Int. J. Appl. Math. Comput. Sci., , Vol. , No. , –
DOI:

FORMAL INTRODUCTION TO ALVIS MODELLING LANGUAGE

MARCIN SZPYRKA [*], PIOTR MATYASIK [*], RAFAŁ MRÓWKA [*], LESZEK KOTULSKI [*], KRZYSZTOF BALICKI [**]

[*] Department of Automatics
AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: mszpyrka,ptm,Rafal.Mrowka,kotulski@agh.edu.pl

[**]Institute of Mathematics
Rzeszów University, Al. Rejtana 16A, 35-959 Rzeszów, Poland
e-mail: kbalicki{@}univ.rzeszow.pl

The paper presents a formal introduction to Alvis, a novel modelling language designed for embedded systems. Alvis has been defined as a kind of a happy medium between formal and practical modelling languages. It combines possibilities of formal models verification with flexibility and simplicity of practical programming languages. Even though Alvis has its origin in process algebras, it combines flexible graphical modelling of interconnections among agents with a high level programming language used for the description of agents behaviour. Users can choose one of a few varieties of Alvis semantics that depend on the so-called system layer. The most universal system layer is denote by α^0 and makes Alvis similar to other formal languages like Petri nets, process algebras, time automata, etc. Alvis with α^0 system layer seems to be valuable modelling language for concurrent systems.

Keywords: Alvis modelling language, embedded systems, formal verification

1. Introduction

The aim of the paper is to present a formal description of the *Alvis modelling language* with the α^0 system layer. Alvis is a successor of the XCCS modelling language (Szpyrka and Matyasik, 2008), (Balicki and Szpyrka, 2009), (Matyasik, 2009), which was an extension of the CCS process algebra (Milner, 1989), (Fencott, 1995), (Aceto *et al.*, 2007).

Alvis combines hierarchical graphical modelling with a high level programming language. A model consists of three layers. The *graphical layer* is used to define data and control flow among agents, where *agent* denotes any distinguished part of the system under consideration with defined identity persisting in time. The *code layer* is used to describe behaviour of individual agents. Instead of algebraic equations, Alvis uses a high level programming language based on the Haskell syntax for this purpose. Moreover, Haskell is used to define data types for parameters and to define functions for data manipulation. The third *system layer* is the predefined one. It gathers information about all agents in a model and their states. Moreover, choosing one of accessible system layers entails choosing a scheduling algorithm and hardware architecture for the model. The system layer is also used e.g. for generation of an LTS (labelled transition system) graph for a model.

Alvis as well as other formal methods like process algebras, Petri nets or time automata, can be used for modelling concurrent systems. In contrast with these methods, Alvis has been worked out especially for embedded systems and can be used for mapping both software and hardware aspects of an embedded system at the same time. Thus, an Alvis model may describe not only an application, but also selected elements of an operating system and a hardware that are essential from the considered system point of view. In our opinion, such holistic approach to the design of embedded systems is more valuable than approaches that disregard the operating system and hardware aspects. Moreover, Alvis seems to be more user-friendly, from engineering point of view, than classical formal methods.

Similar to other formal methods, Alvis provides a possibility of formal verification of models. An Alvis model can be transformed into a *labelled transition sys-*

tem (LTS). After encoding such graph using the *Binary Coded Graphs* (BCG) format, its properties are verified with the CADP toolbox (Garavel *et al.*, 2007).

The paper is organised as follows. Section 2 contains a review of other modelling languages used for embedded systems development and their comparison with Alvis. The α^0 system layer is described in Section 3. Section 4 presents the main features of communication diagrams. Section 5 describes Alvis statements used in the code layer. A formal definition of an Alvis model is given in Section 6. Section 7 deals with a formal description of a model dynamic and Section 8 describes LTS graphs used for verification purposes. A short summary is given in the final section.

2. Related works

Real-time and embedded systems are a strictly distinguished type of computer systems with a set of programming languages used for them in industry. Alvis is a novel proposition for such systems with following advantages:

- a graphical modelling language used to define interconnections among agents;
- a high level programming language used to define behaviour of individual agents (instead of algebraic equations);
- a possibility of a formal model verification.

This section provides a short comparison of Alvis with other modelling languages used in industry for the embedded systems development.

E-LOTOS is an extension of the LOTOS modelling language (Language Of Temporal Ordering Specification) (ISO, 1989). The main intention of the E-LOTOS extension was to enable modelling of the hardware layer of a system. Thus, in the specification, we can find such artifacts as interrupts, signals, and the ability to define events in time. With such extensions, E-LOTOS significantly expanded the possibility of using the algebra of processes, which is the starting point for the specification in this language.

It should be noted that the Alvis language has many features in common with E-LOTOS. First of all, Alvis as E-LOTOS is derived from process algebras. Alvis, like E-LOTOS, was intended to allow formal modelling and verification of distributed real-time systems. To meet the requirements, Alvis provides a concept of time and a delay operator. In contrast to E-LOTOS, Alvis provides graphical modelling language. Moreover, Alvis toolkit supports a LTS graph generation, which significantly simplifies the formal verification of models.

Esterel (Berry, 2000), (Palshikar, 2001) is a high-level formal synchronous language created to program reactive systems at a cycle-accurate behavioral level. The original textual language was later complemented by the SyncCharts (Andre, 2003) hierarchical automata graphical formalism (now called Safe State Machines or SSMs in Esterel). Textual and graphical specifications can be freely mixed. Esterel encompasses state sequencing, signal emission and reception, concurrency, and preemption structures to drive the life and death of control component behaviours in a hierarchical way.

Esterel is one of a family of synchronous languages. It uses the *broadcast* as the unique communication mechanism. The broadcast mechanism means that a signal cannot have any destination specified; all signals are broadcast and any module may listen to and read an emitted signal. Also, signals do not have any unique identifier. In contrast to Esterel, Alvis uses synchronous communication model similar to Ada's *hand shaking* or agents communication in the CCS process algebra. Agents in Alvis may communicate one with another only if a communication channel between their ports is explicitly defined. Both Alvis and Esterel support pure and value passing communication.

SCADE (Est, 2007) is a product developed by the Esterel Technologies company. It is a complex tool for developing a control software for embedded critical systems and for distributed systems. A system is described as an input to output transformation. In every cycle inputs are transformed to outputs according to a specification provided by functions: linear and discrete and state machine. SCADE allows system developer to choose from a large library of predefined components. The KCG code generator, which is a part of the SCADE suite, produces C code that has all the properties required for safety-critical software. SCADE also provides tools for checking system specification and verification of the developed model.

The Alvis approach is very different. The system in Alvis is represented as a set of communicating tasks which are continuously processing their instructions. Alvis also has no code generation phase, because it is an executable specification itself. Moreover, the system verification in Alvis is based on an LTS graph generation instead of specification-model consistency and statical code checking. SCADE and Alvis have also different approaches to types. The first one adopts simple static C language types due to specific runtime requirements, while the second one uses the Haskell type system.

System Modelling Language (SysML)(Obj, 2008) aims to standardize a process of a system specification and modelling. The original language specification was developed as an open source project on behalf of the International Council on Systems Engineering INCOS and the Object Management Group (OMG). SysML is a general purpose modelling language for systems engineering applications. In particular, it adds two new types of diagrams: requirement and parametric diagrams. The Alvis language has many common features with the SysML block diagrams and activity diagrams: ports, property blocks, communication among the blocks, hierarchical models. Unlike SysML, Alvis combines structure diagrams (block diagrams) and behaviour (activity diagrams) into a single diagram. In addition, Alvis defines formal semantics for the various artifacts, which is not the case in SysML. The concept of agent in Alvis corresponds with the SysML block definition. The formal semantics of Alvis allows you to create automated tools for verification, validation and runtime of Alvis models. SysML is a general-purpose systems modelling language, which covers most of the software engineering phases from analysis to testing and implementation. Alvis is focused on the structural model, the behavioural aspects of the system and formal verification of its properties. Its main area of application are distributed and embedded real-time systems. Alvis can be used as an extension to the software engineering process based on SysML.

Another solution used for years in industry is the Very High Speed Integrated Circuits Hardware Description Language (VHDL) (Ashenden, 2008). This language allows for a specification, development and verification of digital circuits. The syntax of VHDL is based on the structures found in programming languages such as Pascal and Ada. VHDL provides a hierarchical construction of models, similar to SysML and Alvis. The concept of agent in Alvis is represented as a design entity in VHDL. The design entity consists of an entity definition and a body architecture. Communication with the environment takes place through declared ports like in Alvis. The body of a design entity contains the following blocks: value-signal assignments, processes and components. Processes allow designers to specify concurrent systems, whereas components enable them to decomposite and combine multiple modules into one system. Due to the Ada origins, VHDL and Alvis have a similar syntax for the communication and parallel processing. However, it should be noted that Alvis is closely linked with its graphical model layer. The graphical composition allows users for an easy identification of a system hierarchy and components. The main purpose of VHDL is a specification of digital electronic circuits and it focuses on a system hardware. However, Alvis integrates the hardware and the software view of a system. In this way, Alvis allows for a comprehensive verification and validation of modelled systems.

3. System layer

The *system layer* (or *meta-data layer*) is the predefined one. It is necessary for a model simulation and analysis. From users point of view the layer works in the read-only mode. It gathers information about all agents in a model and their states. Agents can retrieve some data from the layer, but they cannot directly change them. The system layer provides some functions that are useful for implementation of scheduling algorithms or for retrieving information about other agents states.

User can choose one of a few versions of the layer but it affects the developed model semantic. The system layer is strictly connected with the system architecture and the chosen operating system. Alvis has been worked out especially for embedded systems. One of the starting points for Alvis development has been an analysis of Atmel NGW100 single-board computer, which runs AVR32 microprocessor, and FreeRTOS real-time operating system for embedded devices. Alvis can be used for mapping both software and hardware aspects of an embedded system at the same time. Thus, an Alvis model may contain not only the application under consideration, but also selected elements of its operating system and hardware that are essential from the considered system point of view.

System layers differ in scheduling algorithms and system architectures mainly. There are considered two approaches to the scheduling problem. System layers with α symbol provide a predefined scheduling function that is called after each step automatically. On the other hand, system layers with β symbol do not provide such a function. User must define a scheduling function himself.

Both α and β symbols are usually extended with some indicators put in the superscript or/and subscript. An integer put in the superscript denotes the number of processors in the system. Zero is used to denote the unlimited number of processors. A symbol put in the subscript denotes the selected system architecture or/and chosen scheduling algorithm.

In this paper we consider only the α^0 system layer. This layer makes Alvis an universal formal modelling language similar to Petri nets or process algebras. The α^0 layer scheduler is based on the following assumptions.

- Each active agent has access to its own processor and performs its statements as soon as possible.
- The scheduler function is called after each statement automatically.
- 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 indeterministicly.

  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.

4. Communication diagrams

Communication diagrams are the visual part of the Alvis modelling language. They are used to represent the structure of the system under consideration. A communication diagram is a hierarchical graph that nodes may represent both agents (*active* or *passive*) and parts of the model from the lower level. They are the only way, in Alvis, to point out agents that communicate one with the other. Moreover, the diagrams allow programmers to combine sets of agents into modules that are also represented as agents (called *hierarchical agents*). It should be underlined that a communication diagram is only a part of an Alvis model. The complete model consists of three layers (graphical, code, system).

4.1. Non-hierarchical diagrams. Alvis provides hierarchical communication diagrams used to describe an embedded system from the control and data flow point of view. A hierarchical diagram enable designers to distribute parts of a diagram across multiple subdiagrams called *pages*. Pages are combined using the so-called *substitution mechanism*. An active agent at one level can be replaced by a page on the lower level, which usually gives a more precise and detailed description of the activity represented by the agent. Such a substituted agent is called *hierarchical* one. Replacing hierarchical agents with corresponding subpages results in an equivalent non-hierarchical diagram. Thus, at the beginning, we will provide a formal definition of non-hierarchical diagrams, and next we will describe hierarchical mechanisms.

Let us focus on non-hierarchical diagrams in this subsection. Let **A** denote an Alvis model (with a non-hierarchical communication diagram). Graphical and code layers of a model are closely related one to the other. Each agent from a communication diagram is described in the corresponding code layer and vice versa.

An agent can communicate with other agents through *ports*. There is no distinction between input and output ports on communication diagrams. Any port can be used as an input or output one. Each agent port must have a unique identifier (name) assigned, but ports of different agents may have the same identifier assigned. Thus, each port in a model is identified using its name and its agent name. For simplicity, we will used the so-called *dot notation* – $X.p$ denotes the port p of the agent X.

Let us define the following symbols.

- $\mathcal{P}(X)$ denotes the set of ports of the agent X.
- $\mathcal{P}(D)$ denotes the set of ports of the diagram D (the page D).
- $\mathcal{N}(W)$ denotes the set of names of ports belonging to the set W.

For example, if a diagram contains only agents: X_1 with port p and X_2 also with port p, then $\mathcal{P}(D) = \{X_1.p, X_2.p\}$, and $\mathcal{N}(\mathcal{P}(D)) = \{p\}$.

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
$$\forall i = 1, \ldots, n\, (\mathcal{P}(X_i) \times \mathcal{P}(X_i)) \cap \mathcal{C} = \emptyset, \qquad (1)$$
where
$$\mathcal{P} = \bigcup_{i=1,\ldots,n} \mathcal{P}(X_i). \qquad (2)$$
Each element of the relation $\mathcal{C}$ is called a *connection* or a *communication chanel*.
- $\sigma\colon \mathcal{A}_A \to \{False, True\}$ is the *start function* that points out initially activated agents.

The condition (1) means that two ports of the same agent cannot be connected. The start function σ makes possible delaying activation of some agents (We can make them active later with the *start* statement.

An example of a communication diagram is shown in Fig. 1. Active agents are drawn as rounded boxes while passive ones as rectangles. An agent's identifier (name) is placed inside the corresponding shape. The first character of the identifier must be an upper-case letter. Other characters (if any) must be alphabetic characters, either upper-case or lower-case, digits, or an underscore. Alvis identifiers are case sensitive. Moreover, the Alvis keywords cannot be used as identifiers. Names of agents that are initially activated (represent running processes) are underlined.

Ports are drawn as circles placed at the edges of the corresponding rounded box or rectangle. Ports names must fulfil the same requirements as agents identifiers, but the first character of a port name must be an lower-case letter.

A *communication channel* is defined explicitly between two agents and connects two ports. 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.

4.2. Hierarchical communication diagrams. On the other hand, a part of a communication diagram can be treated as a module and represented by a single agent at the higher level. Thus, communication diagrams support both *top-down* and *bottom-up* approaches. For the effective modelling Alvis communication diagrams enable distributing parts of a diagram across multiple subdiagrams called *pages*. Pages are combined using the so-called *substitution mechanism*. An active agent at one level can be replaced by a page on the lower level, which usually gives a more precise and detailed description of the activity represented by the agent. Such a substituted agent is called

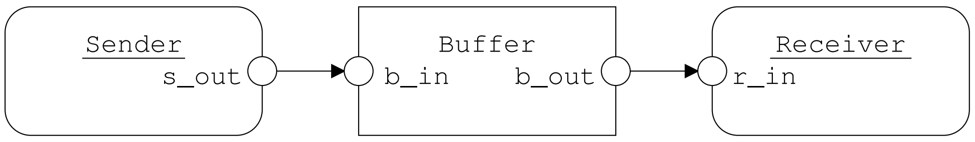


Fig. 1. Sender-Receiver system with buffer – communication diagram.

hierarchical one. All ports of a hierarchical agent must appear on the corresponding subpage.

A hierarchical communication diagram consists of a set of pages.

Definition 2. A *page* in a hierarchical communication diagram is a triple $D^i = (\mathcal{A}^i, \mathcal{C}^i, \sigma^i)$, where:

- $\mathcal{A}^i = \{X^i_1, \ldots, X^i_n\}$ is the set of *agents* with subsets of *active agents* $\mathcal{A}^i_A$, *passive agents* $\mathcal{A}^i_P$, and *hierarchical agents* $\mathcal{A}^i_H$, such that $\mathcal{A}^i = \mathcal{A}^i_A \cup \mathcal{A}^i_P \cup \mathcal{A}^i_H$, and $\mathcal{A}^i_A$, $\mathcal{A}^i_P$, $\mathcal{A}^i_H$ are pairwise disjoint.
- $\mathcal{C}^i$ and σ^i are defined as in Definition 1.

A page of a hierarchical communication diagram is shown in Fig. 2. The main difference between a non-hierarchical and hierarchical communication diagram is that the latter may contain hierarchical agents. They are are indicated by black triangles. A page without hierarchical agents is simply a non-hierarchical communication diagram.

Let a hierarchical agent Y be given and let $join_Y(D^i)$ denotes the set of all *join ports* of the page D^i with respect to Y, i.e.

$$join_Y(D^i) = \{X^i_j.p \in \mathcal{P}(D^i) \colon \mathcal{N}(X^i_j.p) \in \mathcal{N}(\mathcal{P}(Y))\}. \tag{3}$$

In other words, $join_Y(D^i)$ is the set of all ports from the page D^i that names are the same as those of the hierarchical agent Y.

Definition 3. Let a hierarchical agent Y and a page $D^i = (\mathcal{A}^i, \mathcal{C}^i, \sigma^i)$ be given.

- The agent Y and the page D^i satisfy the *simple substitution* requirements, iff
$$card(\mathcal{P}(Y)) = card(join_Y(D^i)), \tag{4}$$
and
$$(join_Y(D^i) \times join_Y(D^i)) \cap \mathcal{C}^i = \emptyset, \tag{5}$$
- The agent Y and the page D^i satisfy the *extended substitution* requirements, iff
$$card(\mathcal{P}(Y)) < card(join_Y(D^i)) \tag{6}$$
and the equality (5) holds.

We will consider a *binding function* π that maps ports of a hierarchical agent to the join ports of the corresponding page. In the case of a simple substitution, the *binding function* π is a bijection. On the other hand, in the case of an extended substitution, one port of a hierarhical agent may have assigned more than one join port on the subpage.

Let us recall the definition of a labelled directed graph.

Definition 4. A *labelled directed graph* is a triple $\mathcal{G} = (V, E, L)$, where:

- V is the set of *nodes*.
- L is the set of *edge labels*.
- $E \subseteq V \times L \times V$ is the set of *edges*.

Definition 5. A *hierarchical communication diagram* is a pair $H = (\mathcal{D}, \gamma)$, where:

- $\mathcal{D} = \{D^1, \ldots, D^k\}$ is the set of *pages* of the hierarchical communication diagram, such that sets of agents $\mathcal{A}^i$ $(i = 1, \ldots, k)$ are pairwise disjoint.
- $\gamma \colon \mathcal{A}_H \to \mathcal{D}$, where $\mathcal{A}_H = \bigcup_{i=1,\ldots,k} \mathcal{A}^i_H$, is the *substitution function*, such that:
  1. γ is an injection.
  2. For any $X^i_j \in \mathcal{A}_H$, X^i_j and $\gamma(X^i_j)$ satisfy the requirements of the simple or extended substitution.
  3. Labelled directed graph $\mathcal{G} = (\mathcal{D}, E, \mathcal{A}_H)$ where $(D^i, X^i_k, D^j) \in E$ iff $\gamma(X^i_k) = D^j$ is a tree or a forest.

The labelled directed graph defined above is called a *page hierarchy graph.* Nodes of such a graph represent pages, while edges (labelled with names of hierarchical agents) represent the substitution function γ. Each edge represents the page to which belongs the hierarchical agent (used as label) and the subpage associated with the agent.

We assume that system definition starts from a page or a sets of page, thus the number of pages must be greater than the number of hierarchical agents. Formally pages from the set $\mathcal{D} - \gamma(\mathcal{A}_H)$ are called *primary pages*, They are roots of trees that constitute a page hierarchy graph.

Following symbols are valid for hierarchical communication diagrams:

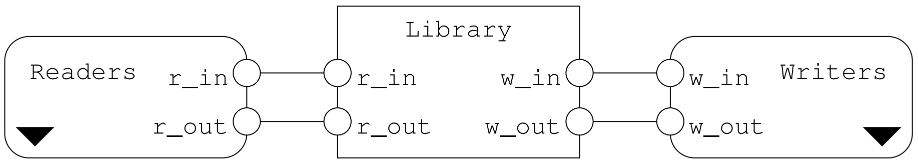


Fig. 2. Readers-Writers system – top level page of the communication diagram.

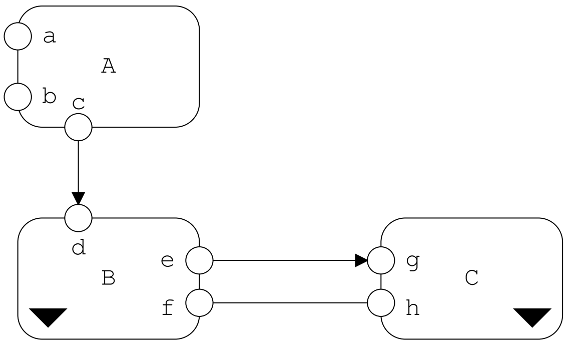


Fig. 3. Page D^1.

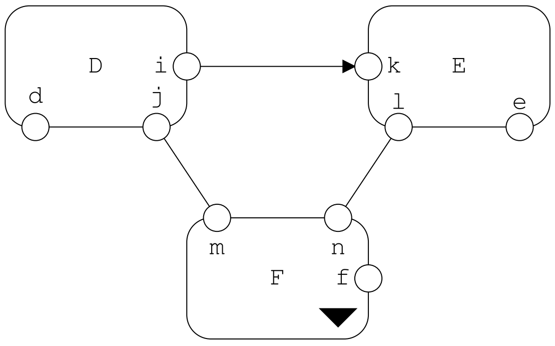


Fig. 4. Page D^2.

- $\mathcal{A}_A = \bigcup_{i=1,\dots,k} \mathcal{A}_A^i$,
- $\mathcal{A}_P = \bigcup_{i=1,\dots,k} \mathcal{A}_P^i$,
- $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$,
- $\sigma\colon \mathcal{A}_A \to \{\mathit{False}, \mathit{True}\}$ and $\forall i = 1,\dots,k\ \forall X_j^i \in \mathcal{A}_A^i\colon \sigma(X_j^i) = \sigma^i(X_j^i)$.

To define the global set of connections, we have to take into account not only connections from sets $\mathcal{C}^i$, but also connections resulting from replacing hierarchical agents with subpages. For any page D^i we define a set of hierarchical connections $\mathcal{C}_H^i$ as follows:

$$\begin{aligned}\mathcal{C}_H^i = &\{(X_l^j.p, X_m^i.q)\colon \exists X_n^j \in \mathcal{A}_H^j \wedge \\ &(X_l^j.p, X_n^j.q) \in \mathcal{C}^j \wedge \gamma(X_n^j) = D^i\} \cup \\ &\{(X_m^i.q, X_l^j.p)\colon \exists X_n^j \in \mathcal{A}_H^j \wedge \\ &(X_n^j.q, X_l^j.p) \in \mathcal{C}^j \wedge \gamma(X_n^j) = D^i\}\end{aligned} \tag{7}$$

Finally, the global set of hierarchical connections $\mathcal{C}_H$ is the sum:

$$\mathcal{C}_H = \bigcup_{i=1,\dots,k} \mathcal{C}^i \cup \mathcal{C}_H^i. \tag{8}$$

An example of the simple substitution is shown in Fig. 3 and 4. The page shown in Fig. 4 is assigned to the agent B. The following equalities hold.

- $\mathcal{P}(B) = \{B.d, B.e, B.f\}$
- $\mathit{join}_B(D^2) = \{D.d, E.e, F.f\}$
- $\mathcal{N}(\mathcal{P}(B)) = \{d, e, f\} = \mathcal{N}(\mathit{join}_B(D^2))$

Of course, the binding function *binds* ports with the same names.

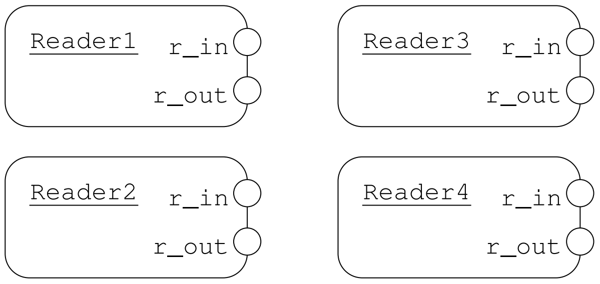


Fig. 5. Readers-Writers system – page *pReaders*.

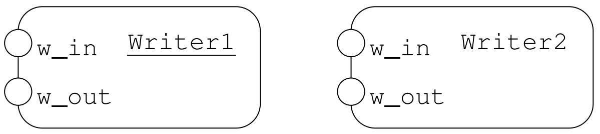


Fig. 6. Readers-Writers system – page *pWriters*.

An example of the extended substitution is shown in Fig. 2, 5 and 6. The page hierarchy graph for the readers-writers model is shown in Fig. 7.

Both substitions used in the considered model are the extended ones. Let focus on the *Readers* agent. The following equalities hold:

- $\mathcal{P}(\mathit{Readers}) = \{\mathit{Readers}.r\_in, \mathit{Readers}.r\_out\}$
- $\mathit{join}_{\mathit{Readers}}(\mathit{pReaders}) = \{\mathit{Reader}1.r\_in, \mathit{Reader}1.r\_out, \mathit{Reader}2.r\_in, \mathit{Reader}2.r\_out, \mathit{Reader}3.r\_in, \mathit{Reader}3.r\_out, \mathit{Reader}4.r\_in, \mathit{Reader}4.r\_out\}$

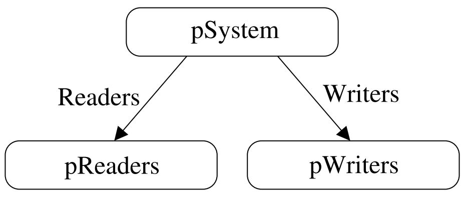


Fig. 7. Page hierarchy graph

- $\mathcal{N}(\mathcal{P}(Readers)) = \{r\_in, r\_out\} = \mathcal{N}(join_{Readers}(pReaders))$

In this case, the binding function π is defined as follows:

- $\pi(Readers.r\_in) = \{Reader1.r\_in, \ldots, Reader4.r\_in\}$
- $\pi(Readers.r\_out) = \{Reader1.r\_out, \ldots, Reader4.r\_out\}$

Instead of *local* binding functions, we can consider one *global* function π:

$$\pi\colon \bigcup_{X \in \mathcal{A}_H} \mathcal{P}(X) \to 2^{\mathcal{P}}. \tag{9}$$

The function π satisfies the following conditions:

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \pi(X.p) \subseteq join_X(\gamma(X)), \tag{10}$$
$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \mathcal{N}(\pi(X.p)) = \{p\}. \tag{11}$$

If a communication diagram contains only simple substitutions, then the function (9) takes the simplified form:

$$\pi\colon \bigcup_{X \in \mathcal{A}_H} \mathcal{P}(X) \to \mathcal{P}, \tag{12}$$

and the condition (10):

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \pi(X.p) \in join_X(\gamma(X)). \tag{13}$$

It can be useful to designate relations between hierarchical agent and agents belonging to its subpage.

Definition 6. Let $X \in A_H$ and a page D^i such that $\gamma(X) = D^i$ be given. For any agent $Y \in A^i$ we say that X is *directly hierarchically dependent on* Y and we will denote it as $X \succ Y$.

4.3. Hierarchy elimination.

The possibility of substitution of an abstract description of an agent by a more detailed one represented by a submodel (subpage) it is very common in a system design. It is however difficult when we would like to understand (or verify) a behaviour of a whole system, associations among their components and so on. Thus, in this section we introduce the *flat* (non-hierarchical) abstraction of a system represented by its *hierarchical communication diagram*. In this representation we will use only agents and connections among them inherited from the *hierarchical communication diagram*.

Definition 7. For any two agents $X \in \mathcal{A}_H$ and $Y \in \mathcal{A}$, X is said to be *hierarchically dependent on* Y, denoted as $X \succeq Y$, iff $X = Y_1 \succ \ldots \succ Y_k = Y$ for some $Y_1, \ldots, Y_n \in \mathcal{A}$.

Definition 8. A *flat representation* of a communication diagram $H = (\mathcal{D}, \gamma)$ is the triple $(\mathcal{F}, \mathcal{C}', \sigma')$ such that:

1. $\forall X, Y \in \mathcal{F} \subseteq \mathcal{A}\colon X \not\succeq Y$,
2. $\forall X \in \mathcal{A} - \mathcal{A}_H\ \exists Y \in \mathcal{F}\colon Y \succeq X$,
3. $\mathcal{C}' = \{(X.p, Y.q) \in \mathcal{C}_H\colon X, Y \in \mathcal{F}\}$,
4. $\sigma' = \sigma|_{\mathcal{A}_A \cap \mathcal{F}}$.

It is easy to check that the set of *primary pages* is a *flat representation* of a system represented by a hierarchical communication diagram.

We can move from one flat system representation to another, more detailed one, using the analysis operation.

Definition 9. Let H be a hierarchical communication diagram, $(\mathcal{F}, \mathcal{C}', \sigma')$ be a flat representation of H, $X \in \mathcal{A}_H \cap \mathcal{F}$ and $\gamma(X) = D^i = (\mathcal{A}^i, \mathcal{C}^i, \sigma^i)$. *Analysis of the flat representation* $(\mathcal{F}, \mathcal{C}', \sigma')$ of the hierarchical diagram H in context of X is the flat representation $(\mathcal{F}^*, \mathcal{C}^*, \sigma^*)$ (denoted $\mathrm{AN}(H, \mathcal{F}, X)$), such that:

1. $\mathcal{F}^* = \mathcal{F} - \{X\} \cup \mathcal{A}^i$,
2. $\mathcal{C}^* = \{(Z.p, Z'.q) \in \mathcal{C}_H\colon Z, Z' \in \mathcal{F}^*\}$,
3. $\sigma^* = \sigma|_{\mathcal{A}_A \cap \mathcal{F}^*}$.

Definition 10. Let H be a hierarchical communication diagram, $(\mathcal{F}, \mathcal{C}', \sigma')$ be a flat representation of H, $Y \in \mathcal{F}$ and $\exists X \in A_H$ such that $X \succ Y$ and $\gamma(X) = D^i = (\mathcal{A}^i, \mathcal{C}^i, \sigma^i)$. *Synthesis of the flat representation* $(\mathcal{F}, \mathcal{C}', \sigma')$ of the hierarchical diagram H in context of Y is the flat representation $(\mathcal{F}^*, \mathcal{C}^*, \sigma^*)$ (denoted as $\mathrm{SN}(H, \mathcal{F}, Y)$) such that:

1. $\mathcal{F}^* = \mathcal{F} - \mathcal{A}^i \cup \{X\}$,
2. $\mathcal{C}^* = \{(Z.p, Z'.q) \in \mathcal{C}_H\colon Z, Z' \in \mathcal{F}^*\}$,
3. $\sigma^* = \sigma|_{\mathcal{A}_A \cap \mathcal{F}^*}$.

Page D' (presented in Fig. 8) is a flat representation of the hierarchical system H defined by pages D^1 and D^2 (presented in Fig. 3 and 4) with the simple substitution mechanism. Flat representation generated by the $\mathrm{AN}(H, D^1, B)$ analysis operation (Fig. 8) is generated by the following algorithm.

1. Remove the agent B from the page D^1 with all its connections.
2. Move the contents of the page D^2 onto the page D^1.

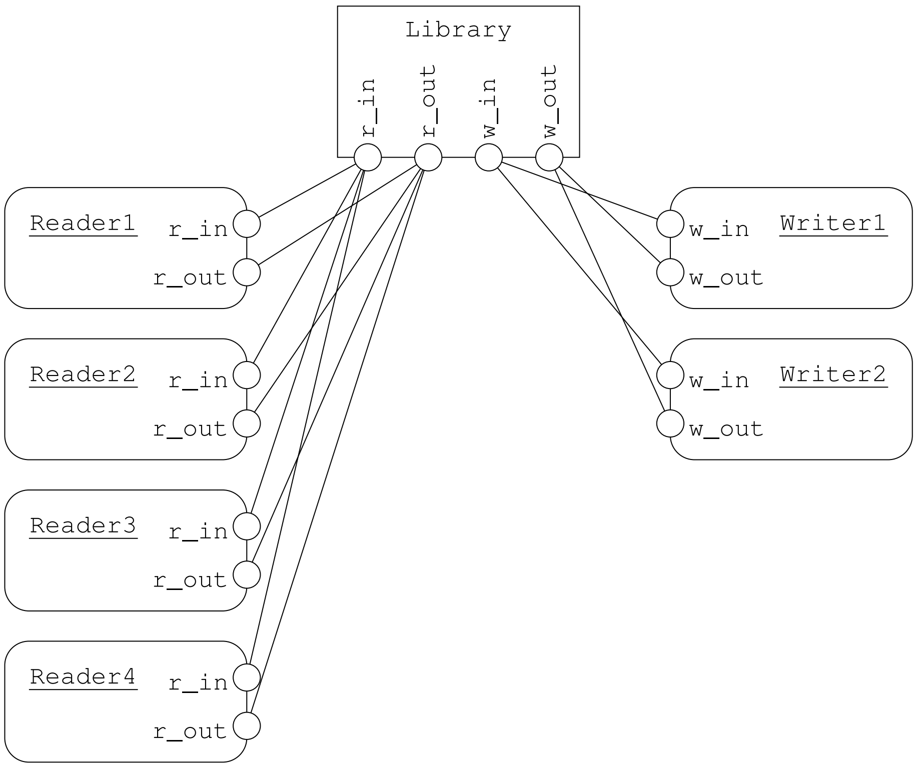


Fig. 9. Application of the extended substitution.

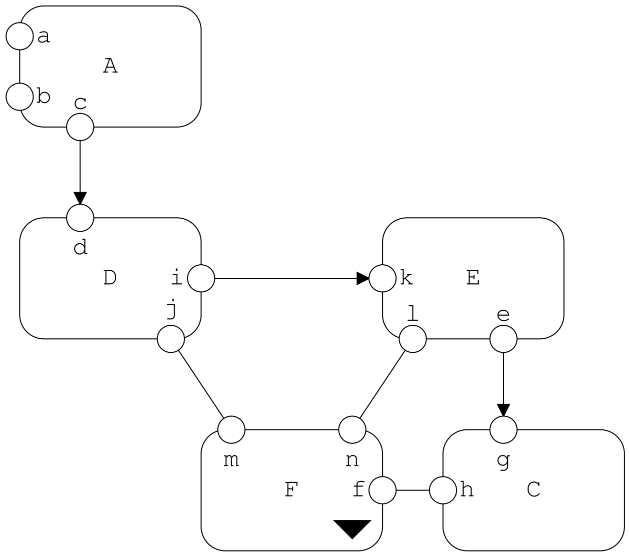


Fig. 8. Application of the simple substitution.

3. Add connections – If after removing of the agent B, from the page D^1, it has been removed a connection between ports $B.a$ and $X^1_i.p$, then we add a connection between ports $X^1_i.p$ and $\pi(B.a)$ with the same direction as the removed one.

Page *pSystem* (presented in Fig. 2) as a primary page is a flat representation of the hierarchical graph presented in Fig. 7 with pages *pReaders* and *pWriters* (presented appropriately in Fig. 5 and Fig. 6) with the extended substitution mechanism. Flat representation generated by the composition of the analysis operations $\mathrm{AN}(H, \mathrm{AN}(H, \mathit{pSystem}, \mathit{Readers}), \mathit{Writers})$ is presented in Fig. 9. This operation is supported by nearly the same algorithm as above with one change (in the third step). If after removing of a hierarchical agent X^i_j, it has been removed a connection between ports $X^i_j.p$ and $X^i_n.q$, then we add similar connections between port $X^i_n.q$ and all ports from the set $\pi(X^i_j.p)$.

In the next section we consider a flat representation without hierarchical agents, such a representation is maximal from the analysis point of view.

Definition 11. A flat representation $(\mathcal{F}, \mathcal{C}', \sigma')$ is called the *maximal flat representation* iff

$$\forall X \in \mathcal{A}\ \exists Y \in \mathcal{F} \colon X \succeq Y. \tag{14}$$

5. Code layer

The *code layer* is used to describe the behaviour of individual agents in Alvis models. The layer uses Alvis behaviour description statements and some elements of the Haskell functional programming language. In spite of the fact that Alvis has its origin in CCS (Aceto *et al.*, 2007), (Fencott, 1995), (Milner, 1989) and XCCS (Balicki and Szpyrka, 2009). (Matyasik, 2009) 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.

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.

```
-- Preamble:
--   types
--   constants
--   functions
--   environment specification

-- Implementation:
agent AgentName;
-- declaration of parameters
-- agent body
```

Listing 1. Structure of the code layer

The general structure of the code layer is presented in Listing 1. 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* contains definitions of the agents' behaviour. This part is encoded using native Alvis statements, but the preamble contents is used to represent parameters values and to manipulate them. It contains at least one *agent block* as shown in Listing 1. It is possible to share one definition among a few agents. In such a case, a few agents' names are placed after the keyword *agent* separated by commas. If necessary, an agent's 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. 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. Selected basic Haskell types recommended to be used in Alvis are as follows:

- *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.

The most common composite data types in Haskell (and Alvis) are *lists* and *tuples*. A *list* is a sequence 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. There is also a possibility to define a synonym for an existing type or to define new composite data types. For more details see for example (O'Sullivan *et al.*, 2008).

Parameters are defined using the 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. It should be underlined that 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.

The subset of Alvis statements used with the α^0 system layer is shown in Table 1. The table does not contain statements that use time explicitly e.g. **`delay`** statement. A more detail survey of Alvis statements can be found in (Szpyrka *et al.*, 2011).

Let us focus on a few selected statements that may need some explanation. The assignment operator is also used as a part of the *exec* statement. The *exec* statement is the default one. Therefore, the *exec* keyword can be omitted. Thus, to assign a literal value 7 to an integer parameter x one of the following statements can be used:

```
exec x = 7;
x = 7;
```

Table 1. Alvis statements used with the α^0 system layer

Statement	Description
`alt (g) {...}`	Defines a branch inside the *select* statement. The guard is optional.
`exec x = expression`	Evaluates the expression and assigns the result to the parameter; the *exec* keyword can be omitted.
`exit`	If an active agent performs the statement, it is terminated. If a passive agent performs the statement, its current procedure is terminated.

`if (g) {...} else {...}`
`if (g1) {...}`
`elseif (g2) {...}`
`elseif (g3) {...}`
`...`
`else {...}`
	If the guard is satisfied the *if* part is executed, otherwise the *else* part is executed.
	Extended version of the conditional statement.

`in p`
`in p x`
	Collects a signal (without value) 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*.

`loop (g) {...}`
 	
`loop {...}`
	Repeats execution of the contents while the guard if satisfied, the guard is checked everytime before entering the loop contents. – It is similar to the while loop in most languages.
	Infinite loop.

`null`	Empty statement.

`out p`
`out p x`
	Sends a signal (without value) via the port p.
	Sends the 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) {...}`
`  alt (g3) {...}`
`  ...`
`}`
	Selects one of the alternative choices. Guards $g1, g2, \ldots$ decide which alternatives can be chosen after entering the *select* statement.

`start A`	Starts the agent A if it is in the *Init* state, otherwise do nothing.

The assignment operator can also be followed by an expression. Alvis uses Haskell to define and manipulate data types, thus, such an expression must be encoded in Haskell and may contain Haskell functions.

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 a port can be a pure communication (a signal without a special value is transmitted) or a single value (probably of a composed type) can be sent/collected. A communication between an active and a passive agent or between two passive agents is similar to a procedure call. On the other hand, a communication between two active agents synchronises them.

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

Let focus on ports that are connected with at least one another port. A port $X.p \in \mathcal{P}$ can be used as an argument of the *in* statement iff there exists a port $X'.p' \in \mathcal{P}$, such that $(X'.p', X.p) \in \mathcal{C}$. Similarly, a port $X.p \in \mathcal{P}$ can be used as an argument of the *out* statement iff there exists a port $X'.p' \in \mathcal{P}$, such that $(X.p, X'.p') \in \mathcal{C}$.

On the other hand, Alvis models can use so-called *border ports* i.e. ports without any connections that are treated as communication channels with the considered embedded 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. Using *in* and *out* clauses, a designer can specify both values sent through the corresponding port and time points (in milliseconds), when the port can be used. 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.

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 can 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 (O'Sullivan *et al.*, 2008).

Border ports names must be unique in a model. It is possible to use a border port name more than once, but it means that more than one agent can send (or collect) signals through the same border port.

```
in  p1 [0,1] [];
in  p2 Bool  [];
out p3 [0,1] [];
out p3 Bool  [];
out p4 []    [];
```

Listing 2. Border ports specification – examples

Let us consider the border ports presented in Listing 2. At any time one of the values 0 or 1 (at random) can be collected through the port $p1$. Similarly, a Boolean value can be collected through the port $p2$. At any time one of the values 0 or 1 or a Boolean value can be sent through the port $p3$, and at any time a parameterless signal can be sent through the port $p4$.

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 *loop* statement.

Alvis provides *loop* and *jump* statements to define agents that repeats a sequence of statements. In the simplest form, the *loop* statement is used to define an infinite loop. If the *loop* keyword is followed by a guard, the loop behaves like the while loop in most programming languages. The guard of the *loop* statement is checked every time before entering the loop contents (put inside curly brackets). The *jump* statement uses labels that are identifiers followed by a colon. The statement is composed of the *jump* key word and a label name (without a colon). The *jump* statement is the key statement for translating algorithms from CCS to Alvis.

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 provides also statements directly related to time e.g., *loop every* or *delay* statements. However, it is out of the scope of the paper to consider statements and LTS graphs directly related to time.

Passive agents are used to store data shared among agents and to avoid the simultaneous use of such data by two or more agents. They provide a set of procedures that can be called by other 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 that is followed by a guard (optionally) and the corresponding port name. The procedure is accessible for other agents only if the guard evaluates to *True*.

6. Models

Formally, we define an Alvis model as a triple with a hierarchical communication diagram as shown in Definition 12.

Definition 12. An Alvis *model* is a triple $\mathbf{A} = (H, B, \varphi)$, where:

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

Moreover, each non-hierarchical agent X belonging to the diagram H must be defined in the code layer, and each agent defined in the code layer must belong to the diagram.

For an Alvis model $\mathbf{A} = (H, B, \varphi)$, its *equivalent non-hierarchical model* is a triple $\overline{\mathbf{A}} = (D, B, \varphi)$, where D is the maximal flat representation of H.

To describe the current state of an agent we need a tuple with four pieces of information:

- agent mode (am);
- program counter (pc);
- context information list (ci);
- parameters values tuple (pv).

Let us focus on passive agents firstly. 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 such a 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 such a case, ci contains the name of the called procedure (i.e. the name of the port used for current communication). The pc points out the index of the next statement to be executed or the current statement if the corresponding active agent is *waiting*.

An active agent can be in one of the following modes: *finished, init, ready, running, waiting*. (The *ready* mode is not used with the α^0 system layer.) An Alvis model contains a fixed number of agents. In other words, there is no possibility to create or destroy agents dynamically. If an active agent starts in the *init* mode, it is inactive until another agent activates it with the *start* statement. 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 using the *exit* statement. The statement is argumentless and an agent can terminate its work itself only. If an agent is in the *finished* mode, 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 names of the agent ports that can be used for the desired communication.

The last mode *running* used here means that an 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's name and the other agent's name (for procedures) are placed into ci. The pc points out the index of the current (e.g. for procedure call) or next agent statement. All possible transitions among modes of an active agent are shown in Fig. 10.

The formal definition of an agent state is as follows.

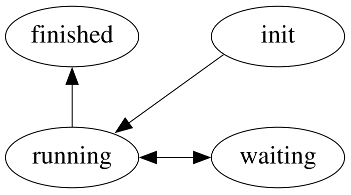


Fig. 10. Possible transitions among modes (without the *ready* mode).

Definition 13. A *state of an agent* X is a tuple

$$S(X) = (am(X), pc(X), ci(X), pv(X)), \qquad (15)$$

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.

It is very important to explain the way Alvis program counters work.

- We say that $pc(X)$ *points out* an *exec* (*exit, jump, null, start*) statement iff the next statement to be executed is an *exec* (*exit, jump, null, start*) statement.
- We say that $pc(X)$ *points out* an *in* or *out* statement iff the next statement to be executed is an *in* or *out* statement or the currently executing statement in *in* or *out* (e.g. an agent is waiting for a communication).
- We say that $pc(X)$ *points out* an *if* statement iff the next statement to be executed is the evaluating of the guard and entering one of the *if* statement alternatives.
- We say that $pc(X)$ *points out* a *loop* statement iff the next statement to be executed is the evaluating of the guard (if any) and possibly entering the *loop* statement.
- We say that $pc(X)$ *points out* a *select* statement iff the next statement to be executed is entering the *select* statement and possibly one of its branches.

Definition 14. A *state* of a model $\overline{\mathbf{A}} = (D, B, \varphi)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ and $\mathcal{A} = \{X_1, \ldots, X_n\}$ is a tuple

$$S = (S(X_1), \ldots, S(X_n)). \qquad (16)$$

7. Transitions

The transitions semantic given below is valid only for models with α^0 system layer i.e. triples $\mathbf{A} = (H, B, \alpha^0)$ (or equivalently $\overline{\mathbf{A}} = (D, B, \alpha^0)$). The set of all possible transitions for such models without time defined explicitly is given in Table 2.

Definition 15. The *initial state* of a model $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is a tuple S_0 as given in (16), where:

- $am(X) = \mathit{running}$ for any active agent X such that $\sigma(X) = \mathit{True}$;
- $am(X) = \mathit{init}$ for any active agent X such that $\sigma(X) = \mathit{False}$;
- $am(X) = \mathit{waiting}$ for any passive agent X;
- $pc(X) = 1$ for any active agent X in the *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.

To define formally results of transitions execution, we have to provide some mechanisms for code analysis. Let us define the following symbols.

- $B(X)$ – the X agent code definition (the agent block);
- $card(B(X))$ – the number of *steps* in $B(X)$;
- $B_i(X)$ for $i = 1, \ldots, card(B(X))$ – the name of the i-th step, $B_i(X) \in \{exec, exit, if, in, jump, loop, null, out, select, start\}$.
- $\mathcal{N}(t)$ – the name of the transition t (possible values the same as for steps).
- If necessary am, pc, ci, pv will be indicated by indexes S, S' etc. to point out the state they refer to.

The set of all transitions available for a particular model will be denoted by $\mathcal{T}$. For example, the t_{start} is available for a model $\overline{\mathbf{A}} = (D, B, \alpha^0)$ iif $\exists X \in \mathcal{A}, \exists i \in \{1, \ldots, card(B(X))\}\colon B_i(X) = start$.

Table 2. Set of transitions

Symbol	Description
t_{start}	starts an inactive agent
t_{exit}	terminates an agent or a procedure
t_{in}	performs communication (input side)
t_{out}	performs communication (output side)
t_{loop}	enters a loop
t_{jump}	jumps to a label
t_{if}	enters an if statement
t_{select}	enters a select statement
t_{exec}	performs an evaluation and assignment
t_{null}	performs an empty statement

Let us focus on the step idea. Statements such as *exec*, *exit*, *in*, *jump*, *null*, *out* and *start* 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.

```
agent A {
  i :: Int = 0;
  loop {                                -- 1
    select {                            -- 2
      alt (i == 0) { in p; i = 1;} -- 3, 4
      alt (i == 1) { in q; i = 0;} -- 5, 6
    }
    if(i == 1) { out p;}               -- 7, 8
    else { null; }                      -- 9
  }
}
```

Listing 3. Steps counting in Alvis code

Let us consider the piece of code shown in Listing 3. It contains 9 steps. The steps number are put inside comments. For example, the step 7 denotes entering the *if* statement, while the step 8 denotes the *out* statement. For passive agents, only statements inside procedures (i.e. inside curly brackets) are taken into consideration while counting steps.

To simplify the formal description of transitions, we will provide *nextpc* function that determine the number of the next step (the next program counter for an agent). For the purposes of this discussion *block* means a piece of a code inside curly brackets and *last block step* means that the step is the last one in the block and is followed by the closing curly bracket. Depending on the considered statement we will consider: *if blocks* (any of the blocks after *if*, *elseif* or *else* clauses), *loop blocks*, *branch blocks* (*alt* clauses), *procedure blocks* and *agent blocks* (a main agent's block). It is possible that there is no the last main block step e.g. if an agent behaviour is defined with an infinite loop (see Listing 3).

The $nextpc$ function takes an agent X state as an argument and returns an integer in the range of 0 to $card(B(X))$. The function satisfies the following requirements.

1. If the current step is not the last block step then:
   - if the step is a jump step then $nextpc(S(X))$ returns the number of the first step after the *jump* statement label.
   - otherwise $nextpc(S(X)) = pc_S(X) + 1$.
2. If the current step is the last main block step then $nextpc(S(X)) = 0$.

3. If the current step is a loop step (i.e. the step that denotes entering a loop) then:
    - if the guard is satisfied or for an infinite loop $\mathit{nextpc}(S(X)) = pc_S(X) + 1$;
    - if the guard is not satisfied then $\mathit{nextpc}(S(X))$ returns the number of the first step after the loop if it exists or 0 otherwise;
4. If the current step is the last loop block step then $\mathit{nextpc}(S(X))$ is equal to the number of the loop step.
5. If the current step is an if step then $\mathit{nextpc}(S(X))$ returns the number of the first step inside the chosen if block.
6. If the current step is the last if block step then $\mathit{nextpc}(S(X))$ returns the number of the first step after the *if* statement if it exists or 0 otherwise;
7. If the current step is a select step then $\mathit{nextpc}(S(X))$ returns the number of the first step inside the chosen branch block.
8. If the current step is the last branch step then $\mathit{nextpc}(S(X))$ returns the number of the first step after the *select* statement if it exists or 0 otherwise;
9. If the current step is the last procedure step then $\mathit{nextpc}(S(X)) = 0$.

Moreover, we will use the *firstpc* function that for a passive agent port returns the number of the first step in the corresponding procedure.

We will consider behaviour of Alvis models at the level of detail of single steps. Each of transitions presented in Table 2 realises a single step. Each step is realised in the context of one active agent. Also procedures of passive agents are realised in context of active agents that called them. Let us consider active agents firstly.

Definition 16. Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model with the current state $S = (S(X_1), \ldots, S(X_n))$ and $X_i \in \mathcal{A}_A$. A transition $t \in \mathcal{T}$ (see Table 2) is *enable* in the state S *with respect to* X_i iff the following requirements hold:

1. $am(X_i) = running$,
2. $B_{pc(X_i)}(X_i) = \mathcal{N}(t)$,
3. $ci(X_i) = [\,]$.

The clause *with respect to* will be omitted if the agent name is obvious or unimportant.

To simplify the description of transitions we will use the f/r abbreviation that means *finished* if the pc counter is equal to 0 in the considered state and $running$ otherwise.

If the transition t_{start} is enable in the state S with respect to the agent X_i and the agent $X_j \in \mathcal{A}_A$ is the argument of t_{start} (Only an active agent can be the argument of the *start* statement.) then the state S' that is the result of executing t_{start} in S (denoted as $S - t_{start} \rightarrow S'$) is defined as follows:

- $S'(X_i) = (f/r, \mathit{nextpc}(S(X_i)), [\,], pv_S(X_i))$
- If $am_S(X_j) = \mathit{init}$ then $S'(X_j) = (running, 1, [\,], pv_S(X_j))$
- If $am_S(X_j) \neq \mathit{init}$ then $S'(X_j) = S(X_j)$.
- States of other agents remain unchanged.

If the transition t_{exit} is enable in the state S with respect to the agent X_i then the state S' such that $S - t_{exit} \rightarrow S'$) is defined as follows:

- $S'(X_i) = (\mathit{finished}, 0, [\,], pv_S(X))$,
- States of other agents remain unchanged.

If a transition $t \in \{t_{jump}, t_{loop}, t_{null}\}$ is enable in the state S with respect to the agent X_i then the state S' such that $S - t \rightarrow S'$ is defined as follows:

- $S'(X_i) = (f/r, \mathit{nextpc}(S(X_i)), [\,], pv_S(X_i))$.
- States of other agents remain unchanged.

The t_{exec} transition changes the state in similar way but the value of one of the X_i parameters is updated. Moreover, the t_{if} transition also changes the state in similar way but the agent X_i mode is always $running$ in the S' state.

If the transition t_{in} is enable in the state S with respect to the agent X_i (assume port $X_i.p$ is the corresponding *in* statement argument and no value is sent through the port) then the state S' such that $S - t_{in} \rightarrow S'$) is defined as follows:

- If $X_i.p$ is a border port then $S'(X_i) = (f/r, \mathit{nextpc}(S(X_i)), [\,], pv_S(X_i))$.
- If $X_i.p$ is not a border port and there exists $X_j \in \mathcal{A}_A$ such that $(X_i.p, X_j.q) \in \mathcal{C}$, $am_S(X_j) = waiting$ and $ci_S(X_j) = [out(q)]$ then
  $S'(X_i) = (f/r, \mathit{nextpc}(S(X_i)), [\,], pv_S(X_i))$,
  $S'(X_j) = (f/r, \mathit{nextpc}(S(X_j)), [\,], pv_S(X_j))$.
- If $X_i.p$ is not a border port and there exists $X_k \in \mathcal{A}_P$ such that $(X_i.p, X_k.r) \in \mathcal{C}$, $am_S(X_k) = waiting$ and $out(r) \in ci_S(X_k)$ then
  $S'(X_i) = (running, pc_S(X_i), [in(p), X_j], pv_S(X_i))$,
  $S'(X_k) = (taken, \mathit{firstpc}(X_k.r), [out(r)], pv_S(X_k))$.

- If $X_i.p$ is not a border port, and there does not exists $X_j \in \mathcal{A}_A$ such that $(X_i.p, X_j.q) \in \mathcal{C}$, $am_S(X_j) = \mathit{waiting}$ and $ci_S(X_j) = [\mathit{out}(q)]$, and there does not exists $X_k \in \mathcal{A}_P$ such that $(X_i.p, X_k.r) \in \mathcal{C}$, $am_S(X_k) = \mathit{waiting}$ and $\mathit{out}(r) \in ci_S(X_k)$ then $S'(X_i) = (\mathit{waiting}, pc_S(X_i), [\mathit{in}(p)], pv_S(X_i))$.

- States of other agents remain unchanged.

The transition t_{in} changes the state in similar way also for a value passing communication. The only difference is that as a result of communication with an active agent or through a border port (the first and second case in the above description), the value of one of the X_i parameters is updated.

The t_{out} transition works in very similar way, t_{in} and t_{out} differ only in the direction of information sending. For example, in the above second case instead of the condition $ci_S(X_j) = [\mathit{out}(q)]$, we use $ci_S(X_j) = [\mathit{in}(q)]$.

If the transition t_{select} is enable in the state S with respect to the agent X_i then the state S' such that $S - t_{select} \rightarrow S'$) is defined as follows:

- If at least one branch of the statement is open then $S'(X_i) = (\mathit{running}, \mathit{nextpc}(S(X_i)), [\,], pv_S(X))$.

- If all branches are closed then $S'(X_i) = (\mathit{waiting}, pc_S(X_i), c, pv_S(X))$, where list c contains names of all ports used in the *select* statement guards together with the directions of parameters transfer. One port may be placed in c twice if both directions are possible.

- States of other agents remain unchanged.

Let us focus on passive agents now. The t_{select} transition is not allowed in procedures.

Steps of passive agents are always considered in the context of an active one. Thus, to define enable transitions for passive agents, it is necessary to consider behaviour of a pair of agents.

Definition 17. Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model with the current state $S = (S(X_1), \ldots, S(X_n))$ and $X_i \in \mathcal{A}_P$, $X_j \in \mathcal{A}_A$ are agents such that: $(X_i.p, X_j.q) \in \mathcal{C}$, $am_S(X_j) = \mathit{running}$, $ci_S(X_j) = [\mathit{in}(q), X_i]$, $am_S(X_i) = \mathit{taken}$ and $ci_S(X_i) = [\mathit{out}(p)]$. A transition $t \in \mathcal{T} - \{t_{select}\}$ (see Table 2) is *enable* in the state S *with respect to* X_i iff $B_{pc(X_i)}(X_i) = \mathcal{N}(t)$.

A similar definition can be given for agents $X_i \in \mathcal{A}_P$, $X_j \in \mathcal{A}_A$ such that $(X_j.q, X_i.p) \in \mathcal{C}$.

Assume $X_i \in \mathcal{A}_P$, $X_j \in \mathcal{A}_A$ are agents that fulfil requirements from Definition 17.

If a transition $t \in \{t_{if}, t_{loop}\}$ is enable in the state S with respect to the agent X_i then the state S' such that $S - t \rightarrow S'$ is defined as follows:

- $S'(X_i) = (\mathit{taken}, \mathit{nextpc}(S(X_i)), ci_S(X_i), pv_S(X_i))$.

- States of other agents remain unchanged.

Transitions t_{null} and t_{jump} work in the same way if the current step is not the last procedure step. Otherwise the state S' such that $S - t \rightarrow S'$ is defined as follows:

- $S'(X_i) = (\mathit{waiting}, 0, c, pv_S(X_i))$, where c contains names of all accessible ports (procedures) together with the directions of parameters transfer.

- $S'(X_j) = (f/r, \mathit{nextpc}(S(X_j)), [\,], pv_S(X_j))$.

- States of other agents remain unchanged.

The t_{exec} transition changes the state in similar way like t_{null} but the value of one of the X_i parameters is updated. The t_{start} transition changes the state of X_i and X_j in same way as t_{null} but the state of the active agent X_k that is the argument of the *start* statement is also changed in same manner as previously described for t_{start} with respect to an active agent.

If the t_{exit} transition is enable in the state S with respect to the agent X_i then the state S' such that $S - t_{exit} \rightarrow S'$) is defined as follows:

- $S'(X_i) = (\mathit{waiting}, 0, c, pv_S(X_i))$, where c contains names of all accessible ports (procedures) together with the directions of parameters transfer.

- $S'(X_j) = (f/r, \mathit{nextpc}(S(X_j)), [\,], pv_S(X_j))$.

- States of other agents remain unchanged.

There are three cases we have to consider, while discussing transitions t_{in} and t_{out}. The first one corresponds to a communication with an active agent that called a procedure, the second case corresponds to a communication through a border port and the third one to calling a procedure of another passive agent.

Suppose that X_i realises an input procedure. If the t_{in} transition is enable in the state S with respect to the agent X_i, the port $X_i.p$ is the argument of the *in* statement (a communication with the active agent X_j), no value is sent through the port and the current step is not the last procedure step then the state S' such that $S - t_{in} \rightarrow S'$ is defined as follows:

- $S'(X_i) = (\mathit{taken}, \mathit{nextpc}(S(X_i)), ci_S(X_i), pv_S(X_i))$.

- States of other agents remain unchanged.

The t_{in} transition works in the same way also for a communication through a border port $X_i.p'$. The transition t_{in} changes the state in similar way also for a value passing communication. The only difference is that as a result of a communication with an active agent or through a border port (the first and the second case), the value of one of the X_i parameters is updated.

If the t_{in} transition is enable in the state S with respect to the agent X_i, the port $X_i.p'$ is the argument of the *in* statement and $X_i.p'$ is not a border port then the state S' such that $S - t_{in} \to S'$ is defined as follows:

- If there exists $X_k \in \mathcal{A}_P$ such that $(X_i.p', X_k.r) \in \mathcal{C}$, $am_S(X_k) = \mathit{waiting}$ and $\mathit{out}(r) \in ci_S(X_k)$ then $S'(X_i) = (\mathit{taken}, pc_S(X_i), [\mathit{out}(p), \mathit{in}(p')], pv_S(X_j))$.
  $S'(X_j) = (\mathit{running}, pc_S(X_i), [\mathit{in}(p), X_j, X_k], pv_S(X_i))$,
  $S'(X_k) = (\mathit{taken}, \mathit{firstpc}(X_k.r), [\mathit{out}(r)], pv_S(X_j))$.
- If there does not exists $X_k \in \mathcal{A}_P$ such that $(X_i.p, X_k.r) \in \mathcal{C}$, $am_S(X_k) = \mathit{waiting}$ and $\mathit{out}(r) \in ci_S(X_k)$ then
  $S'(X_i) = (\mathit{taken}, pc_S(X_i), [\mathit{in}(p), \mathit{out}(p')], pv_S(X_i))$.
  $S'(X_j) = (\mathit{waiting}, pc_S(X_i), [\mathit{in}(p), X_j], pv_S(X_i))$.
- States of other agents remain unchanged.

In such a case, even if the current step in the last procedure block step then it is not the last procedure $X_i.p$ step. The last step of the $X_k.r$ procedure is now treated as the last step of the $X_i.p$ procedure. Of course, if the $X_k.r$ procedure calls another one, we follow the relationships.

The t_{out} transition works in very similar way, t_{in} and t_{out} differ only in the direction of information sending.

In connection with the above description of transitions it worth to emphasize the difference between two types of communication in Alvis. 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.

On the other hand, a communication between an active and a passive agent can be initialised only by the former. Any procedure in Alvis uses only one either input or output parameter (or signal in case of parameterless communication). In case of an input procedure, an active agent calls the procedure using the *out* statement (and provides the parameter, if any, at the same time). If the corresponding passive agent is in the *waiting* mode and the procedure is accessible, the agent starts it in the active agent context. The passive agent collects the signal/parameter using the *in* statement, but it is not necessary to put the statement as the first procedure step. Similarly, in case of an output procedure, an active agent calls the procedure using the *in* statement. The passive agent returns the result using the *out* statement, but it is not necessary to put the statement as the last procedure step.

For simplicity, we say that an agent $X_i \in \mathcal{A}$ is *enable* in a state S iff there exists a transition $t \in \mathcal{T}$ such that t is *enable* in the state S with respect to X_i.

If $X_i \in \mathcal{A}_A$ and t_{in} or t_{out} is enable in a state S with respect to X_i then we say that X_i *try to communicate* with another agent in the state S. For a passive agent $X_j \in \mathcal{A}_P$ we say that X_j *try to communicate* with another agent in a state S only if t_{in} or t_{out} is enable in S with respect to X_j and the port that is argument of the corresponding statement and the current procedure port differ.

We say that there is a *conflict* between agents $X_i, X_j \in \mathcal{A}$ in a state S iff both agents try to communicate with the same agent. In such case, the step to execute is chosen indeterministically.

For the above considerations we assumed implicitly that all agents have the same priority what is not necessary. If we consider agents with different priorities, a conflict between agents $X_i, X_j \in \mathcal{A}$ exists only if both agents try to communicate with the same agent and they have the same priority. If both agents try to communicate with the same agent but they have different priorities, we say that there is a *potential conflict* between them.

Priorities affect also enabling of the t_{in} and t_{out} transitions. To be enable with respect to an agent X_i a transition must not only satisfy the requirements of Definition 16 or 17 but also it cannot be in a potential conflict between the agent X_i and an agent with a higher priority.

8. LTS graphs

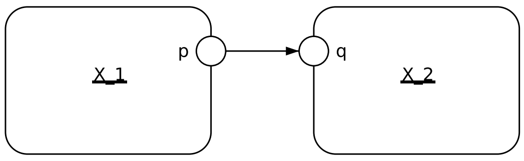
Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model. For a pair of states S, S' we say that S' is *directly reachable* from S iff there exists $t \in \mathcal{T}$ such that $S - t \to S'$. We say that S' is *reachable* from S iff there exist a sequence of states $S^1, \ldots, S^{k+1}$ and a sequence of transitions $t^1, \ldots, t^k \in \mathcal{T}$ such that $S = S^1 - t^1 \to S^2 - t^2 \to \ldots - t^k \to S^{k+1} = S'$. 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 directed graph LTS $= (V, E, L)$, such that $V = \mathcal{R}(S_0)$, $L = \mathcal{T}$, and $E = \{(S, t, S') \colon S - t \to S' \wedge S, S' \in \mathcal{R}(S_0)\}$. In other words, an LTS graph presents all reachable states and transitions among them in the form of the directed graph.

To illustrate the idea of LTS graph let us consider two simple examples of Alvis models. The first model shown in Fig. 11 represents two active agents that communicate one with another. The X_1 agent is a sender and X_2 is a receiver. The LST graph for this model is shown in Fig 12. The graph is another approach to explain the rules of the Alvis communication between active agents.

The second model is presented in Fig. 13. It deals with a communication between an active and a passive or two passive agents. The states in the LTS graph illustrate the way agents states change while such a communication. The most interesting parts of these states are modes and context information lists.

The graphical form of LTS graphs presentation is very useful from users point of view. An LTS graph gen-



```
agent X_1 {
  loop {          -- 1
    out p; }      -- 2
}

agent X_2 {
  loop {          -- 1
    in q; }       -- 2
}
```

Fig. 11. Example 1.

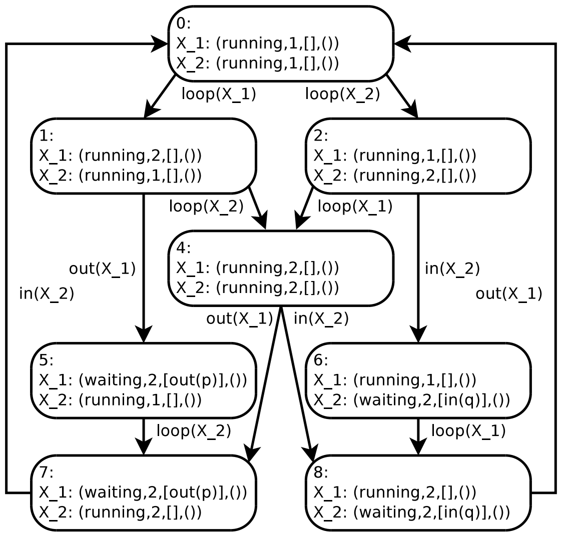


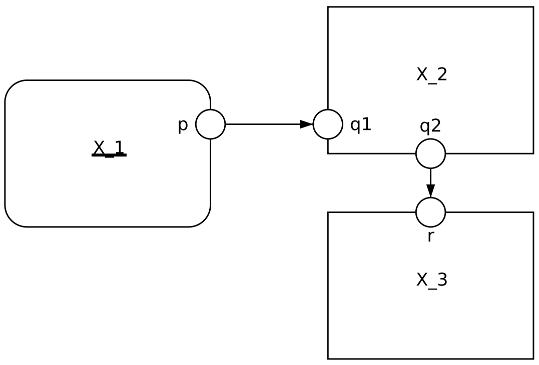
Fig. 12. Example 1 – LTS graph.

erated automatically for a model is stored in a textual file. For verification purposes such graphs are transformed into the *Binary Coded Graphs* (BCG) format. Finally, its properties are verified with the CADP toolbox (Garavel *et al.*, 2007). CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking.

9. Summary

A description of Alvis, a formal language for modelling of concurrent (especially embedded) systems has been presented in the paper. With the α^0 system layer Alvis provides an alternative approach to modelling of such systems and may be more interesting, from the engineering point of view, than formal languages like Petri nets, time automata or process algebras. The main differences between Alvis and formal methods, especially process algebras, are: the syntax that is more user-friendly from the programmers point of view, and the visual modelling language (communication diagrams) that is used to define connections among agents. Furthermore, system layers provides a flexible way to specify an embedded system running environment e.g. a scheduling algorithm.

The language is still under development (especially another predefined system layers). Moreover, a computer software called Alvis Toolkit that supports modelling with Alvis is also under implementation. For more information about the current status of the project visit `http://fm.ia.agh.edu.pl`.



```
agent X_1 {
  loop {                 -- 1
    out p; }             -- 2
}

agent X_2 {
  proc q1 { in q1;       -- 1
            out q2 } -- 2
}

agent X_3 {
  proc r { in r;         -- 1
           null; }       -- 2
}
```

Fig. 13. Example 2.

Acknowledgment

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

References

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

Andre, C. (2003). *Semantics of SyncCharts*, University of Nice-Sophia Antipolis.

Ashenden, P. (2008). *The Designer's Guide to VHDL*, Vol. 3, third edn, Morgan Kaufmann.

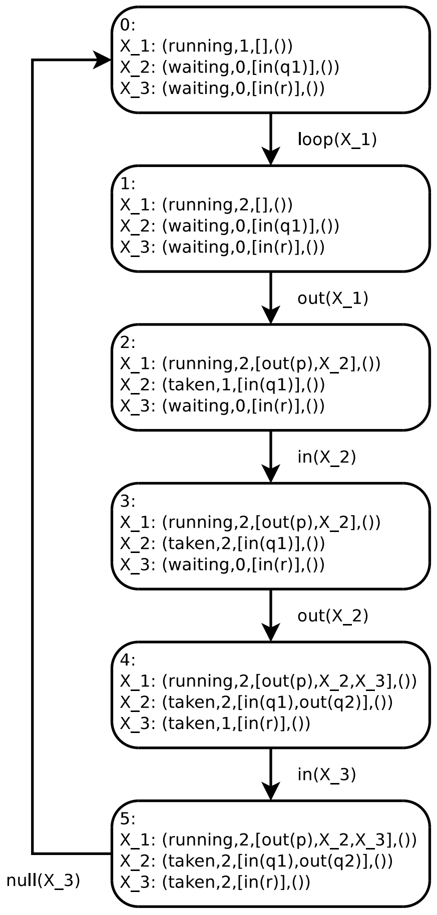

0:
X_1: (running,1,[],())
X_2: (waiting,0,[in(q1)],())
X_3: (waiting,0,[in(r)],())
loop(X_1)
1:
X_1: (running,2,[],())
X_2: (waiting,0,[in(q1)],())
X_3: (waiting,0,[in(r)],())
out(X_1)
2:
X_1: (running,2,[out(p),X_2],())
X_2: (taken,1,[in(q1)],())
X_3: (waiting,0,[in(r)],())
in(X_2)
3:
X_1: (running,2,[out(p),X_2],())
X_2: (taken,2,[in(q1)],())
X_3: (waiting,0,[in(r)],())
out(X_2)
4:
X_1: (running,2,[out(p),X_2,X_3],())
X_2: (taken,2,[in(q1),out(q2)],())
X_3: (taken,1,[in(r)],())
in(X_3)
5:
X_1: (running,2,[out(p),X_2,X_3],())
X_2: (taken,2,[in(q1),out(q2)],())
X_3: (taken,2,[in(r)],())
null(X_3)

Fig. 14. Example 2 – LTS graph.

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

Berry, G. (2000). *The Esterel v5 Language Primer Version v5 91*, Centre de Mathématiques Appliquées Ecole des Mines and INRIA.

Est (2007). *Welcome to SCADE 6.0.*

Fencott, C. (1995). *Formal Methods for Concurrency*, International Thomson Computer Press, Boston, MA, USA.

Garavel, H., Lang, F., Mateescu, R. and Serwe, W. (2007). CADP 2006: A toolbox for the construction and analysis of distributed processes, *Computer Aided Verification (CAV'2007)*, Vol. 4590 of *LNCS*, Springer, Berlin, Germany, pp. 158–163.

ISO (1989). Information processing systems, open systems interconnection LOTOS, *Technical Report ISO 8807*.

Matyasik, P. (2009). *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.

Milner, R. (1989). *Communication and Concurrency*, Prentice-Hall.

Obj (2008). *OMG Systems Modeling Language (OMG SysML).*

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

Palshikar, G. (2001). An introduction to Esterel, *Embedded Systems Programming* **14**(11).

Szpyrka, M. and Matyasik, P. (2008). Formal modelling and verification of concurrent systems with XCCS, *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008)*, Krakow, Poland, pp. 454–458.

Szpyrka, M., Matyasik, P. and Mrówka, R. (2011). Alvis – modelling language for concurrent systems, Studies in Computational Intelligence, Springer-Verlag. (to appear).

Marcin Szpyrka. Prof. Marcin Szpyrka holds a position of associate professor in AGH UST in Krakow, Poland, Department of Automatics. He has a MSc in Mathematics and PhD and DSc (habilitation) in Computer Science. He is the author of over 70 publications, from the domains of formal methods, software engineering and knowledge engineering. Among other things, he is author of 3 books on Petri nets. His fields of interest also include theory of concurrency and functional programming. He is currently leader of the Alvis project. He also worked out the idea of RTCP-nets (real time coloured Petri nets) for modelling real-time embedded systems.

Piotr Matyasik. Assistant Professor at AGH University of Science and technology, Department of Automatics. He has MSc in Automatics and PhD in Computer Science. His interest covers formal methods, robotics, artificial intelligence and programming languages. Currently involved in Alvis project. He is the author of publications on artificial intelligence, formal methods, embedded systems and software engineering.

Rafał Mrówka. Dr. Rafał Mrówka holds a position of assistant professor in AGH UST in Krakow, Poland, Department of Automatics. He has a MSc in Automatics and PhD in Computer Science. His fields of interest include software engineering, formal methods, robotics and programming languages. He is currently involved in Alvis project.

Leszek Kotulski. Prof. Leszek Kotulski holds a position of associate professor in AGH UST in Krakow, Poland, Department of Automatics. He has a MSc, PhD and DSc (habilitation) in Computer Science. He is the author of over 80 publications, from the domains of formal methods, concurrent programming and software engineering His fields of special interest include distributed graph transformations and agents system. He is currently leader of the GRADIS project.

Krzysztof Balicki. Krzystof Balicki is employed in the Institute of Mathematics at Rzeszow University. He has a MSc in Mathematics and actually works on PhD thesis in Computer Science. His interests focus on modelling and analysis of concurrent systems with the use of various formal methods e.g. Petri nets, process algebras. He is also keen on applying mathematical techniques in the field of Computer Science.