.e. names of all accessible procedures) together with the direction of parameters transfer, e.g. $in(a)$, $out(b)$, etc.
- For any agent X, $pv(X)$ contains X parameters with their initial values.

- For any *in* and *out* clause from the *environment* statement a suitable entry (if necessary) is put into $ci(*)$.

We consider behaviour of Alvis models at the level of detail of single steps. More statements e.g. $exec$, $exit$, in, etc. are *single-step* statements. On the other hand, if, $loop$ and $select$ are *multi-step* statements. We use recursion to count the number of steps for multi-step statements. For each of these statements, the first step enters the statement interior. Then, we count steps of statements put inside curly brackets. Steps performed by a model are described using the *transition* idea. The set of all possible transitions for Alvis models considered in the paper is given in Table IV. Transitions with numbers range from 1 to 15 are called *system transitions* and transitions with numbers range from 16 to 18 are called *environment transitions*.

TABLE IV
SET OF TRANSITIONS

	Symbol	Description
1	t_{cli}	performs a cli statement
2	$t_{critical}$	enters an critical statement
3	t_{delay}	performs a delay statement
4	t_{exec}	performs an evaluation and assignment
5	t_{exit}	terminates an agent or a procedure
6	t_{if}	enters an if statement
7	t_{in}	performs communication (input side)
8	t_{jump}	jumps to a label
9	t_{loop}	enters a *while* or *infinite* loop
10	$t_{loopevery}$	enters a *loop every* loop
11	t_{null}	performs an empty statement
12	t_{out}	performs communication (output side)
13	t_{select}	enters a select statement
14	t_{start}	starts an inactive agent
15	t_{sti}	performs a sti statement
16	t_{*in}	activates an input border port
17	t_{*out}	activates an output border port
18	t_{*time}	denotes time passage

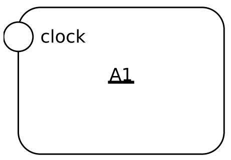
Executing of each of presented transitions changes the current state of a model. It is out of the scope of the paper to describe each of the transitions in details. We focus on these transitions that are connected with a communication with environment. More details about the transition idea in Alvis models can be found in [15]. However, the paper describes models with α^0 system layer i.e. with unlimited number of processors, and a system environment is not taken under consideration.

V. COMMUNICATION WITH ENVIRONMENT

The fact that a transition t is enabled in a state S with respect to an agent X and that a state S' is the result of executing t in S will be denoted by $S{-}t(X){\rightarrow}S'$. Sometimes, an extended version of this notation will be used:

- $S{-}t_{start}(X,Y){\rightarrow}S'$, where Y is the argument of the corresponding *start* statement;
- $S{-}t_{in}(X.p,T){\rightarrow}S'$, $S{-}t_{out}(X.p,T){\rightarrow}S'$, where $X.p$ is the port used for the communication and T is the type of sent/collected value. If necessary, the special *Empty* type will be used to denote a valueless communication.

Let us start with signals that are generated by an embedded system periodically. Suppose an interrupt signal (without) any specified value) is provided to the port $A1.clock$ every 10 ms (see Fig. 5).



```
environment {
  in clock [] (map (10*) [1..]);
}

agent A1 {
  n :: Int = 0;                          -- step no
  loop {                                       -- 1
    in clock;                                  -- 2
    n = n + 1;                                 -- 3
  }
}
```

Fig. 5. Collecting periodical signals from environment.

The model presented in Fig. 5 contains only one agent that is initially activated. The set $\mathcal{T}$ of all model transitions contains 4 elements:

- t_{loop} – enters the main loop interior,
- t_{in} – collects a signal from port $clock$,
- t_{exec} – assigns a new value to parameter n,
- t_{*in} – provides a signal from environment via port $clock$.

Let $\Delta(X_i,k)$ denote the duration of the k-th step execution for the agent X_i. Suppose, $\Delta(A1,k) = 1$ for any system transition in this model.

The first signal generated by the environment appears in the 10th millisecond. Agent $A1$ enters the loop, executes the in statement and waits for the signal. Finally, it receives the signal, increases its parameter n of 1 and repeats its behaviour. It should be underlined, that the agent must be ready to collect the signal, otherwise the signal is lost. In this example, the time taken by the three steps the agent realises (entering the loop, executing the *in* statement and executing the *exec* statement) is significantly less than 10 ms, thus $A1$ is always ready to handle the signal.

The initial state is defined as follows:
$S_0 = ((running, 1, [\,], (0)), [in(clock, 10)])$
Next, $S_0{-}t^1_{loop}(A1){\rightarrow}S_1$, where $t^1_{loop}(A1)$ denotes that $\Delta(A1,1) = 1$ (1 is the step number for the *loop* statement):
$S_1 = ((running, 2, [\,], (0)), [in(clock, 9)])$.

For a pair of states S, S' we say that S' is *directly reachable* from S iff there exists $t \in \mathcal{T}$ such that $S{-}t{\rightarrow}S'$. We say that S' is *reachable* from S iff there exists a sequence of states $S^1, \ldots, S^{k+1}$ and a sequence of transitions $t^1, \ldots, t^k \in \mathcal{T}$ such that $S = S^1{-}t^1{\rightarrow}S^2{-}t^2{\rightarrow}\ldots{-}t^k{\rightarrow}S^{k+1} = S'$. The set of all states that are reachable from the initial state S_0 is denoted by $\mathcal{R}(S_0)$.

States of an Alvis model and transitions among them are represented using a labelled transition system (LST graph for short). An LTS *graph* is a directed graph LTS $= (V, E, L)$, such that $V = \mathcal{R}(S_0)$, $L = \mathcal{T}$, and $E = \{(S,t,S')\colon S{-}t{\rightarrow}S' \wedge S, S' \in \mathcal{R}(S_0)\}$. In other words, an LTS graph presents all reachable states and transitions among them in the form of the directed graph.

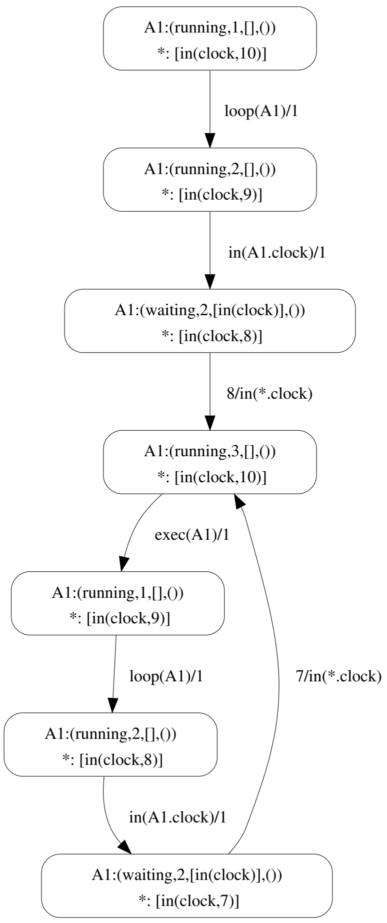


Fig. 6. LTS for the model from Fig. 5.

The LTS graph for the model from Fig. 5 is presented in Fig. 6. We have omitted the parameter n value in order to receive a finite graph. Labels in the presented graph are of the form $t_1/transitions/t_2$, where t_1 stands for the time the system stays in the *old* state and t_2 stands for the duration of the step. If any of the time values is equal to 0, it is omitted together with the corresponding slash. The $loop(A1)/1$ means that the agent executes the *loop* transition and it takes 1 ms.

Let us focus on the state S_2 ($n = 0$):
$S_2 = ((waiting, 2, [in(clock)], (0)), [in(clock, 8)])$.
In the state S_1 agent A performs an *in* step to collect a signal from the input border port $clock$. There is no signal to collect thus, the agent changes its mode to $waiting$ and waits for such a signal (additional entry is included into the agent ci list). Seven milliseconds later such a signal is generated by the environment that is represented by the t_{*in} transition. This transition finalises the agent $A1$ communication with its environment. The signal is collected and the agent program counter takes the next value.

Suppose, $in(clock) \notin ci_2(A1)$. Thus, an execution of the t_{*in} transition denotes a generation of a signal by the environment, but the signal is not handled by the system.

Suppose, the border port $clock$ is specified as follows:

```
in clock [] (map (10*) [1..]) durable;
```

end $in(clock) \notin ci(A1)$ when the t_{*in} transition is executed. Thus, as a result of the transition execution the $in(*.clock)$ is included into the ci list. It means that a signal provided via port $clock$ waits for handling. When executing the t_{in} transition, agent A collects the signal and immediately moves to a state with $pc(A1) = 3$. If the *queue* modifier is used instead of *durable*, a context information list of an agent may contain more than one $in(*.clock)$ entry. Each of such entries represents a signal waiting for handling.

Suppose, the border port $clock$ is specified as follows:

```
in clock [] (map (10*) [1..]) signal;
```

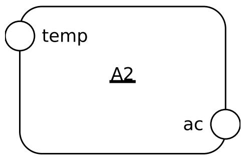
This means that the environment only may provide a signal via port $clock$ every 10 ms, but we have no confidence that such a signal will be generated. This modifier can be connected with *durable* and *queue* ones.

To specify that a signal via port may be provided at any time, we can use the following clause:

```
out clock [] [] signal;
```

In such a case agent $A1$ still counts the collected signals, but this time there is not any regularity with respect to time points the signals appear. Moreover, some signals can be lost, if they come too often.

Let us focus on the $A2$ agent that is a part of an air conditioner controller. The agent reads the current temperature and turns on/off the air conditioner depends on the comparison of the current temperature with a set threshold temperature. An Alvis model of such a system is given in Fig. 7.



```
environment {
  in  temp [10..30] [];
  out ac Bool [];
}

agent A2 {
  t :: Int = 0;
  threshold :: Int = 21;
  loop {                                    -- 1
    in temp t;                              -- 2
    if(t > threshold) { out ac True; }  -- 3,4
    else              { out ac False; } -- 5
    delay 30000;                            -- 6
  }
}
```

Fig. 7. Collecting a sensor value from the environment.

Using the empty list in the ac border port specification means that agent $A2$ can send a Boolean value via this port any time it is necessary.

The LTS graph for the model from Fig. 7 is shown in Fig. 8. To reduce the graph size we have used letters *L* and *G* instead of values less than or equal to the threshold and greater than the threshold respectively. The *time* label represents the passage of time. This transition is executed only if there is no

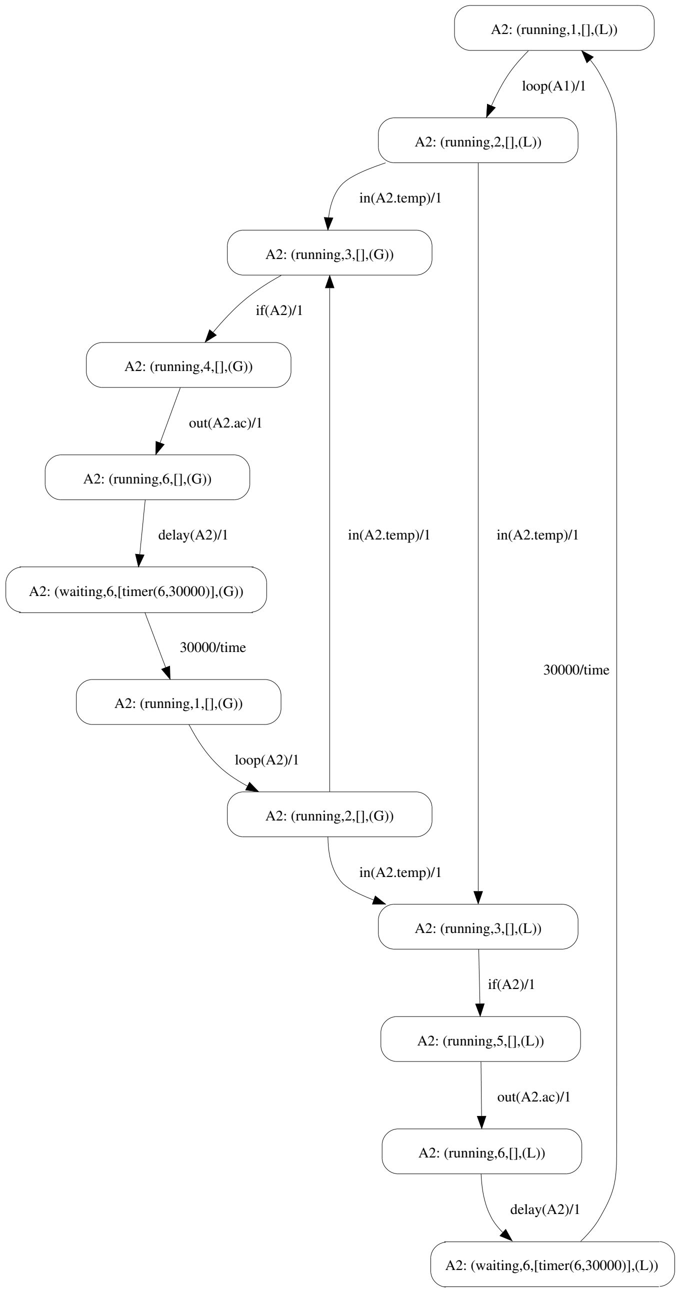


Fig. 8. LTS graph for the model from Fig. 7.

any other active one and at least one agent contains a *timer* entry on its context information list.

VI. Summary

Alvis communication with environment methods have been presented in this paper, together with a basic language description. The environment specification was also introduced, which is crucial for an effective LTS generation. From designers point of view, the Alvis system description is composed of two parts. The first one is a graphical system view, which shows agents and the way they communicate one with another. The second one is a textual specification of agents and the environment behaviour. Together with a system layer, they provide a possibility to simulate and verify an Alvis model. A formal verification of Alvis models is based on LTS graphs generated by the Alvis Translator [16] and external tools provided by the CADP package [8]. CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking. This approach provides a possibility to use also new methods of a state space exploration in future.

In spite of process algebra origins, Alvis seems to be more convenient from engineering point of view than process algebras. Moreover, a system behaviour is related to selected execution scenario, which is determined by the hardware that executes a model. Alvis is able to perform an execution similar to process algebra formalisms with unlimited numbers of processing units (α^0 system layer), but also can behave as a one or two processors units with selected scheduling functions.

References

[1] M. Szpyrka, P. Matyasik, and R. Mrówka, "Alvis – modelling language for concurrent systems," in *Intelligent Decision Systems in Large-Scale Distributed Environments*, ser. SCI. Springer-Verlag, 2011, vol. 362, pp. 315–342.

[2] M. Szpyrka, *Alvis On-line Manual*, AGH UST, Krakow, Poland, 2012. [Online]. Available: http://fm.ia.agh.edu.pl/alvis:manual

[3] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

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

[5] M. Szpyrka and T. Szmuc, "Verification of automatic train protection systems with RTCP-nets," in *Computer Safety, Reliability and Security*, ser. LNCS, J. Górski, Ed. Springer-Verlag, 2006, vol. 4166, pp. 344–357.

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

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

[8] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A toolbox for the construction and analysis of distributed processes," in *Computer Aided Verification*, ser. LNCS, vol. 4590. Springer-Verlag, 2007, pp. 158–163.

[9] K. Balicki and M. Szpyrka, "Formal definition of XCCS modelling language," *Fundamenta Informaticae*, vol. 93, no. 1–3, pp. 1–15, 2009.

[10] P. Matyasik, "Design and analysis of embedded systems with XCCS process algebra," Ph.D. dissertation, AGH University of Science and Technology, Kraków, Poland, 2009.

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

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

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

[14] M. Szpyrka, L. Kotulski, and P. Matyasik, "Specification of embedded systems environment behaviour with Alvis modelling language," in *Proc. of the 2011 Int. Conf. on Embedded Systems and Applications ESA'11 (Worldcomp 2011)*, Las Vegas, Nevada, USA, July 18–21 2011, pp. 79–85.

[15] M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski, "Formal modelling and verification of concurrent systems with Alvis," *International Journal of Applied Mathematics and Computer Science*, 2012, (to appear).

[16] L. Kotulski, M. Szpyrka, and A. Sędziwy, "Labelled transition system generation from Alvis language," in *Knowledge-Based and Intelligent Information and Engineering Systems – KES 2011*, ser. LNCS. Springer-Verlag, 2011, vol. 6881, pp. 180–189.