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

amcs

FORMAL MODELLING AND VERIFICATION OF CONCURRENT SYSTEMS WITH ALVIS

MARCIN SZPYRKA, PIOTR MATYASIK, RAFAŁ MRÓWKA, LESZEK KOTULSKI

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

The paper presents a formal description of a subset of the Alvis language designed for the modelling and formal verification of concurrent systems. Alvis has been defined as a kind of a happy medium between formal and practical modelling languages. It combines possibilities of a 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 α^0, described in the paper, makes Alvis similar to other formal languages like Petri nets, process algebras, time automata, etc. Alvis with the 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* (Szpyrka, Matyasik and Mrówka, 2011), (Szpyrka, 2012) with the α^0 system layer. Alvis is a successor of the XCCS modelling language (Balicki and Szpyrka, 2009), (Matyasik, 2009), which was an extension of the CCS process algebra (Milner, 1989), (Fencott, 1995), (Aceto *et al*., 2007).

An Alvis model is a system of agents that usually run concurrently, communicate one with another, compete for shared resources etc. Agents are divided into three groups: *active agents* can be treated as processing nodes, *passive agents* represent shared resources and *hierarchical agents* represent submodels.

Alvis combines a 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. 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. Choosing one of accessible system layers entails choosing a scheduling algorithm and hardware architecture 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. Similar to other formal methods, Alvis provides a possibility of formal verification of models. An Alvis model can be transformed into a *labelled transition system* (LTS). After encoding such a 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. Due to the fact that Alvis has been worked out especially for modelling of embedded and real-time systems, 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 veri-

fication 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 anal-

ysis 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 system layer makes Alvis a formal modelling language for concurrent systems. The layer is based on the following assumptions:

- Each active agent has access to its own processor and performs its statements as soon as possible.
- The predefined α^0 scheduler function is called after each statement automatically and makes agents running as soon as possible.
- In case of conflicts, agents priorities are taken under consideration. If two or more agents with the same highest priority compete for the same resources, the system works indeterministically.

  A *conflict* is a state when two or more active agents try to call a procedure of the same passive agent or two or more active agents try to communicate with the same active agent.

In this paper, we will consider only Alvis statements that do not use time explicitly and do not use border ports (a communication with an embedded system environment (Szpyrka, Kotulski and Matyasik, 2011)). This subset of Alvis can be used for modelling concurrent systems instead of Petri nets, process algebras and similar formal languages. The main advantage of the presented approach is its form that seams to be significantly more suitable for engineers than the mentioned formalisms.

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 communica-

tion diagram is a hierarchical graph that nodes may represent both kind of 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 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 subsystem represented by the agent. Such a substituted agent is called a *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 $\mathbf{A}$ 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 active and passive agent from a communication diagram is described in the corresponding code layer and vice versa.

An agent can communicate with other agents through *ports*. Each agent port must have a unique identifier 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 port p of agent X.

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

- $\mathcal{P}_{in}(X)$ denotes the set of *input ports* of agent X. An input port is a 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 port with at least one one-way connection leading from this port or with at least one two-way connection.
- $\mathcal{P}_{unc}(X) = \mathcal{P}(X) - (\mathcal{P}_{in}(X) \cup \mathcal{P}_{out}(X))$ denotes the set of *unconnected ports*.
- $\mathcal{P}_{proc}(X)$ 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.

Let $\mathcal{N}(X)$ denote the set of ports names of agent X, and $\mathcal{N}(W) = \sum_{X\in W}\mathcal{N}(X)$. For example, if a diagram contains only agents: X_1 with port p and X_2 also with port p, then $\mathcal{P} = \{X_1.p, X_2.p\}$, and $\mathcal{N}(\mathcal{P}) = \{p\}$.

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

$$\forall X \in \mathcal{A}\colon (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C} = \emptyset, \tag{1}$$

$$\mathcal{P}_{proc} \cap \mathcal{P}_{in} \cap \mathcal{P}_{out} = \emptyset, \tag{2}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_A) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow \Rightarrow q \in \mathcal{P}_{proc}, \tag{3}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_A)) \cap \mathcal{C} \Rightarrow \Rightarrow p \in \mathcal{P}_{proc}, \tag{4}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow \Rightarrow (p \in \mathcal{P}_{proc} \wedge q \notin \mathcal{P}_{proc}) \vee \vee (q \in \mathcal{P}_{proc} \wedge p \notin \mathcal{P}_{proc}). \tag{5}$$

Each element belonging to $\mathcal{C}$ is called a *connection* or a *communication channel*.

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

Let us focus on the conditions from Definition 1.

- (1) – A connection cannot be defined between two ports of the same agent.
- (2) – Procedure ports are either input or output ones.
- (3), (4) – A connection between an active and a passive agent must be a procedure call.
- (5) – 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 – We can make them active later with

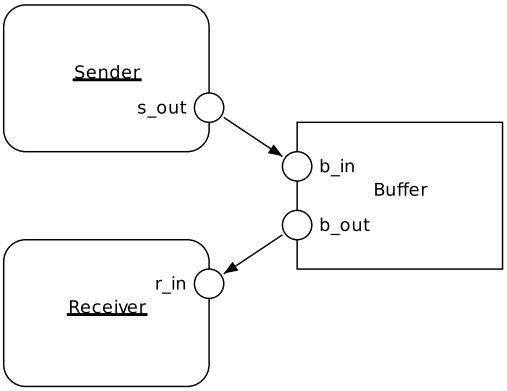


Fig. 1. Sender-Receiver system with buffer – communication diagram.

the *start* statement. Names of agents that are initially activated (represent running processes) are underlined in a communication diagram.

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 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 (Szpyrka, 2012). 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. A communication diagram can be treated as a module and represented by a single hierarchical agent at the higher level. Hierarchical agents are not defined in the model code layer. We divide ports of hierarchical agents into three subsets based on the connections defined in the model: $\mathcal{P}_{in}(X)$, $\mathcal{P}_{out}(X)$, and $\mathcal{P}_{unc}(X)$. Ports of hierarchical agents cannot be defined as procedure ones.

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

- $\mathcal{A}^i = \{X_1^i, \ldots, X_n^i\}$ is the set of *agents* with subsets of *active agents* $\mathcal{A}_A^i$, *passive agents* $\mathcal{A}_P^i$, and *hierarchical agents* $\mathcal{A}_H^i$, such that $\mathcal{A}^i = \mathcal{A}_A^i \cup \mathcal{A}_P^i \cup \mathcal{A}_H^i$, and $\mathcal{A}_A^i$, $\mathcal{A}_P^i$, $\mathcal{A}_H^i$ are pairwise disjoint.
- $\mathcal{C}^i \subseteq \mathcal{P}^i \times \mathcal{P}^i$, where $\mathcal{P}^i = \sum_{X \in \mathcal{A}^i} \mathcal{P}(X)$, is the *communication relation*, such that:

$$\forall X \in \mathcal{A}^i \colon (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C}^i = \emptyset, \tag{6}$$

$$\mathcal{P}_{proc}^i \cap \mathcal{P}_{in}^i \cap \mathcal{P}_{out}^i = \emptyset, \tag{7}$$

$$\mathcal{P}_{proc}^i \cap \mathcal{P}(\mathcal{A}_H^i) = \emptyset, \tag{8}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_A^i) \times \mathcal{P}(\mathcal{A}_P^i)) \cap \mathcal{C}^i \Rightarrow \Rightarrow q \in \mathcal{P}_{proc}^i, \tag{9}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_P^i) \times \mathcal{P}(\mathcal{A}_A^i)) \cap \mathcal{C}^i \Rightarrow \Rightarrow p \in \mathcal{P}_{proc}^i, \tag{10}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_P^i) \times \mathcal{P}(\mathcal{A}_P^i)) \cap \mathcal{C}^i \Rightarrow \Rightarrow (p \in \mathcal{P}_{proc}^i \wedge q \notin \mathcal{P}_{proc}^i) \vee \vee (q \in \mathcal{P}_{proc}^i \wedge p \notin \mathcal{P}_{proc}^i), \tag{11}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_P^i) \times \mathcal{P}(\mathcal{A}_H^i)) \cap \mathcal{C}^i \Rightarrow \Rightarrow (q,p) \notin \mathcal{C}^i, \tag{12}$$

$$(p,q) \in (\mathcal{P}(\mathcal{A}_H^i) \times \mathcal{P}(\mathcal{A}_P^i)) \cap \mathcal{C}^i \Rightarrow \Rightarrow (q,p) \notin \mathcal{C}^i. \tag{13}$$

  Each element of the relation $\mathcal{C}^i$ is called a *connection* or a *communication channel*.
- $\sigma^i \colon \mathcal{A}_A^i \to \{False, True\}$ is the *start function* that points out initially activated agents.

Let us focus on the conditions from Definition 2.

- (6) – A connection cannot be defined between two ports of the same agent.
- (7) – Procedure ports are either input or output ones.
- (8) – Hierarchical agents cannot have procedure ports.
- (9), (10) – A connection between an active and a passive agent must be a procedure call.
- (11) – A connection between two passive agents must a procedure call from a non-procedure port.
- (12), (13) – A connection between a hierarchical and a passive agent must be a one-way connection.

The above definition treats hierarchical agents almost like active ones. However, connections with ports of hierarchical agents can make some substitution of pages illegal, i.e. after the transformation of a hierarchical diagram into the equivalent flat one, all connections must satisfy the conditions (1)-(5).

[0]We will use two notations to denote ports in equations. 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. $X.p$.

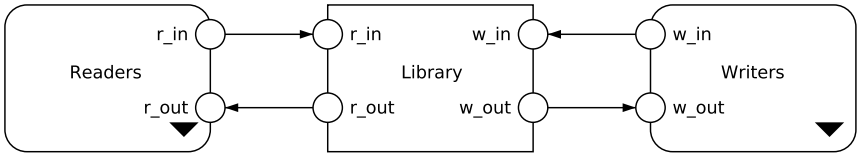


Fig. 2. Readers-Writers system – top level page of the communication diagram.

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 $X \in \mathcal{A}_H^i$ be given and let $\mathcal{P}_{join}^X(D^j)$ denotes the set of all *join ports* of the page D^j with respect to X, i.e.:

$$\mathcal{P}_{join}^X(D^j) = \{X_k^j.p \in \mathcal{P}(D^j)\colon p \in \mathcal{N}(\mathcal{P}(X))\}. \quad (14)$$

In other words, $\mathcal{P}_{join}^X(D^j)$ is the set of all ports from the page D^j that names are the same as those of the hierarchical agent X.

An attempt to assign a page D^j to a hierarchical agent X results in the following set of hierarchical communication channels:

$$\begin{aligned}\mathcal{C}_X^j = &\{(X_k^i.p, X_m^j.q)\colon (X_k^i.p, X.q) \in \mathcal{C}^i\} \cup \\ &\cup \{(X_m^j.q, X_k^i.p)\colon (X.q, X_k^i.p) \in \mathcal{C}^i\} \quad (15)\end{aligned}$$

Definition 3. Let a hierarchical agent $X \in \mathcal{A}_H^i$ and a page $D^j = (\mathcal{A}^j, \mathcal{C}^j, \sigma^j)$ be given. Agent X and page D^j satisfy the *simple substitution* requirements, iff

$$card(\mathcal{P}(X)) = card(\mathcal{P}_{join}^X(D^j)), \quad (16)$$

$$\mathcal{P}_{join}^X(D^j) \subseteq \mathcal{P}_{unc}^j, \quad (17)$$

and the page $D' = (\mathcal{A}', \mathcal{C}', \sigma')$, where

- $\mathcal{A}' = \mathcal{A}^i \cup \mathcal{A}^j - \{X\}$,
- $\mathcal{C}' = \mathcal{C}^i \cup \mathcal{C}^j \cup \mathcal{C}_X^j - \{(p,q)\colon p \in \mathcal{P}(X) \vee q \in \mathcal{P}(X)\}$,
- $\sigma'(Y) = \begin{cases} \sigma^i(Y)\colon Y \in \mathcal{A}_A^i \\ \sigma^j(Y)\colon Y \in \mathcal{A}_A^j \end{cases}$,

satisfies all conditions from Definition 2.

If instead of the condition (16), the following condition is satisfied:

$$card(\mathcal{P}(X)) < card(\mathcal{P}_{join}^X(D^j)), \quad (18)$$

we say that agent X and page D^j satisfy the *extended substitution* requirements.

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 $\mathcal{A}_H = \bigcup_{i=1,\ldots,k} \mathcal{A}_H^i$, is the *substitution function*, such that:
  1. γ is an injection.
  2. For any $X \in \mathcal{A}_H$, X and $\gamma(X)$ satisfy the requirements of the simple or extended substitution.
  3. Labelled directed graph $\mathcal{G} = (\mathcal{D}, E, \mathcal{A}_H)$ where $(D^i, X_k^i, D^j) \in E$ iff $\gamma(X_k^i) = D^j$ 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 set of pages, 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:

- $\mathcal{A}_A = \bigcup_{i=1,\ldots,k} \mathcal{A}_A^i$,
- $\mathcal{A}_P = \bigcup_{i=1,\ldots,k} \mathcal{A}_P^i$,
- $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$,
- $\sigma\colon \mathcal{A}_A \to \{False, True\}$ and $\forall i = 1,\ldots,k\ \forall X_j^i \in \mathcal{A}_A\colon \sigma(X_j^i) = \sigma^i(X_j^i)$,
- $\mathcal{C} = \bigcup_{i=1,\ldots,k} \mathcal{C}^i \cup \bigcup_{X \in \mathcal{A}_H \wedge \gamma(X) = D^j} \mathcal{C}_X^j$.

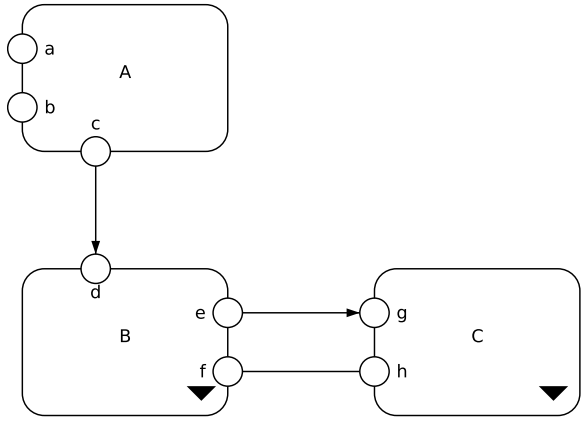


Fig. 3. Page D^1.

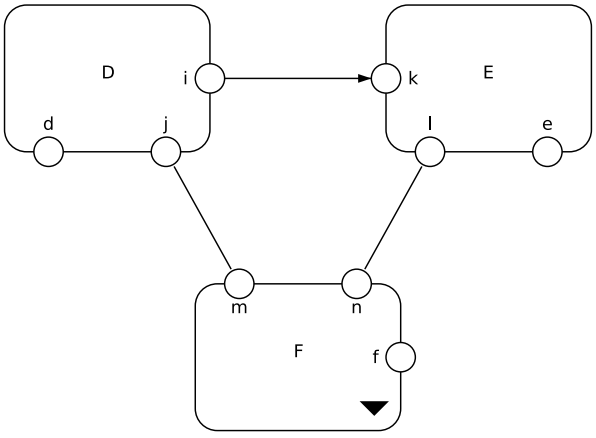


Fig. 4. Page D^2.

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

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

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

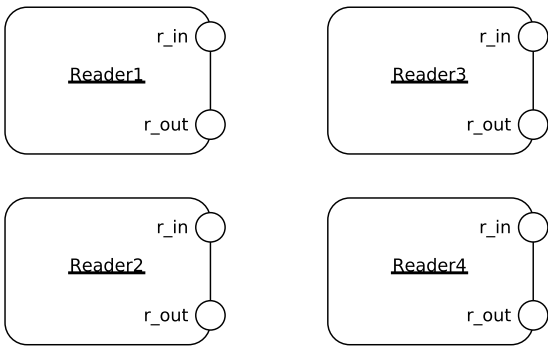


Fig. 5. Readers-Writers system – page *Readers*.

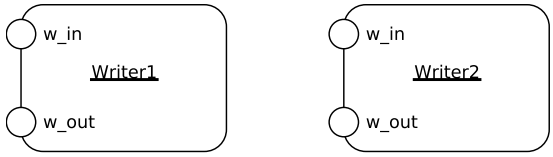


Fig. 6. Readers-Writers system – page *Writers*.

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.

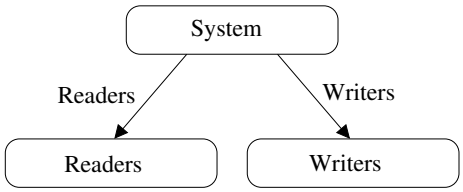


Fig. 7. Page hierarchy graph

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

- $\mathcal{P}(Readers) = \{Readers.r\_in, Readers.r\_out\}$
- $\mathcal{P}_{join}^{Readers}(pReaders) = \{Reader1.r\_in, Reader1.r\_out, Reader2.r\_in, Reader2.r\_out, Reader3.r\_in, Reader3.r\_out, Reader4.r\_in, Reader4.r\_out\}$
- $\mathcal{N}(\mathcal{P}(Readers)) = \{r\_in, r\_out\} = \mathcal{N}(\mathcal{P}_{join}^{Readers}(pReaders))$

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:

$$\pi\colon \bigcup_{X\in\mathcal{A}_H} \mathcal{P}(X) \to 2^{\mathcal{P}}. \tag{19}$$

The function π satisfies the following conditions:

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \pi(X.p) \subseteq \mathcal{P}^X_{join}(\gamma(X)), \tag{20}$$

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \mathcal{N}(\pi(X.p)) = \{p\}. \tag{21}$$

If a communication diagram contains only simple substitutions, then the function (19) takes the simplified form:

$$\pi\colon \bigcup_{X\in\mathcal{A}_H} \mathcal{P}(X) \to \mathcal{P}, \tag{22}$$

and the condition (20):

$$\forall X \in \mathcal{A}_H\ \forall X.p \in \mathcal{P}(X)\colon \pi(X.p) \in \mathcal{P}^X_{join}(\gamma(X)). \tag{23}$$

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) the 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* (Kotulski and Szpyrka, 2011). 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. $\forall X, Y \in \mathcal{F} \subseteq \mathcal{A}\colon X \neq Y \Rightarrow X \not\succeq Y$,
2. $\forall X \in \mathcal{A} - \mathcal{A}_H\ \exists Y \in \mathcal{F}\colon Y \succeq X$,
3. $\mathcal{C}' = \{(X.p, Y.q) \in \mathcal{C}\colon X, Y \in \mathcal{F}\}$,
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. $\mathcal{C}^* = \{(Z.p, Z'.q) \in \mathcal{C}\colon Z, Z' \in \mathcal{F}^*\}$,
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. $\mathcal{C}^* = \{(Z.p, Z'.q) \in \mathcal{C}\colon Z, Z' \in \mathcal{F}^*\}$,
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.
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 *System* (presented in Fig. 2) as a primary page is a flat representation of the hierarchical graph presented in Fig. 7 with pages *Readers* and *Writers* (presented appropriately in Fig. 5 and Fig. 6) with the

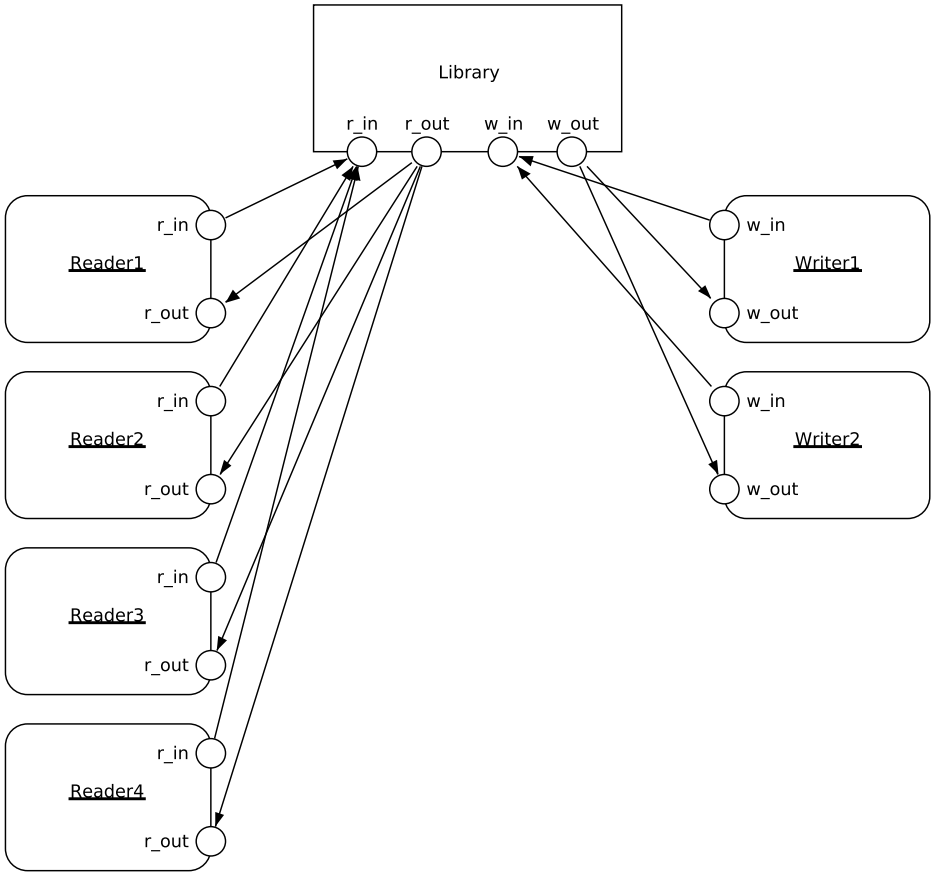


Fig. 9. Application of the extended substitution.

extended substitution mechanism. Flat representation generated by the composition of the analysis operations $\mathrm{AN}(H, \mathrm{AN}(H, System, Readers), Writers)$ 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_j^i, it has been removed a connection between ports $X_j^i.p$ and $X_n^i.q$, then we add similar connections between port $X_n^i.q$ and all ports from the set $\pi(X_j^i.p)$.

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

$$\forall X \in \mathcal{A}\ \exists Y \in \mathcal{F}\colon X \succeq Y. \tag{24}$$

Such a maximal flat representation does not contain hierarchical agents.

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,
- to define functions for data manipulation,
- to specify an embedded system environment,
- to define behaviour of individual agents.

A detailed description of the Alvis statements can be found in (Szpyrka, Matyasik and Mrówka, 2011) and at the Alvis project web site (Szpyrka, 2012). As we consider untimed Alvis models with the α^0 system layer and without border ports, the set of allowed statements is given in Table 1. To simplify the syntax presentation, 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), and `expr` stands for an expression.

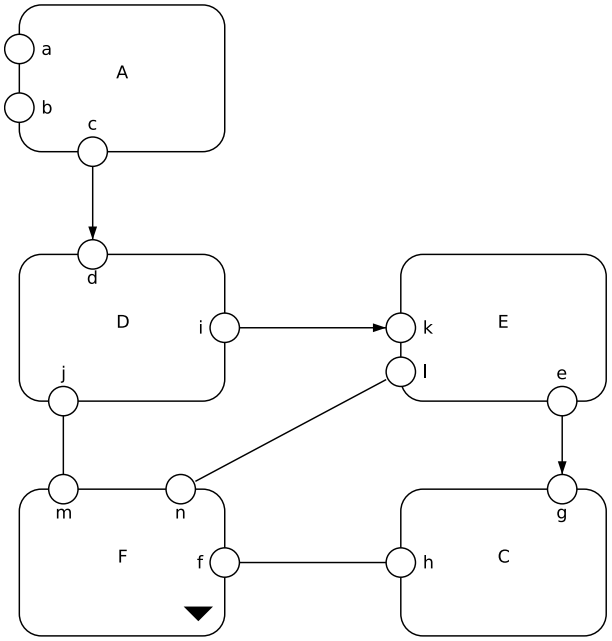


Fig. 8. Application of the simple substitution.

6. Models

An Alvis model is defined 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.

It should be underlined that a syntactically correct code layer means also that the following condition for ports is also satisfied. 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 $(X.p, X'.p') \in \mathcal{C}$.

As it was already shown, one can consider the maximal flat representation of a communication diagram instead of a hierarchical model. Thus, from now on, we will consider only $\overline{\mathbf{A}} = (D, B, \alpha^0)$ models. To define a state of an Alvis model, we need to define a state of a single agent.

Table 1. Alvis statements allowed in untimed models with α^0 system layer

Statement	Description
`exec x = expr`	Evaluates the expression and assigns the result to the parameter; the *exec* keyword can be omitted.
`exit`	Terminates an active agent or a passive agent procedure.

`if (g1) {...}`
`elseif (g2) {...}`
`elseif (g3) {...}`
`...`
`else {...}`
	Conditional statement.

`in p` `in p x`	Collects a signal/value through port p.
`jump label`	Transfers the control to the line of code identified with the *label*.
`loop {...}` `loop (g) {...}`	Infinite loop. Repeats execution of the contents while the guard if satisfied..
`null`	Empty statement.
`out p` `out p x`	Sends a signal/value through the port p.
`proc (g) p {...}`	Defines the procedure for port p of a passive agent. The guard is optional.

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

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

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

$$S(X) = (am(X), pc(X), ci(X), pv(X)), \qquad (25)$$

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. 10.

finished – The mode means that an agent has finished its work or it has been terminated using the *exit* statement.

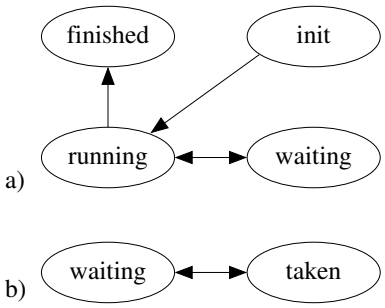


Fig. 10. Possible transitions among modes: a) active agents, b) passive agents.

init – This 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 – The mode means that an agent is performing one of its statements.

taken – The mode means that one of the passive agent procedures has been called and the agent is executing it.

waiting – For passive agents, the mode 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.

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 2.

- 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 *if* statement iff the next statement to be executed is the evaluating of the guard and possibly 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.
- 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 last executed statement is *in* or *out* and the agent is waiting for the communication to be completed (either with an active or a passive agent).

Table 2. Relationships between the mode and the program counter of an agent

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

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 3. If an agent is in the *init* or *finished* mode, its context information list is empty.

The *parameters values list* contains the current values of the agent parameters.

A model state is sequence (list) of all agents states.

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

$$S = (S(X_1), \ldots, S(X_n)). \tag{26}$$

Definition 15. The *initial state* of a model $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is a tuple S_0 as given in (26), where:

- $am(X) = running$, 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 $running$ 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.

Table 3. Relationships between the mode and the context information list of an agent

agent X	$am(X)$	$ci(X)$ entry	description
active	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
active	waiting	$in(a)$, $in(a\|T)$	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
		$out(a)$, $out(a\|T)$	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
passive	waiting	$in(a)$	input procedure $X.a$ is accessible
		$out(a)$	output procedure $X.a$ is accessible

Steps performed by a model are described using the *transition* idea. The set of all possible transitions for the considered Alvis models is given in Table 4.

Table 4. Set of transitions

	Symbol	Description
1	t_{exec}	performs an evaluation and assignment
2	t_{exit}	terminates an agent or a procedure
3	t_{if}	enters an if statement
4	t_{in}	performs communication (input side)
5	t_{jump}	jumps to a label
6	t_{loop}	enters a *while* or *infinite* loop
7	t_{null}	performs an empty statement
8	t_{out}	performs communication (output side)
9	t_{select}	enters a select statement
10	t_{start}	starts an inactive agent
11	t_{io}	performs communication (both sides)

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 agent X 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 $\exists X \in \mathcal{A}, \exists i \in \{1, \ldots, card(B(X))\}\colon B_i(X) = start$.

Let us focus on the *step* idea. It is necessary to distinguish between code statements and 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. For a given statement s, let $stepno(s)$ denote the number of the step related to s. For multi-step statements, $stepno(s)$ denotes the number of the step connected with entering the statement interior.

```
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 1. Steps counting in Alvis code

Let us consider the piece of code shown in Listing 1. 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 need a function that determines 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 statement* means that the statement is the last one in the block and is followed by the closing curly bracket. Depending on the surrounding 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).

Let us focus on code statements first. The $nextst$ function (*next statement*) is used to determine the successor statement for a given one. The function returns *empty statement* if there is no a successor statement for the considered one. The number of the *empty statement* is equal to 0. This recursive function is based on the following rules:

1. If s is a *jump* statement then $nextst(s)$ is the first statement after the *jump* statement label.
2. If s is an *exit* statement then $nextst(s)$ is the empty statement.
3. If $s \in \{exec, if, in, loop, null, out, select, start\}$ and s is not the last block statement then $nextst(s)$ is the statement that follows s in the code layer.
4. If $s \in \{exec, if, in, loop, null, out, select, start\}$ and s is the last main block statement or the last procedure block then $nextst(s)$ is the empty statement.
5. If $s \in \{exec, if, in, loop, null, out, select, start\}$ and s is the last *if* (*loop*, *select*) block statement then $nextst(s) = nextst(s')$, where s' is the surrounding *if* (*loop*, *select*) statement.

A graph representation of the *nextst* function for the code presented in Listing 1 is shown in Fig. 11. For example, the next statement for the *exec* statement number 6 is the *if* statement (statement number 7).

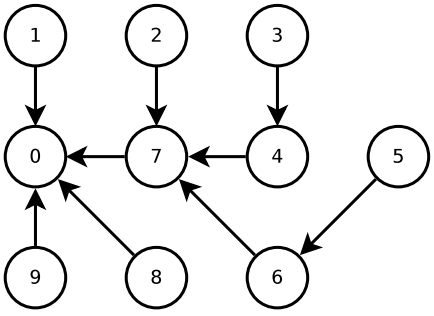


Fig. 11. Graph representation of the *nextst* function for the code presented in Listing 1.

A similar $nextpc$ (next program counter) function determines the number of the next step (the next program counter for an agent). Similarly, we use concepts like *last block step*, *last if block step*, etc. to point out the last step inside a given code 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 1).

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 for the current step t:

1. If $t = exit$ then $nextpc(S(X)) = 0$.
2. If $t \in \{exec, in, null, out, start\}$ then:
   - if t is not the last block step then: $nextpc(S(X)) = pc_S(X) + 1$;
   - if t is the last main block step or the last procedure block step then $nextpc(S(X)) = 0$;
   - if t is the last *loop* block step then: $nextpc(S(X)) = stepno(s)$, where s is the *loop* statement;
   - if t is the last branch block step then: $nextpc(S(X)) = stepno(nextst(s))$, where s is the surrounding *select* statement.
   - if t is the last *if* block step then: $nextpc(S(X)) = stepno(nextst(s))$, where s is the surrounding *if* statement.
3. If $t = jump$ step then $nextpc(S(X))$ returns the number of the first step after the corresponding label.
4. If $t = if$ then:
   - $nextpc(S(X))$ is equal to the number of the first step inside the chosen *if* block, if such a block has been chosen;
   - $nextpc(S(X)) = stepno(nextst(s))$, where s is the *if* statement otherwise.
5. If $t = loop$ then:
   - if the loop guard is satisfied or for an infinite loop $nextpc(S(X)) = pc_S(X) + 1$;
   - if the loop guard is not satisfied then: $nextpc(S(X)) = stepno(nextst(s))$, where s is the *loop* statement.
6. If $t = select$ then $nextpc(S(X))$ returns the number of the first step inside the chosen branch block.

A graph representation of the *nextpc* function for the code presented in Listing 1 is shown in Fig. 12.

To describe a ci list modifications we will use the following operators:

- $e \in ci$ – returns *true* if the element e belongs to the ci list and *false* otherwise.
- $ci \oplus e$ – if $e \notin ci$ then adds the element e at the end of the list.
- $ci \ominus e$ – if $e \in ci$ then removes the element e from the list.

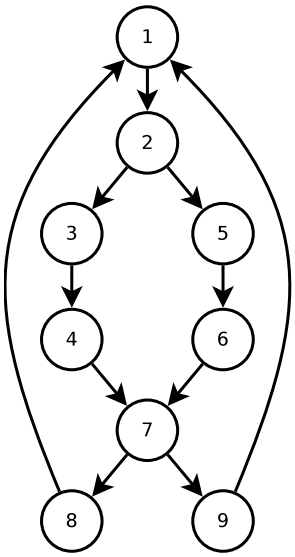


Fig. 12. Graph representation of the *nextpc* function for the code presented in Listing 1.

7. Transitions

We will consider behaviour of Alvis models at the level of detail of single steps. Each of transitions presented in Table 4 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. Firstly, we will focus on active agents only.

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 $t \in \mathcal{T}$ is *enable* in the state S *with respect to* X_i (denoted as $S{-}t(X){\rightarrow}$) iff the following requirement holds:

$$am(X_i) = running \ \wedge \ B_{pc(X_i)}(X_i) = \mathcal{N}(t). \quad (27)$$

The fact that a transition t is enable in a state S with respect to an agent X and the state S' that is the result of executing t in S will be denoted by $S{-}t(X){\rightarrow}S'$. If case of four transitions, 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 send/collected value. If necessary, the special *Empty* type will be used to denote a valueless communication.
- $S{-}t_{io}(X.p, Y.q, T){\rightarrow}S'$, where $X.p$ and $Y.q$ are the input and output ports for the communication respectively and T is the type of the transferred value.

This section describes the states that are results of executing all possible steps. We will limit the definitions to description of agents, which states change. Agent, which states remain unchanged are omitted in the description.

Let $pv_S(X)|_{v=a}$ denote the list of parameters values $pv_S(X)$, but with the parameter v assigned to a new value a. If $X \in \mathcal{A}_A$, $S{-}t_{exec}(X){\rightarrow}S'$, and a parameter v is assign a value a with the corresponding *exec* statement, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X)|_{v=a})$, if $nextpc(S(X)) \neq 0$,
- $S'(X) = (finished, 0, [\,], pv_{S'}(X))$, if $nextpc(S(X)) = 0$.

If $X \in \mathcal{A}_A$ and $S{-}t_{exit}(X){\rightarrow}S'$, then:

- $S'(X) = (finished, 0, [\,], pv_S(X))$,

If $t \in \{if, loop, null\}$, $X \in \mathcal{A}_A$ and $S{-}t(X){\rightarrow}S'$ then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$, if $nextpc(S(X)) \neq 0$.
- $S'(X) = (finished, 0, [\,], pv_S(X))$, if $nextpc(S(X)) = 0$.

If $X \in \mathcal{A}_A$ and $S{-}t_{jump}(X){\rightarrow}S'$, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$.

If $X \in \mathcal{A}_A$ and $S{-}t_{select}(X){\rightarrow}S'$, then:

- If at least one branch of the statement is open then $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$.
- If all branches are closed then $S'(X_i) = (waiting, pc_S(X_i), ci_S(X) \oplus guard, pv_S(X))$.

If $X, Y \in \mathcal{A}_A$ and $S{-}t_{start}(X, Y){\rightarrow}S'$, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X))$, if $nextpc(S(X)) \neq 0$.
- $S'(X) = (finished, 0, [\,], pv_S(X))$, if $nextpc(S(X)) = 0$.
- If $am_S(Y) = init$, then $S'(Y) = (running, 1, [\,], pv_S(Y))$, otherwise $S'(Y) = S(Y)$.

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 at least a pair of agents.

Definition 17. Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model with the current state $S = (S(X_1), \ldots, S(X_n))$, $X \in \mathcal{A}$ and $Y \in \mathcal{A}_P$.

- We say that X is *(directly) performing input procedure* $Y.q$ via its port p iff $(X.p, Y.q) \in \mathcal{C}$, $proc(Y.q, p) \in ci_S(X)$ and $am_S(Y) = taken$.

- We say that X is *(directly) performing output procedure* $Y.q$ via its port p iff $(Y.q, X.p) \in \mathcal{C}$, $proc(Y.q, p) \in ci_S(X)$ and $am_S(Y) = taken$.

- We say that X is *performing a procedure* of an agent Y iff X is performing an input or output procedure of Y via one of its ports.

- We say that X is *indirectly performing input (output) procedure* $Y.q$ iff exist $X_k^1, \ldots, X_k^m$, $m > 0$ such that X is performing a procedure of X_k^1, X_k^1 is performing a procedure of $X_k^2, \ldots, X_k^m$ is performing input (output) procedure $Y.q$ via one of its ports. For any passive agent Y performing one of its procedures, $context(Y)$ will denote the active agent X that directly or indirectly performs the procedure.

Definition 18. 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 $X \in \mathcal{A}_A$, $Y \in \mathcal{A}_P$ are agents such that X is directly or indirectly performing input (or output) procedure $Y.q$ via its port p. A transition $t \in \mathcal{T}$ is *enable* in the state S *with respect to* Y iff the following requirement holds:

$$am(X) = running \ \wedge \ B_{pc(Y)}(Y) = \mathcal{N}(t). \tag{28}$$

Performing the *in* or *out* statements may influence more than one agent state. Very often two agents connected with a communication channel perform their communication statements simultaneously. Such a communication is possible only if types of sending and collecting values are the same. Moreover, if a few agents try to communicate at the same time, their priorities are taken into consideration to determine pairs of agents that perform their communication statements simultaneously using the same communication channels. Similarly, if an agents calls an available procedure of a passive agent, states of two agents change (in spite of the fact that only one of them performs a step).

Assume $\overline{\mathbf{A}} = (D, B, \alpha^0)$ is an Alvis model with the current state $S = (S(X_1), \ldots, S(X_n))$. Let $type(X.p)$ denote the type of a procedure p argument for a passive agent X (The *Empty* type can be used if none argument is used.) In case of active agents, $type_S(X.p)$ will denote the type of sent (expected) argument for already performed *out* (*in*) step. Let us define the following set of pairs:

$$\begin{aligned} Comm_S^{AA} = \{(X,Y)\colon X, Y \in \mathcal{A}_A \ \wedge \\ \wedge \ S{-}t_{in}(X.p,T){\rightarrow} \wedge \ S{-}t_{out}(Y.q,T){\rightarrow} \ \wedge \\ \wedge \ (Y.q, X.p) \in \mathcal{C}\} \end{aligned} \tag{29}$$

$$\begin{aligned} Comm_S^{AP} = \{(X,Y)\colon X \in \mathcal{A}_A \wedge Y \in \mathcal{A}_P \ \wedge \\ \wedge \ S{-}t_{in}(X.p,T){\rightarrow} \wedge \ am(Y) = waiting \ \wedge \\ \wedge \ out(q) \in ci(Y) \wedge type(Y.q) = T \ \wedge \\ \wedge \ (Y.q, X.p) \in \mathcal{C}\} \ \cup \\ \cup \ \{(X,Y)\colon X \in \mathcal{A}_A \wedge Y \in \mathcal{A}_P \ \wedge \\ \wedge \ S{-}t_{out}(X.p,T){\rightarrow} \wedge \ am(Y) = waiting \ \wedge \\ \wedge \ in(q) \in ci(Y) \wedge type(Y.q) = T \ \wedge \\ \wedge \ (X.p, Y.q) \in \mathcal{C}\} \end{aligned} \tag{30}$$

$$\begin{aligned} Comm_S^{PP} = \{(X,Y)\colon X, Y \in \mathcal{A}_P \ \wedge \\ \wedge \ S{-}t_{in}(X.p,T){\rightarrow} \wedge \ am(Y) = waiting \ \wedge \\ \wedge \ out(q) \in ci(Y) \wedge type(Y.q) = T \ \wedge \\ \wedge \ (Y.q, X.p) \in \mathcal{C}\} \ \cup \\ \cup \ \{(X,Y)\colon X, Y \in \mathcal{A}_P \wedge S{-}t_{out}(X.p,T){\rightarrow} \ \wedge \\ \wedge \ am(Y) = waiting \wedge in(q) \in ci(Y) \ \wedge \\ \wedge \ type(Y.q) = T \wedge (X.p, Y.q) \in \mathcal{C}\} \end{aligned} \tag{31}$$

$$\begin{aligned} Comm_S^{F} = \{(X,Y)\colon X, Y \in \mathcal{A}_A \wedge \\ \wedge \ S{-}t_{in}(X.p,T){\rightarrow} \ \wedge \\ \wedge \ am(Y) = waiting \wedge out(q) \in ci(Y) \ \wedge \\ \wedge \ type_S(Y.q) = T \wedge (Y.q, X.p) \in \mathcal{C}\} \ \cup \\ \cup \ \{(X,Y)\colon X, Y \in \mathcal{A}_A \wedge S{-}t_{out}(X.p,T){\rightarrow} \ \wedge \\ \wedge \ am(Y) = waiting \wedge in(q) \in ci(Y) \ \wedge \\ \wedge \ type_S(Y.q) = T \wedge (X.p, Y.q) \in \mathcal{C}\} \end{aligned} \tag{32}$$

$$Comm_S^{*} = Comm_S^{AA} \cup Comm_S^{AP} \ \cup \ \cup \ Comm_S^{PP} \cup Comm_S^{F} \tag{33}$$

Next, we divide all agents enable for communication in the state S into two disjoint sets:

$$\begin{aligned} Comm_S^2 = \{X \in \mathcal{A}\colon \exists Y \in \mathcal{A} \ \wedge \\ \wedge \ ((X,Y) \in Comm_S^{AA} \cup Comm_S^{AP} \ \cup \\ \cup \ Comm_S^{PP} \cup Comm_S^{F} \ \vee \\ \vee \ (Y,X) \in Comm_S^{AA})\} \end{aligned} \tag{34}$$

$$\begin{aligned} Comm_S^1 = \{X \in \mathcal{A}\colon (S{-}t_{in}(X.p,T){\rightarrow} \ \vee \\ \vee \ S{-}t_{out}(X.p,T){\rightarrow}) \wedge X \notin Comm_S^2\} \end{aligned} \tag{35}$$

It's hardly possible that all agents from the set $Comm_S^1 \cup Comm_S^2$ can perform they communication steps simultaneously. Usually, agents compete for the same agents and we can observe some conflicts in a model. Let us consider the communication diagram shown in Fig. 13. Suppose, that for a considered state S the following conditions hold:

- $S{-}t_{out}(A.p,T){\rightarrow}$,
- $S{-}t_{out}(B.p,T){\rightarrow}$,
- $S{-}t_{in}(C.p,T){\rightarrow}$,

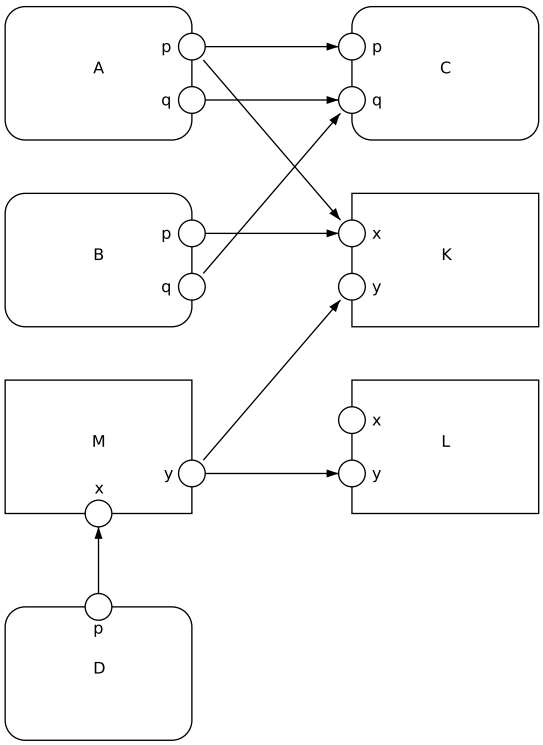


Fig. 13. Communication conflicts

- $am_S(D) = running$,
- $context(M) = D$, $S{-}t_{out}(M.y, T){\rightarrow}$,
- $am_S(K) = waiting$, $ci_S(K) = [in(x), in(y)]$, $type(K.x) = type(K.y) = T$,
- $am_S(L) = waiting$, $ci_S(L) = [in(x), in(y)]$, $type(L.x) = type(L.y) = T$.

Thus, we have:

- $Comm_S^{AA} = \{(C, A)\}$,
- $Comm_S^{AP} = \{(A, K), (B, K)\}$,
- $Comm_S^{PP} = \{(M, K), (M, L)\}$,
- $Comm_S^{F} = \emptyset$,
- $Comm_S^{2} = \{A, B, C, M\}$
- $Comm_S^{1} = \emptyset$.

It easy to see that have conflicts in the state S: agents A and B compete for the procedure $K.x$, and agents B and M compete for an access to agent K.

Alvis uses a reverse priorities range. The code layer priorities range from 0 to 9, where 0 is the higher system priority. From the theoretical point of it is more convenient to use the pr function defined as follows

$$pr(X) = 9 - codePriority(X) \tag{36}$$

Using different agents priorities can eliminate most of potentials conflicts in a model. For example, if $pr(A) > pr(B)$ then there is no conflict between agents A and B, but it does not mean that agent A will perform the procedure $K.x$. Selecting communication steps that can be perform in a given state is based on Algorithm 1.

The output of the algorithm are two sets. The set $Comm_S^{2\prime} \subset Comm_S^2$ contains pair of agents representing communication steps that are to be perform in state S concurrently and concern pairs of agents. The set $Comm_S^{1\prime}$ contains agents that may perform communication steps on their own.

Let us go back to the model considered previously (see Fig. 13). Suppose, the code priority for all active agents is equal to 0 and for all passive agents to 1. Thus,

$$pr(A) = pr(B) = pr(C) = pr(D) = 9 \tag{37}$$
$$pr(K) = pr(L) = pr(M) = 8 \tag{38}$$

After the first performing of the *while* loop interior (see Algorithm 1) we have:

- $Comm'_S = \{A, B, C\}$,
- $Comm''_S = \{(C, A), (A, K), (B, K)\}$,
- $Comm_S = \{(C, A)\}$,
- $Comm_S^{AA} = \emptyset$,
- $Comm_S^{AP} = \{(B, K)\}$,
- $Comm_S^{PP} = \{(M, K), (M, L)\}$,
- $Comm_S^{F} = \emptyset$,
- $Comm_S^{*} = \{(B, K), (M, K), (M, L)\}$,
- $Comm_S^{2} = \{B, M\}$.

Then, after the second performing of the *while* loop interior we have:

- $Comm'_S = \{B\}$,
- $Comm''_S = \{(B, K)\}$,
- $Comm_S = \{(C, A), (B, K)\}$,
- $Comm_S^{AA} = Comm_S^{AP} = \emptyset$,
- $Comm_S^{PP} = \{(M, L)\}$,
- $Comm_S^{F} = \emptyset$,
- $Comm_S^{*} = \{(M, L)\}$,
- $Comm_S^{2} = \{M\}$.

Finally, we have $Comm_S^{2\prime} = \{(C, A), (B, K), (M, L)\}$ and $Comm_S^{1\prime} = \emptyset$.

Algorithm 1 Selecting concurrent communication steps

$Comm_S = \emptyset$
calculate $Comm_S^{AA}$, $Comm_S^{AP}$, $Comm_S^{PP}$, $Comm_S^{F}$, $Comm_S^{*}$, $Comm_S^{2}$, $Comm_S^{1}$
▷ (see (29)-(35))
while $Comm_S^{2} \neq \emptyset$ **do**
$Comm'_S = \{X \in Comm_S^{2}\colon X \text{ has the highest priority in } Comm_S^{2}\}$
$Comm''_S = \{(X,Y) \in Comm_S^{*}\colon \big(X \in Comm'_S \vee Y \in Comm'_S\big)\}$
take a pair $(X',Y') \in Comm''_S$ with the highest sum of agents' priorities
▷ if there is a few such pairs take one of them
$Comm_S = Comm_S \cup \{(X',Y')\}$
$Comm_S^{AA} = Comm_S^{AA} - \{(P,Q)\colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
$Comm_S^{AP} = Comm_S^{AP} - \{(P,Q)\colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
$Comm_S^{PP} = Comm_S^{PP} - \{(P,Q)\colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
$Comm_S^{F} = Comm_S^{F} - \{(P,Q)\colon P = X' \vee P = Y' \vee Q = X' \vee Q = Y'\}$
calculate sets $Comm_S^{*}$, $Comm_S^{2}$ ▷ (see (33)-(34))
end while
calculate $Comm_S^{1}$ ▷ (see 35)
$Comm_S^{1\prime} = Comm_S^{1} - Comm_S$
$Comm_S^{2\prime} = Comm_S$
▷ the new set $Comm_S^{1\prime}$ may contain more elements than initially $Comm_S^{1}$

Suppose the priority function pr is defined as follows:

$$pr(A) = pr(B) = pr(C) = pr(D) = 8 \tag{39}$$
$$pr(K) = pr(L) = pr(M) = 9 \tag{40}$$

Then, while the first performing of the *while* loop interior we have:

- $Comm'_S = \{M\}$,
- $Comm''_S = \{(M,K),(M,L)\}$,

In such a case, we an indeterministic choice between these two pairs. If (M,L) is chosen, then we have next another indeterministic choice between (A,K) and (B,K). Then, if (A,K) is chosen, we have finally, $Comm_S^{2\prime} = \{(M,L),(A,K)\}$ and $Comm_S^{1\prime} = \{B,C\}$.

Now, let us focus on performing communication steps. There are following possible cases:

1. $(X,Y) \in Comm_S^{2\prime} \cap Comm_S^{AA}$,
2. $(X,Y) \in Comm_S^{2\prime} \cap Comm_S^{AP}$,
3. $(X,Y) \in Comm_S^{2\prime} \cap Comm_S^{PP}$,
4. $(X,Y) \in Comm_S^{2\prime} \cap Comm_S^{F}$,
5. $X \in Comm_S^{1\prime} \cap \mathcal{A}_A$ and $S{-}t_{in}(X.p,T){\rightarrow}S'$,
6. $X \in Comm_S^{1\prime} \cap \mathcal{A}_A$ and $S{-}t_{out}(X.p,T){\rightarrow}S'$,
7. $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p,T){\rightarrow}S'$, and $p \notin \mathcal{P}_{proc}(X)$,
8. $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{out}(X.p,T){\rightarrow}S'$, and $p \notin \mathcal{P}_{proc}(X)$,
9. $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p,T){\rightarrow}S'$, and $p \in \mathcal{P}_{proc}(X)$,
10. $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{out}(X.p,T){\rightarrow}S'$, and $p \in \mathcal{P}_{proc}(X)$.

ad. 1. Suppose, $(X,Y) \in Comm_S^{2\prime} \cap Comm_S^{AA}$, $S{-}t_{in}(X.p,T){\rightarrow}$, x is the second argument of the corresponding *in* statement, $S{-}t_{out}(Y.q,T){\rightarrow}$, and value w of type T is sent. In such a case, instead of transitions t_{in} and t_{out}, the transition t_{io} is used to represent the communication. Let $S{-}t_{io}(X.p,Y,q,T){\rightarrow}S'$, then:

- $S'(X) = (running, nextpc(S(X)), ci_S(X), pv_S(X)|_{x=w})$ if $nextpc(S(X)) \neq 0$.
- $S'(X) = (finished, 0, [\,], pv_S(X)|_{x=w})$ if $nextpc(S(X)) = 0$.
- $S'(Y) = (running, nextpc(S(Y)), ci_S(Y), pv_S(Y))$ if $nextpc(S(Y)) \neq 0$.
- $S'(Y) = (finished, 0, [\,], pv_S(Y))$ if $nextpc(S(Y)) = 0$.

If a valueless communication is considered then $pv_S(X)$ remains unchanged.

ad. 2. Suppose, $(X,Y) \in Comm_S^{2\prime} \cap Comm_S^{AP}$, $S{-}t_{in}(X.p,T){\rightarrow}S'$ (or $S{-}t_{out}(X.p,T){\rightarrow}S'$), and procedure $Y.q$ is called. Then:

- $S'(X) = (running, pc_S(X), ci_S(X) \oplus proc(Y.q, p), pv_S(X))$.
- $S'(Y) = (taken, 1, [\,], pv_S(Y))$.

ad. 3. Suppose, $(X, Y) \in Comm_S^{2\prime} \cap Comm_S^{PP}$, $S{-}t_{in}(X.p, T){\rightarrow}S'$ (or $S{-}t_{out}(X.p, T){\rightarrow}S'$), and procedure $Y.q$ is called. Then:

- $S'(X) = (taken, pc_S(X), ci_S(X) \oplus proc(Y.q, p), pv_S(X))$.
- $S'(Y) = (taken, 1, [\,], pv_S(Y))$.

ad. 4. This case is similar to the first case. The new state is defined in the same way. The only difference is that one of these agents has performed its communication step earlier.

ad. 5. Let $X \in Comm_S^{1\prime} \cap \mathcal{A}_A$, $S{-}t_{in}(X.p, T){\rightarrow}S'$. Then:

- $S'(X) = (waiting, pc_S(X), ci_S(X) \oplus in(p|T), pv_S(X))$.

If the port p is used to collect values of one type only, then the $in(p)$ entry is used instead of $in(p|T)$.

ad. 6. This case is similar to the previous one, but the $out(p|T)$ (or $out(p)$) entry is used.

ad. 7. Let $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p, T){\rightarrow}S'$, $p \notin \mathcal{P}_{proc}(X)$ (X calls a procedure of another agent), and $Y = context(X)$. Then:

- $S'(X) = (taken, pc_S(X), ci_S(X) \oplus in(p|T), pv_S(X))$.
- $S'(Y) = (waiting, pc_S(Y), ci_S(Y), pv_S(Y))$.

If the port p is used to collect values of one type only, then the $in(p)$ entry is used instead of $in(p|T)$.

ad. 8. This case is similar to the previous one, but the $out(p|T)$ (or $out(p)$) entry is used.

ad. 9. Let $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{in}(X.p, T){\rightarrow}S'$, $p \in \mathcal{P}_{proc}(X)$, and $Y = context(X)$. This means that Y is directly or indirectly performing the procedure $X.p$. Performing the $S{-}t_{in}(X.p, T){\rightarrow}S'$ step means that the procedure collects its input parameter. Let x be the second argument of the corresponding *in* statement and a value w was sent while the procedure call. If the *in* statement is not the last procedure statement then:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X)|_{x=w})$.

If a valueless communication is considered then $pv_S(X)$ remains unchanged.

If the *in* statement is the last procedure statement then the procedure is finished. Suppose, Y is indirectly performing the procedure $X.p$ and exist $X^1, \ldots, X^m$, $m > 0$ such that Y is performing a procedure of X^1, X^1 is performing a procedure of X^2, ..., X^m is performing input procedure $X.p$ via port p^m.

Let $called_S(X, P)$ denote the set of agents that *potentially called* one of X procedures from the set $P \subseteq \mathcal{P}_{proc}(X)$:

$$\begin{aligned} called_S(X, P) = \{Z\colon ((Z \in \mathcal{A}_A \ \wedge \\ \wedge\, am_S(Z) = waiting \wedge guard \notin ci_S(Z)) \ \vee \\ \vee\, (Z \in \mathcal{A}_P \wedge am_S(context(Z)) = waiting \ \wedge \\ \wedge\, guard \notin ci_S(Z))) \wedge (\exists p \in P,\ \exists q \in \mathcal{P}(Z)\colon \\ ((X.p, Z.q) \in \mathcal{C} \wedge in(q|type(X.p)) \in ci_S(Z)) \ \vee \\ \vee\, ((Z.q, X.p) \in \mathcal{C} \ \wedge \\ \wedge\, out(q|type(X.p)) \in ci_S(Z)))\} \end{aligned} \tag{41}$$

The term *potentially called* means that it is possible that port $Z.q$ is connected with a few ports and the communication via this port can be finalized as a communication with another active agent or another passive one (different than X). When performing of a procedure $X.p$ is finished, guards of all procedures of X are evaluated and a set P of available procedures is received. If at least one of them has been already potentially called i.e. $called_S(X, P) \neq \emptyset$ then X starts another procedure immediately. If $card(called_S(X, P)) > 1$ then one agent with the highest priority is chosen.

Suppose, $Z \in \mathcal{A}_A$ is the chosen agent that starts performing a procedure $X.p'$ via port r. Then:

- $S'(X) = (taken, 1, [\,], pv_S(X)|_{x=w})$.
- $S'(Z) = (running, pc_S(Z), ci_S(Z) \oplus proc(X.p', r), pv_S(Z))$.
- $S'(X^m) = (taken, nextpc(S(X^m)), ci_S(X^m) \ominus proc(X.p, p^m), pv_S(X^m))$, if calling the procedure $X.p$ was not the last procedure block step for X^m. Otherwise, the corresponding X^m procedure has finished and the new state for X^m and X^{m-1} is determined as previously for X and X^m (if an input procedure has been called) or as described at point 10 (if an output procedure has been called).

Suppose, Y is directly performing the procedure $X.p$ via its port q. Then, the new state for X (and Z if any) is defined as above, and:

- $S'(Y) = (running, nextpc(S(Y)), ci_S(Y) \ominus proc(X.p, q), pv_S(Y))$

Suppose, $called_S(X,P) = \emptyset$. Besides agents belonging to the set $called_S(X,P)$, there may exist agents that are waiting for accessibility of X procedures in order to fulfil their *select* statements guards. Let the set $callable_S(X,P)$ be defined as follows:

$$\begin{aligned} &callable_S(X,P) = \{Z\colon ((Z \in \mathcal{A}_A \ \wedge \\ &\wedge\ am_S(Z) = waiting \wedge guard \in ci_S(Z)) \ \vee \\ &\vee\ (Z \in \mathcal{A}_P \wedge am_S(context(Z)) = waiting \ \wedge \\ &\quad \wedge\ guard \in ci_S(Z))) \\ &\wedge \text{ accessibility of procedures belonging to } P \\ &\text{makes at least one branch of the corresponding} \\ &\textit{select} \text{ statement open}\} \end{aligned} \tag{42}$$

Thus, the state S' is defined as follows:

- $S'(X) = (waiting, 0, ci'_S(X), pv_S(X)|_{x=w})$, where $ci'_S(X)$ contains all ports from P together with the direction of parameters sending e.g. $in(p_1)$, $out(p_2)$, etc.
- States of $Y, X^1, \ldots, X^m$ are defined as previously.
- $S'(Z) = (running, nextpc(S(Z)), ci_S(Z) \ominus guard, pv_S(Z))$, for any $Z \in callable_S(X,P) \cap \mathcal{A}_A$.
- $S'(Z) = (taken, nextpc(S(Z)), ci_S(Z) \ominus guard, pv_S(Z))$, for any $Z \in callable_S(X,P) \cap \mathcal{A}_P$.
- $S'(Z') = (running, pc_S(Z'), ci_S(Z'), pv_S(Z'))$, for any $Z \in callable_S(X,P) \cap \mathcal{A}_P$ and $Z' = context(Z)$.

In all above cases, if a valueless communication is considered then $pv_S(X)$ remains unchanged.

ad. 10. Let $X \in Comm_S^{1\prime} \cap \mathcal{A}_P$, $S{-}t_{out}(X.p,T){\rightarrow}S'$, $p \in \mathcal{P}_{proc}(X)$, and $Y = context(X)$. As previously, Y is directly or indirectly performing the procedure $X.p$. Performing the $S{-}t_{out}(X.p,T){\rightarrow}S'$ step means that the procedure returns its result. Let x be the second argument of the corresponding *out* statement and a value w is sent.

Suppose, Y is directly performing the procedure $X.p$ and the *out* statement is not the last procedure statement. In such a case, the state S' is defined as follows:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X))$.
- $S'(Y) = (running, pc_S(Y), ci_S(Y), pv_S(Y)|_{x=w})$.

If a valueless communication is considered then $pv_S(X)$ remains unchanged.

Suppose, Y is indirectly performing the procedure $X.p$ and exist $X^1, \ldots, X^m$, $m > 0$ such that Y is performing a procedure of X^1, X^1 is performing a procedure of X^2, ..., X^m is performing output procedure $X.p$ via one of its ports. Then:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X))$.
- $S'(X^m) = (taken, pc_S(X^m), ci_S(X^m), pv_S(X^m)|_{x=w})$.

As previously, if a valueless communication is considered then $pv_S(X)$ remains unchanged.

If the *out* statement is the last procedure statement then states of agents changes as described at point 9. The only difference is the direction of a value transfer i.e. the parameters value list that is updated belongs to the agent that called the corresponding procedure.

Results of transitions performing for passive agents are defined similarly as for active ones. The only difference is the problem of the last statement in a procedure block. For example, if $X \in \mathcal{A}_P$, $S{-}t_{exec}(X){\rightarrow}S'$, a parameter v is assign a value a with the *exec* statement, and the statement is not the last one in the corresponding procedure block, then the state S' is defined as follows:

- $S'(X) = (taken, nextpc(S(X)), ci_S(X), pv_S(X)|_{v=a})$.

If the *exec* statement is the last one in the corresponding procedure block, then the procedure is finished and the state of the model changes as described previously for the t_{in} transition.

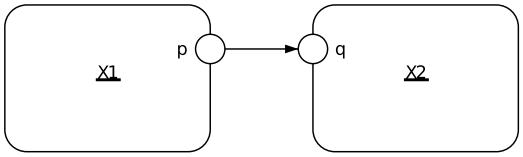
In similar way are defined new states for a transition $t \in \{t_{if}, t_{jump}, t_{loop}, t_{null}, t_{select}, t_{start}\}$. The *exit* statement always finishes the corresponding procedure. It cannot be placed before the procedure *in* (*out*) statement used to collect the procedure argument (return the procedure result).

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 $S{-}t{\rightarrow}S'$. 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 $S = S^1{-}t^1{\rightarrow}S^2{-}t^2{\rightarrow}\ldots{-}t^k{\rightarrow}S^{k+1} = S'$. 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 (Szpyrka and Kotulski, 2011), (Kotulski *et al.*, 2011)). An LTS *graph* is directed graph LTS $= (V, E, L)$, such that $V = \mathcal{R}(S_0)$, $L = \mathcal{T}$, and $E = \{(S,t,S')\colon S{-}t{\rightarrow}S' \wedge S, S' \in \mathcal{R}(S_0)\}$. In other words, an LTS graph presents all reachable states and transitions among them in the form of the directed graph.

To illustrate the idea of LTS graph let us consider two simple examples of Alvis models. The first model shown in Fig. 14 represents two active agents that communicate one with another. The $X1$ agent is a sender and $X2$ is a




```
agent X1 {
  loop {        -- 1
    out p; }    -- 2
}

agent X2 {
  loop {        -- 1
    in q; }     -- 2
}
```

Fig. 14. Example 1.

receiver. The LST graph for this model is shown in Fig 15. The graph is another approach to explain the rules of the Alvis communication between active agents.

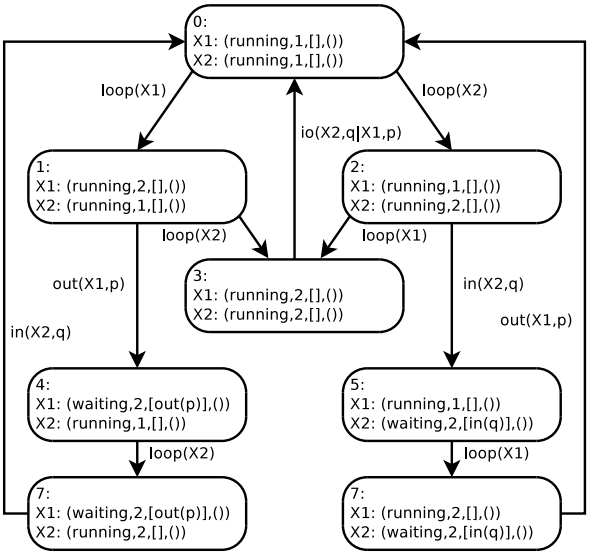
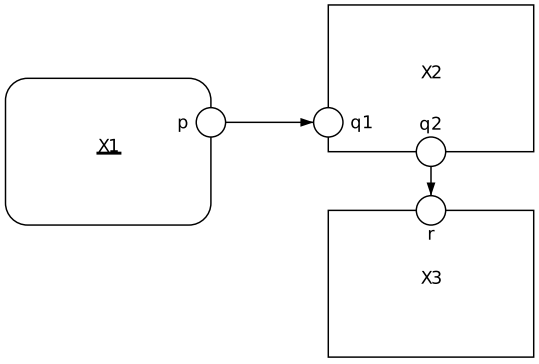


Fig. 15. Example 1 – LTS graph.

The second model is presented in Fig. 16. 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 generated 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.




```
agent X1 {
  loop {              -- 1
    out p; }          -- 2
}

agent X2 {
  proc q1 { in q1;    -- 1
            out q2 }  -- 2
}

agent X3 {
  proc r { in r;      -- 1
           null; }    -- 2
}
```

Fig. 16. Example 2.

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.

The language is still under development. 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`.

References

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

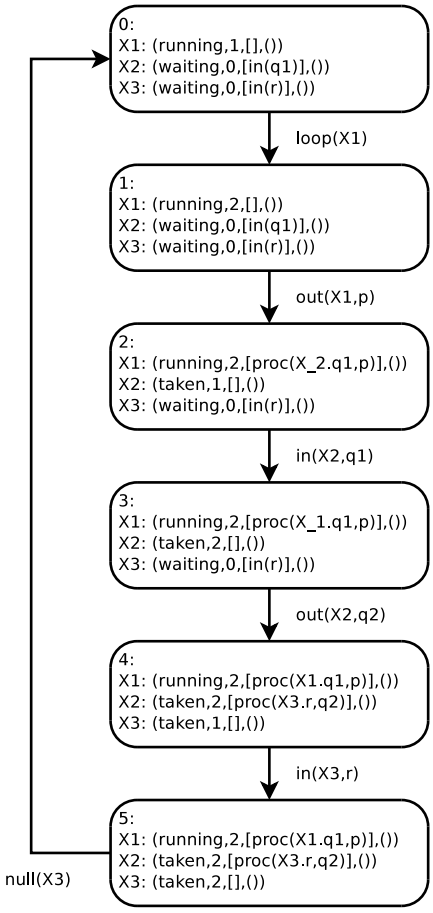


Fig. 17. Example 2 – LTS graph.

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.

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*.

Kotulski, L. and Szpyrka, M. (2011). Graph representation of hierarchical Alvis model structure, *Proc. of the 2011 International Conference on Foundations of Computer Science FCS'11 (part of Worldcomp 2011)*, Las Vegas, Nevada, USA, pp. 95–101.

Kotulski, L., Szpyrka, M. and Sędziwy, A. (2011). Labelled transition system generation from Alvis language, *in* A. König, A. Dengel, K. Hinkelmann, K. Kise, R. Howlett and L. Jain (Eds), *Knowledge-Based and Intelligent Information and Engineering Systems – KES 2011*, Vol. 6881 of *LNCS*, Springer-Verlag, Berlin, Heidelberg, pp. 180–189.

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)*.

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

Szpyrka, M. (2012). *Alvis On-line Manual*, AGH University of Science and Technology.
URL: *http://fm.ia.agh.edu.pl/alvis:manual*

Szpyrka, M. and Kotulski, L. (2011). Snapshot reachability graphs for Alvis models, *in* A. König, A. Dengel, K. Hinkelmann, K. Kise, R. Howlett and L. Jain (Eds), *Knowledge-Based and Intelligent Information and Engineering Systems – KES 2011*, Vol. 6881 of *LNCS*, Springer-Verlag, Berlin, Heidelberg, pp. 190–199.

Szpyrka, M., Kotulski, L. and Matyasik, P. (2011). Specification of embedded systems environment behaviour with Alvis modelling language, *Proc. of the 2011 International Conference on Embedded Systems and Applications ESA'11 (part of Worldcomp 2011)*, Las Vegas, Nevada, USA, pp. 79–85.

Szpyrka, M., Matyasik, P. and Mrówka, R. (2011). Alvis – modelling language for concurrent systems, *in* P. Bouvry, H. Gonzalez-Velez and J. Kołodziej (Eds), *Intelligent Decision Systems in Large-Scale Distributed Environments*, Vol. 362 of *SCI*, Springer-Verlag, pp. 315–342.

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 100 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 systems. He is currently leader of GRADIS project.