Computing and Informatics, Vol. 35, 2016, 55–83

HIERARCHICAL COMMUNICATION DIAGRAMS

Marcin Szpyrka, Piotr Matyasik, Jerzy Biernacki
Agnieszka Biernacka, Michał Wypych, Leszek Kotulski

Department of Applied Computer Science
AGH University of Science and Technology
Mickiewicza 30, 30-059 Kraków, Poland
e-mail: {mszpyrka, ptm, jbiernac, abiernac, mwypych, kotulski}@agh.edu.pl

Abstract. Formal modelling languages range from strictly textual ones like process algebra scripts to visual modelling languages based on hierarchical graphs like coloured Petri nets. Approaches equipped with visual modelling capabilities make developing process easier and help users to cope with more complex systems. Alvis is a modelling language that combines possibilities of formal models verification with flexibility and simplicity of practical programming languages. The paper deals with hierarchical communication diagrams – the visual layer of the Alvis modelling language. It provides all necessary information to model system structure with Alvis, to manipulate a model hierarchy and to understand a model semantics. All considered concepts are discussed using illustrative examples.

Keywords: Alvis language, hierarchical communication diagrams, flat representation, analysis operation, synthesis operation

1 INTRODUCTION

Research on formal methods are particularly intense in the last 20 years. These resulted in formulation of a number of formalisms and verification methods, as well as tools for their practical application. The continuous progress of the computational capabilities allows these tools to verify increasingly more complex systems. Therefore, the field of potential applications of formal methods is quickly expanding. Unfortunately, there is a gap between the formal mathematical modelling languages and languages used in everyday engineering practice. The most popular formal methods currently include Petri nets, process algebra and timed automata. Most model checkers accept only models represented in these formalisms. A problem with their use is that the practice of modelling systems uses them in a significantly different way from the software development practice. For this reason, many software engineers are reluctant to use these formalisms.

Alvis [27, 28] is a formal modelling language being developed at AGH-UST in Krakow, Department of Applied Computer Science. The main motivation behind the creation and development of Alvis language was to make the modelling and verification process more simple and accessible to software developers. In the proposed approach, the heavy mathematical foundations are hidden from the user without compromising the capabilities and expressive power of the formalism. Model description language is also very similar to the popular programming languages which further increases the convenience of its usage for developers. Alvis actually combines the advantages of formal methods and practical modelling languages. Main differences between Alvis and more classical formal methods, like Petri nets and process algebras, include the syntax that is more user-friendly from engineers' point of view, and the visual modelling language (communication diagrams) that is used to define communication among distinguished parts of a model called *agents*. The main difference between Alvis and industry programming languages is a possibility of formal verification of Alvis models using model checking techniques [2].

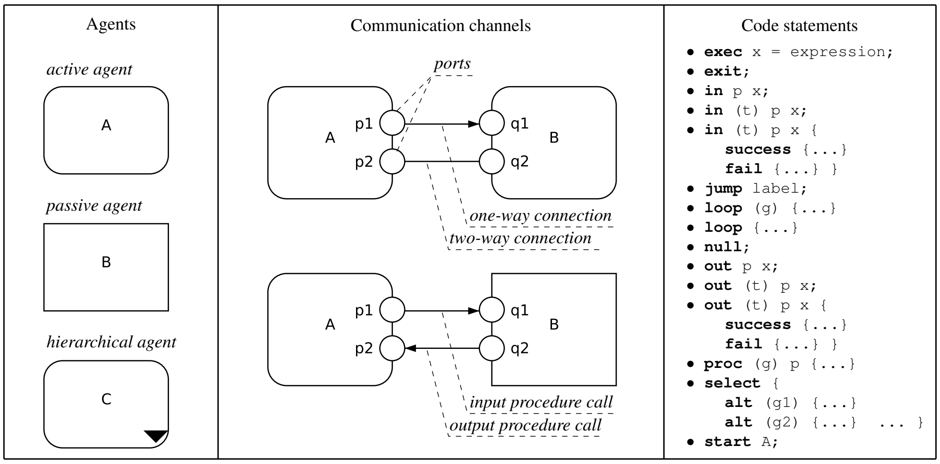
Alvis has its origins in the CCS process algebra [23, 1], the XCCS language [3, 26] and the Ada programming language [4]. The main result of this fact is the concept of *agent* borrowed from CCS. Agent denotes any distinguished part of the system under consideration with defined identity persisting in time. In contrast to process algebras, Alvis uses a high level programming language to define behaviour of agents instead of algebraic equations. Moreover, the communication mechanisms used in Alvis are similar to the Ada rendez-vous mechanism and calling entries of Ada protected objects.

The concept of the communication diagram is a successor of the XCCS language diagram [3, 26]. The main differences are: generalized ports, double direction communication channels, passive agents, and hierarchical structure of diagrams. A communication diagram takes the form of a directed graph with agents represented by nodes and communication channels represented by arcs – a two-way connection should be treated as a pair of arcs. To introduce hierarchical dependencies into communication diagrams we adopted the concept of substitution transitions from coloured Petri nets [16, 25]. A part of a communication diagram can be placed at separate *page* and represented at the higher level by the so-called *hierarchical agent*. In other words, a hierarchical agent represents a subsystem (module) of the considered system and the (sub)page attached to this agent describes the subsystem in details. Of course such subpage may contain another hierarchical agents etc. Moreover, the same page may be attached to more than one hierarchical agent, so a designer may reuse some parts of the model.

The paper deals with theoretical and practical aspects of Alvis communication diagrams. It provides information necessary to understand semantics of diagrams and equips a user with techniques necessary to construct and manipulate models hierarchy. The paper is organised as follows. Section 2 provides a short introduction to the Alvis language and the process of modelling and verification of concurrent systems with Alvis. Section 3 contains a formal definition of non-hierarchical communication diagrams, while Section 4 provides a formal definition of hierarchical communication diagrams. Transformations of hierarchical diagrams are described in Section 5. Advantages of using hierarchy in Alvis are presented in Section 6. Section 7 provides an Alvis communication diagram example used to illustrate usefulness of the hierarchy. A short summary is given in the final section.

2 ALVIS LANGUAGE AT A GLANCE

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 *active* and *passive* ones. *Active agents* perform some activities and each of them can be treated as a thread of control in a concurrent or distributed system. *Passive agents* do not perform any individual activity, but provide a mechanism for the mutual exclusion and data synchronization.



```
• exec x = expression;
• exit;
• in p x;
• in (t) p x;
• in (t) p x {
    success {...}
    fail {...} }
• jump label;
• loop (g) {...}
• loop {...}
• null;
• out p x;
• out (t) p x;
• out (t) p x {
    success {...}
    fail {...} }
• proc (g) p {...}
• select {
    alt (g1) {...}
    alt (g2) {...}  ... }
• start A;
```

Figure 1. Elements of Alvis modelling language

An agent can communicate with other agents through *ports*. To make communication available to two agents a communication channel between their ports must be defined in the graphical layer. Some ports of passive agents represent procedures (services) used to access shared data stored by the agents. Communication with a procedural port is treated as a procedure call. From the point of view of the control and data flow, the Alvis model structure is represented as a directed graph where nodes may represent both kinds of agents (*active* or *passive*) and parts of the model from the lower level. Such a graph is called a *communication diagram.* To cope with complex systems, parts of a communication diagram can be distributed across multiple subdiagrams called *pages.* Each such subdiagram is represented by a *hierarchical agent.* A communication diagram with at least one hierarchical agent is called *hierarchical communication diagram.* Behaviour of each active and passive agent is defined in the *code layer.* Alvis uses statements typical for high level programming languages and some elements of the Haskell functional programming language [24]. A survey of Alvis graphical items and code statements is given in Figure 1. For more details see [27] and the project website `http://alvis.kis.agh.edu.pl`.

From the user's point of view, only graphical and code layers must be designed. A complete model contains also a *system layer.* The layer is strictly connected with the system architecture. For modelling concurrent systems, the α^0 system layer is used. The layer is based on the assumption that each active agent has access to its own processor and performs its statements in parallel with other agents.

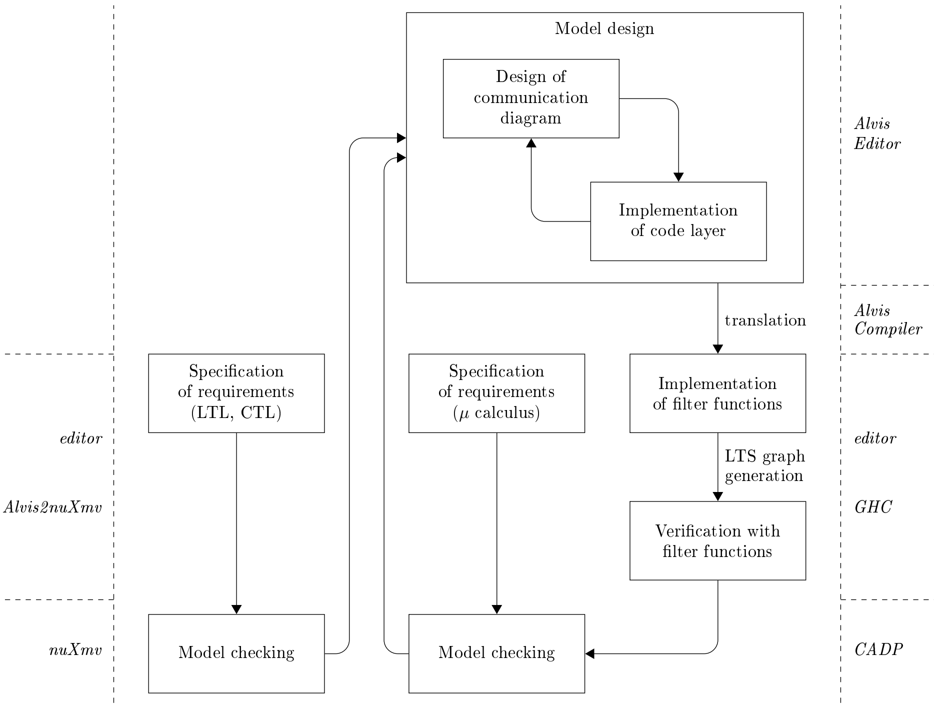

Model design
Design of communication diagram
Implementation of code layer
Alvis Editor
translation
Alvis Compiler
Specification of requirements (LTL, CTL)
Specification of requirements (μ calculus)
Implementation of filter functions
LTS graph generation
Verification with filter functions
editor
Alvis2nuXmv
editor
GHC
nuXmv
Model checking
Model checking
CADP

Figure 2. Modelling and verification process with Alvis

The scheme of the modelling and verification process with Alvis is shown in Figure 2. From a user's perspective, it starts from designing a model using prototype modelling environment called *Alvis Editor.* The designed model is stored using XML file format. Then *Alvis Compiler* is used to translate it into Haskell source code and its Haskell representation is used to generate the LTS graph (*labelled transition system*). We use Haskell as a middle-stage representation of an Alvis model in similar way as CPN Tools uses SML to generate reachability graphs for coloured Petri nets [16]. The main difference between these approaches is that Alvis users have access to the generated Haskell source files and may include some extra Haskell code into them.

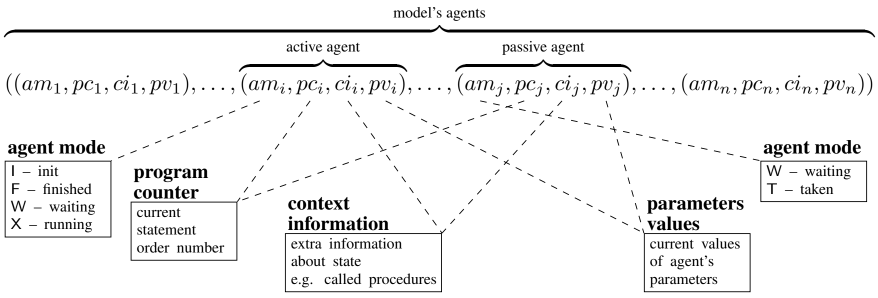

model's agents
active agent
passive agent
$((am_1, pc_1, ci_1, pv_1), \ldots, (am_i, pc_i, ci_i, pv_i), \ldots, (am_j, pc_j, ci_j, pv_j), \ldots, (am_n, pc_n, ci_n, pv_n))$
agent mode
I – init
F – finished
W – waiting
X – running
program counter
current statement order number
context information
extra information about state e.g. called procedures
parameters values
current values of agent's parameters
agent mode
W – waiting
T – taken

Figure 3. Representation of an Alvis model state

An Alvis model semantics finds expression in an LTS graph. Execution of any language statement is expressed as a transition between formally defined states of such a model [28]. A state of an Alvis model is represented as a sequence of agents' states. To describe the current state of an agent we use a four-tuple containing: agent's mode, its program counter, context information list and values of its parameters (see Figure 3). An LTS graph is an ordered graph with nodes representing states of the considered system and edges representing transitions among states. The edges are labelled with names of executed statements.

Alvis LTS graphs can be verified using the CADP toolbox [12]. It offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking. Alvis Compiler provides a possibility to export an LTS graph into CADP Aldebaran format. We use the CADP *evaluator* tool to check whether an LTS graph satisfies requirements given as μ-calculus formulae [10, 19, 22]. It should be emphasized that this is an action based approach. A μ-calculus formula concerns actions labels while states of the considered model are represented using their numbers only.

Furthermore, the so-called filtering functions can be used to verify Alvis models. The internal representation of an LTS graph is a Haskell [24] list of model states. User-defined Haskell functions (called *filtering functions*) that search an LTS graph for some states or parts of the graph that meet given requirements can be included into the model. Moreover, the Haskell approach can be used to implement user defined verification algorithms that search for some specified parts of an LTS graph and are not provided by verification toolboxes.

A state-oriented approach is also provided by the nuXmv model checker [6] (the previous version known as NuSMV). The *Alvis2nuXmv* translator provides automatic translation of an Alvis LTS graph into an equivalent nuXmv state machine. The nuXmv tool enables automatic verification if formulae specified in a temporal logic (LTL, CTL or RTCTL) is satisfied by the model [2, 9].

3 NON-HIERARCHICAL COMMUNICATION DIAGRAMS

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

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

For a set of agents W we define: $\mathcal{P}(W) = \bigcup_{X\in W} \mathcal{P}(X)$, $\mathcal{P}_{in}(W) = \bigcup_{X\in W} \mathcal{P}_{in}(X)$, etc. Moreover, let $\mathcal{P}$ denote the set of all model ports, $\mathcal{P}_{in}$ denote the set of all model input ports, etc. We use two notations for ports. A single lower-case letter e.g. p denotes a port p of an agent. If it is necessary to point out the agent, the dot notation is used e.g. $X.p$. Let $\mathcal{N}(Y)$ denote the set of port names of ports belonging to set Y. 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: $\mathcal{A} = \{X_1, \dots, X_n\}$ 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:

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

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

Each element belonging to $\mathcal{C}$ is called a *connection* or a *communication channel.* The restrictions from Definition 1 have the following meaning. (1) – A connection cannot be defined between ports of the same agent. (2) – Procedural ports are either input or output ones. (3), (4) – A connection between an active and a passive agent must be a procedure call. From conditions (2)-(4) it follows that any connection with a passive agent must be an one-way connection. (5) – A connection between two passive agents must be a procedure call from a non-procedural port. If $(p, q) \in \mathcal{C}$ then p is an output port and q is an input port of the (p, q) connection. The start function σ makes possible delaying activation of some agents. Names of agents that are initially activated are underlined in a communication diagram.

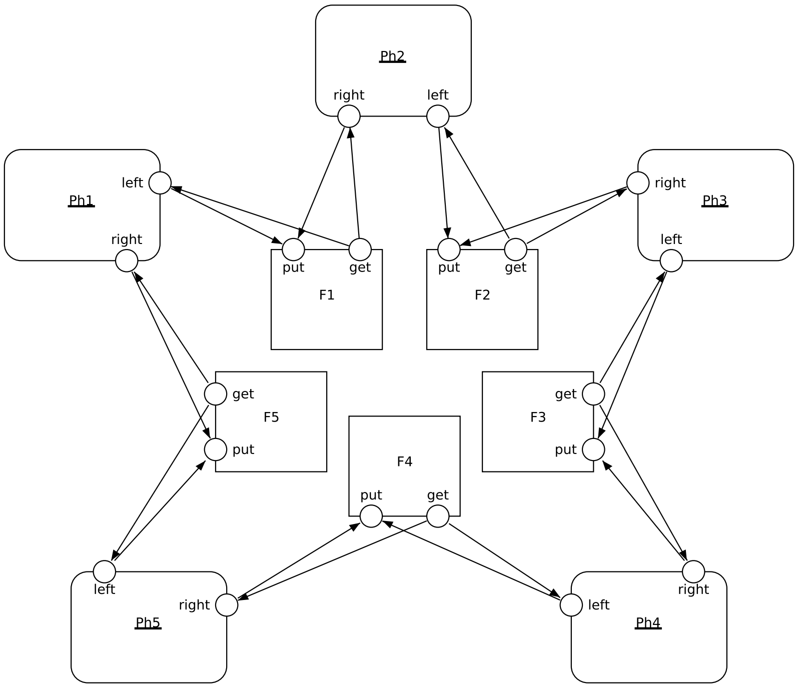


Figure 4. Communication diagram for model of dining philosophers

Let us consider the well-known problem of dining philosophers. Five philosophers sit around a circular table. They spend their life alternately thinking and eating. There is a large bowl of spaghetti in the centre of the table. There are also five plates and five forks set between them. Eating the spaghetti requires the use of two forks. Each philosopher thinks. When he gets hungry, he picks up the two forks closest to him. If a philosopher can pick up both forks, he eats for a while, then he puts down the forks and starts thinking. The communication diagram for the considered model is shown in Figure 4. It contains 5 active and 5 passive agents that represent philosophers and forks respectively. For a given philosopher, ports *right* and *left* are used to take up and put back his right and left fork respectively. On the other hand, ports *get* and *put* represent possible fork's procedures.

4 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 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 procedural ones.

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

- $\mathcal{A}^i = \{X^i_1, \dots, X^i_n\}$ 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.
- $\mathcal{C}^i \subseteq \mathcal{P}^i \times \mathcal{P}^i$, $(\mathcal{P}^i = \bigcup_{X \in \mathcal{A}^i} \mathcal{P}(X))$, is the *communication relation*, such that:

$$
\begin{aligned}
\forall_{X \in \mathcal{A}^i} (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C}^i &= \emptyset, && (6)\\
\mathcal{P}^i_{proc} \cap \mathcal{P}^i_{in} \cap \mathcal{P}^i_{out} &= \emptyset, && (7)\\
\mathcal{P}^i_{proc} \cap \mathcal{P}(\mathcal{A}^i_H) &= \emptyset, && (8)\\
(p,q) \in (\mathcal{P}(\mathcal{A}^i_A) \times \mathcal{P}(\mathcal{A}^i_P)) \cap \mathcal{C}^i &\Rightarrow q \in \mathcal{P}^i_{proc}, && (9)\\
(p,q) \in (\mathcal{P}(\mathcal{A}^i_P) \times \mathcal{P}(\mathcal{A}^i_A)) \cap \mathcal{C}^i &\Rightarrow p \in \mathcal{P}^i_{proc}, && (10)\\
(p,q) \in (\mathcal{P}(\mathcal{A}^i_P) \times \mathcal{P}(\mathcal{A}^i_P)) \cap \mathcal{C}^i &\Rightarrow (p \in \mathcal{P}^i_{proc} \wedge q \notin \mathcal{P}^i_{proc}) \\
&\qquad \vee (q \in \mathcal{P}^i_{proc} \wedge p \notin \mathcal{P}^i_{proc}), && (11)\\
(p,q) \in (\mathcal{P}(\mathcal{A}^i_P) \times \mathcal{P}(\mathcal{A}^i_H)) \cap \mathcal{C}^i &\Rightarrow (q,p) \notin \mathcal{C}^i, && (12)\\
(p,q) \in (\mathcal{P}(\mathcal{A}^i_H) \times \mathcal{P}(\mathcal{A}^i_P)) \cap \mathcal{C}^i &\Rightarrow (q,p) \notin \mathcal{C}^i. && (13)
\end{aligned}
$$

- $\sigma^i \colon \mathcal{A}^i_A \to \{False, True\}$ is the *start function*.

The restrictions from Definition 2 have similar meaning as the ones from Definition 1. Moreover, (8) – Hierarchical agents cannot have procedural ports. (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 substitutions illegal, i.e. after the transformation of a hierarchical diagram into the equivalent flat one, all connections must satisfy the conditions (1)–(5).

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

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

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

X^i results in the following set of hierarchical communication channels:

$$\mathcal{C}^j_{X^i} = \{(Z^i.p, Y^j.q) \colon (Z^i.p, X^i.q) \in \mathcal{C}^i\} \cup \{(Y^j.q, Z^i.p) \colon (X^i.q, Z^i.p) \in \mathcal{C}^i\}. \quad (15)$$

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

$$card(\mathcal{P}(X^i)) = card(\mathcal{P}^{X^i}_{join}(D^j)), \quad (16)$$
$$\mathcal{N}(\mathcal{P}(X^i)) = \mathcal{N}(\mathcal{P}^{X^i}_{join}(D^j)), \quad (17)$$
$$\mathcal{P}^{X^i}_{join}(D^j) = \mathcal{P}^j_{unc}, \quad (18)$$

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

$$\mathcal{A}' = \mathcal{A}^i \cup \mathcal{A}^j \setminus \{X^i\}, \quad (19)$$
$$\mathcal{C}' = \mathcal{C}^i \cup \mathcal{C}^j \cup \mathcal{C}^j_{X^i} \setminus \{(p, q) \colon p \in \mathcal{P}(X^i) \vee q \in \mathcal{P}(X^i)\}, \quad (20)$$
$$\sigma'(Y) = \begin{cases} \sigma^i(Y) \colon Y \in \mathcal{A}^i_A \\ \sigma^j(Y) \colon Y \in \mathcal{A}^j_A \end{cases}, \quad (21)$$

satisfies all conditions from Definition 2. If instead of the condition (16), it holds:

$$card(\mathcal{P}(X^i)) < card(\mathcal{P}^{X^i}_{join}(D^j)), \quad (22)$$

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

The idea of simple substitution is illustrated by Figure 5, while the idea of extended substitution is illustrated by Figure 6.

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 *labels* of arcs, and $E \subseteq V \times L \times V$ is the set of *arcs*.

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*, such that sets of agents $\mathcal{A}^i$ $(i = 1, \ldots, k)$ are pairwise disjoint, and $\gamma \colon \mathcal{A}_H \to \mathcal{D}$, 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 $X \in \mathcal{A}_H$, X and $\gamma(X)$ satisfy the requirements of the simple or extended substitution.
3. Graph $\mathcal{G} = (\mathcal{D}, E, \mathcal{A}_H)$ where $(D^i, X^i, D^j) \in E$ iff $\gamma(X^i) = D^j$ 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 represent the substitution function γ. Each edge represents the page to which belongs the hierarchical agent and the subpage associated with the agent. Formally pages from the set $\mathcal{D} \setminus \gamma(\mathcal{A}_H)$ are called

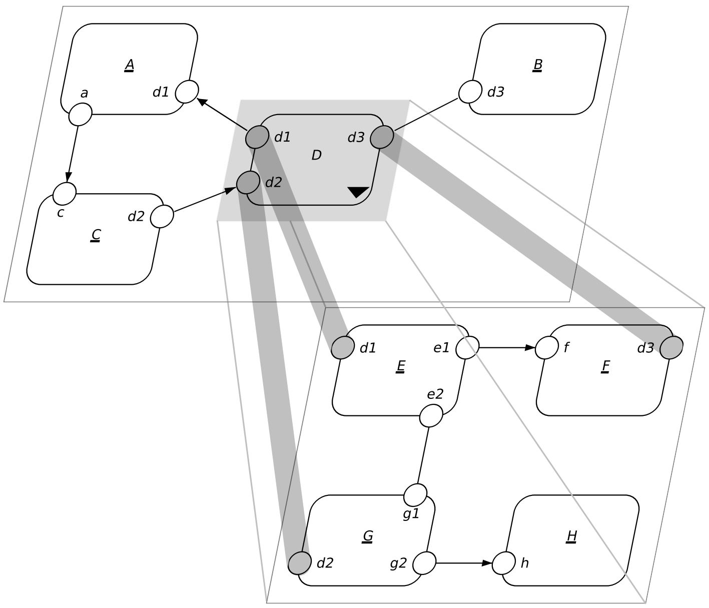


Figure 5. Simple substitution

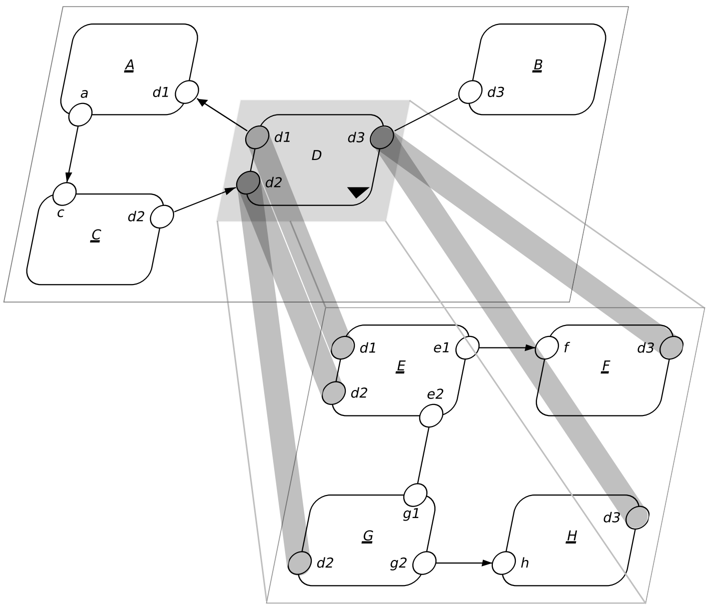


Figure 6. Extended substitution

primary pages, They are roots of trees that constitute the page hierarchy graph. Following symbols are valid for hierarchical communication diagrams:

$$\mathcal{A}_A = \bigcup_{i=1,\dots,k} \mathcal{A}_A^i, \quad \mathcal{A}_P = \bigcup_{i=1,\dots,k} \mathcal{A}_P^i, \quad \mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P \cup \mathcal{A}_H, \tag{23}$$

$$\sigma : \mathcal{A}_A \to \{\mathit{False}, \mathit{True}\} \text{ and } \forall_{i=1,\dots,k} \forall_{X^i \in \mathcal{A}_A} \sigma(X^i) = \sigma^i(X^i), \tag{24}$$

$$\mathcal{C} = \bigcup_{i=1,\dots,k} \mathcal{C}^i \cup \bigcup_{X \in \mathcal{A}_H \wedge \gamma(X) = D^j} \mathcal{C}_X^j. \tag{25}$$

5 HIERARCHY ELIMINATION

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

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 $X = Y_1 \succ \dots \succ Y_k = Y$ for some $Y_1, \dots, Y_k \in \mathcal{A}$. Moreover, it is assumed that for any agent $X \in \mathcal{A} \setminus \mathcal{A}_H$, $X \succeq X$.

Definition 7. A *flat representation* of a communication diagram $H = (\mathcal{D}, \gamma)$ is the triple $(\mathcal{F}, \mathcal{C}', \sigma')$ such that:

$$\forall_{X,Y \in \mathcal{F} \subseteq \mathcal{A}} \; X \neq Y \Rightarrow X \not\succeq Y, \tag{26}$$

$$\forall_{X \in \mathcal{A} \setminus \mathcal{A}_H} \exists_{Y \in \mathcal{F}} \; Y \succeq X, \tag{27}$$

$$\mathcal{C}' = \{(X.p, Y.q) \in \mathcal{C} \colon X, Y \in \mathcal{F}\}, \tag{28}$$

$$\sigma' = \sigma|_{\mathcal{A}_A \cap \mathcal{F}}. \tag{29}$$

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 representation to another, more detailed one, using the analysis operation.

Definition 8. 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:

$$\mathcal{F}^* = \mathcal{F} \setminus \{X\} \cup \mathcal{A}^i, \tag{30}$$

$$\mathcal{C}^* = \{(Y.p, Z.q) \in \mathcal{C} \colon Y, Z \in \mathcal{F}^*\}, \tag{31}$$

$$\sigma^* = \sigma|_{\mathcal{A}_A \cap \mathcal{F}^*}. \tag{32}$$

Definition 9. Let H be a hierarchical communication diagram, $(\mathcal{F}, \mathcal{C}', \sigma')$ be a flat representation of H, $Y \in \mathcal{F}$ and there 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:

$$\mathcal{F}^* = \mathcal{F} \setminus \mathcal{A}^i \cup \{X\}, \tag{33}$$

$$\mathcal{C}^* = \{(Y.p, Z.q) \in \mathcal{C} \colon Y, Z \in \mathcal{F}^*\}, \tag{34}$$

$$\sigma^* = \sigma|_{\mathcal{A}_A \cap \mathcal{F}^*}. \tag{35}$$

Definition 10. A flat representation $(\mathcal{F}, \mathcal{C}', \sigma')$ is called the *maximal flat representation* iff $\forall_{X \in \mathcal{A}} \exists_{Y \in \mathcal{F}} X \succeq Y$.

Such a maximal flat representation does not contain hierarchical agents.

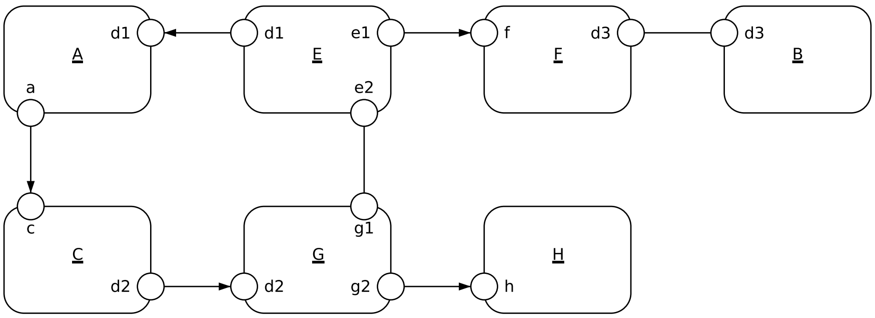


Figure 7. Result of analysis operation for model from Figure 5

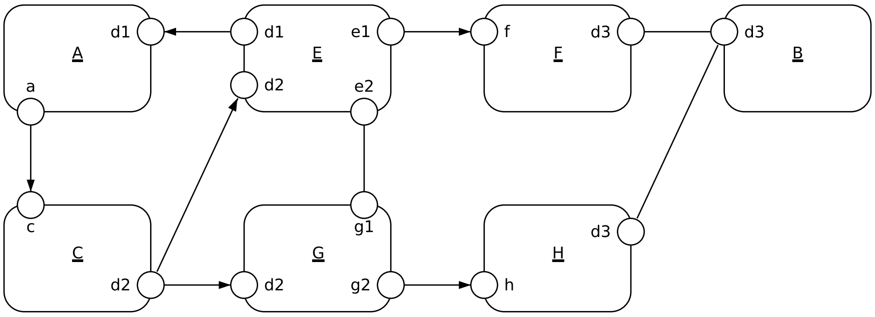


Figure 8. Result of analysis operation for model from Figure 6

Let us consider the hierarchical model H presented in Figure 5. The primary page D^1 (composed of agents A, B, C and D) is a flat representation of the model.

The flat representation generated by $\mathrm{AN}(H, D^1, D)$ is given in Figure 7 (let us denote the page by D^3). On the other hand, any of the synthesis operations $\mathrm{SN}(H, D^3, E)$, $\mathrm{SN}(H, D^3, F)$, $\mathrm{SN}(H, D^3, H)$, $\mathrm{SN}(H, D^3, G)$ gives back the flat representation with page D^1. Similarly, the flat representation generated by $\mathrm{AN}(H, D^1, D)$ for model from Figure 6 is given in Figure 8.

6 HIERARCHICAL COMMUNICATION DIAGRAMS IN PRACTICE

One of the main motivations behind formulating the Alvis language was to make the formal verification more intelligible and easy to use for the average engineer. The graphical layer of the Alvis model was expected to be both easy to model and to understand. This section is introducing standard situations in which hierarchy usage is greatly improving the readability of the model.

6.1 Modules

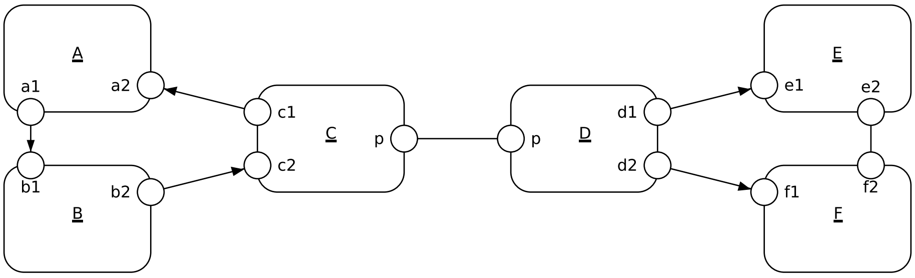


Figure 9. Original communication diagram

Splitting the system into smaller parts is one of the most basic concepts in software engineering. Its fundamental purpose it to avoid situations in which having everything in one module, class or file one have to worry about everything at once when expanding the existing solution. Although this practice may work for small systems, for big ones it quickly becomes next to impossible. To solve this problem, fragments of functionality are split into their own modules which encapsulate the separated pieces of logic. Then, when working on a particular module, one does not have to directly consider the implications of the work on other parts of the system. This is invaluable for working efficiently. There are many other benefits to breaking system into modules, e.g. the model is more maintainable, testable and reusable. Moreover, such a module can be used as a reusable component.

Breaking the system into modules is the most common and general application of hierarchy. Modelling in Alvis language incorporates this practice. Figure 9 presents an example of a communication diagram. Supposing its logic can be divided and

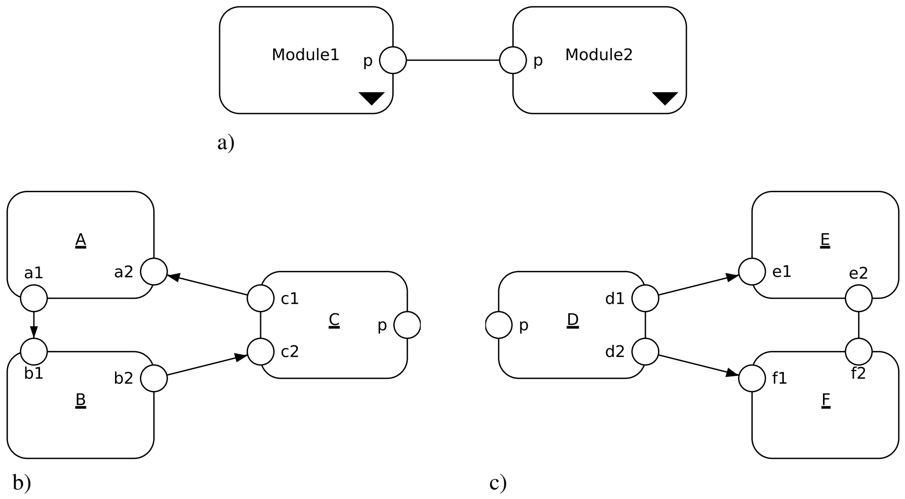


Figure 10. a) Hierarchical agents for two modules; b) subpage for agent Module1; c) subpage for agent Module2

encapsulated into two separate modules, the analysis operation can be used. Its result is presented in Figure 10. There is a hierarchical diagram containing two hierarchical agents on the primary page a) and two subpages with agents belonging to the corresponding modules (b) and c)).

6.2 Removing Multiple Connections

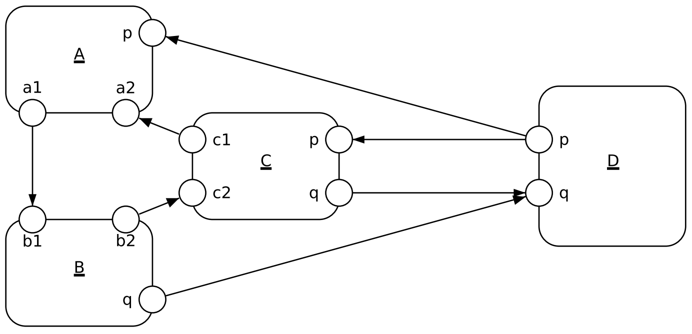


Figure 11. Original communication diagram

Creating even a moderately complex model can lead to situations in which the communication diagram becomes difficult to read due to the increasing amount of

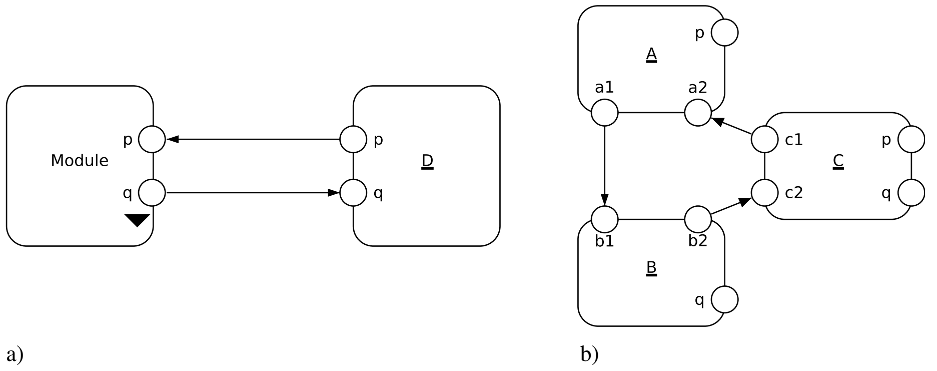


Figure 12. a) Inserted hierarchical agent Module; b) subpage for agent Module

connections between the agents. The warning signal is definitely the moment in which some of the connections are intersecting each other. A simple model with 4 agents and 7 connections between their ports is shown in Figure 11. Although this diagram is readable, it is mostly so because of its small size. It is not difficult to notice that agent `D` is connected to every other agent in the diagram. Adding more agents connected to it will result in progressing obscuration of the model. However, a smart use of an analysis operation can reduce the amount of visible connections. Figure 12 presents the same model with all the agents communicating with agent `D` grouped and brought to a subpage. The total amount of connections between the agents is reduced to 5 and the legibility of the model is definitely increased.

6.3 Replacing Multiple Agent's Instances with a Single Representation

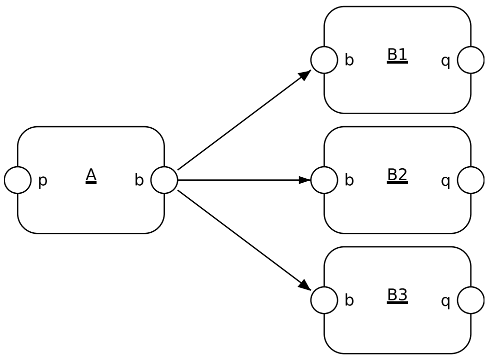


Figure 13. Original communication diagram

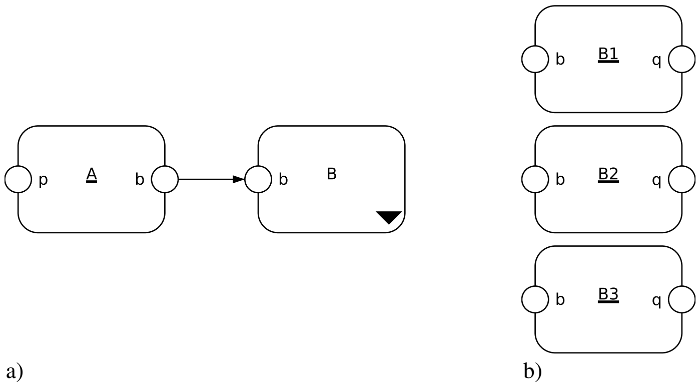


Figure 14. a) Inserted hierarchical agent B; b) subpage for agent B

Another condition for using hierarchy is when the model contains multiple instances of the same agent. This is actually a very common scenario in real-time systems. Elements such as sensors, indicators, displays and other subdevices are often repeated in the scope of a single system. Each of them can be on some level of abstraction represented by a separate agent. Therefore the communication diagram can quickly become crowded with the great amount of reoccurring instances of specific agents. Utilising hierarchy in such case enables the possibility of replacing all these instances with a single hierarchical agent. An example is shown in Figure 13. The original diagram contains 3 instances of the same agent ($B1$–$B3$). These instances can be easily replaced by a single hierarchical agent B and placed on one subpage (Figure 14). This operation has one more great advantage. When new instances are added, it is enough to add them to the defined subpage. One does not have to worry about drawing connections with other agents. This also applies to the removal of unwanted instances.

6.4 Grouping Repeating Fragments of a Model

In extensive systems, repeating fragments of the model can often occur. Each of these excerpts represents similar functionality but is placed in a different part of the system and is connected to different agents. Such a model is therefore potentially easy to disrupt. Assuming that these fragments need some kind of a change, one would have to find all of these fragments and update them one by one. The solution to this problem is once again in hierarchy. Figure 15 contains a simple example of a model with repeating fragments. In the example these fragments are placed in the same place and are easy to identify. In model of a real system it may not be so. Therefore, placing them on a single subpage may be very useful. In the Figure 16 these excerpts are represented by a single hierarchical agent and moved to the lower level. This way, any changes to be made are

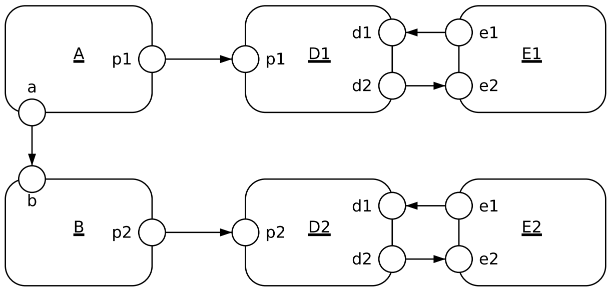


Figure 15. Original communication diagram

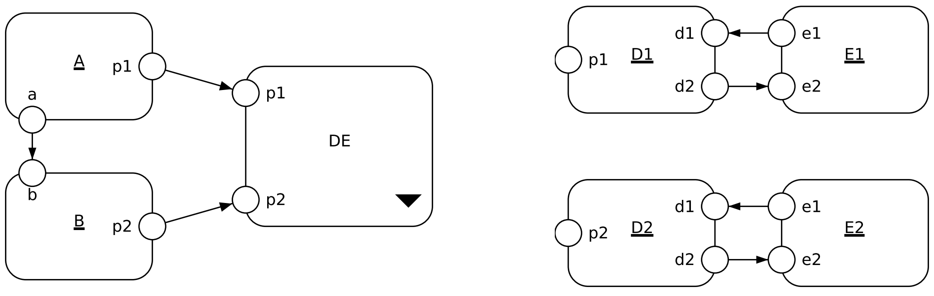


Figure 16. a) Inserted hierarchical agents DE; b) subpage for agent DE

less prone to cause mistakes because all repeated fragments are gathered in one place.

7 CASE STUDY

A model of a fire alarm control panel (FACP) is used to illustrate the usefulness of hierarchy in modelling of complex systems in Alvis language. Fire alarm system is an excellent example of a safety critical system. Its failures always cause major losses. If it raises the alarm too late, many people may die or become seriously injured. Yet false alarms result in high costs due to, inter alia, the stoppage of technological processes or activation of automatic extinguishing system. Hence, comprehensive formal verification of such systems is crucial.

According to the SITP (Polish Association of Fire Engineers and Technicians) alarm variants usage is a common practice in construction of fire alarm control panels [8]. This method aims at the reduction of false fire alarms. Its most popular variant is two-stage alarming which scheme is presented in Figure 17. It is a scheme of an actual solution designed by the SITP association.

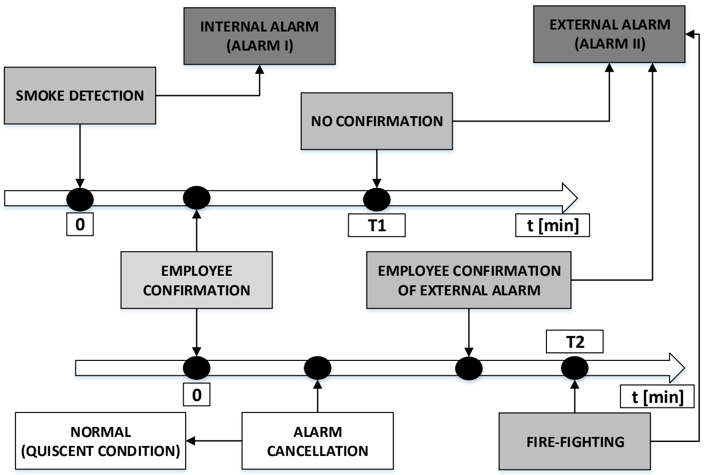


Figure 17. Fire alarm control panel scheme [8]

When one of the smoke detectors detects the fire, the internal alarm is raised. Personnel have only a limited amount of time for reaction before the external alarm is being raised. This period of time is here denoted by the T1 variable. If one of the employees confirms the alarm he has another specified amount of time to assess the threat and take an action. Again, if he fails to react in time, the external alarm is raised. This amount of time is denoted as T2. There are 4 possible scenarios of what can happen in that time. The first possibility is that an employee will find that the alarm is false e.g. one of the smoke detectors is broken down. In this case an employee can simply turn off the alarm. The second scenario is that a fire actually broke out but is small enough for the personnel to handle it by themselves. In this case they must cancel the alarm before the T2 time runs out. If they fail, the external alarm is being raised, which is the third scenario. The last one is when the fire threat is overwhelming and employees turn on the external alarm right away, without waiting for the T2 to run out. External alarm means that the internal extinguishing systems are activated and the fire brigade is automatically called. The last enhancement, not depictured explicitely in the scheme, is a coincidence detection system. Its basic behaviour is as follows: if only one smoke detector detects the fire, only the internal alarm is raised. However, when a specified number of detectors detect the fire at the same time, the external fire alarm is called right away, no matter in which state the system is at the moment.

Non-hierarchical communication diagram of the described FACP system is presented in Figure 18. It depictures quite basic version of the system, with only 2 smoke detectors, 2 manual call points and 2 sprinklers per every of 3 floors of a building. Analysing this diagram one great advantage of Alvis should be noted –

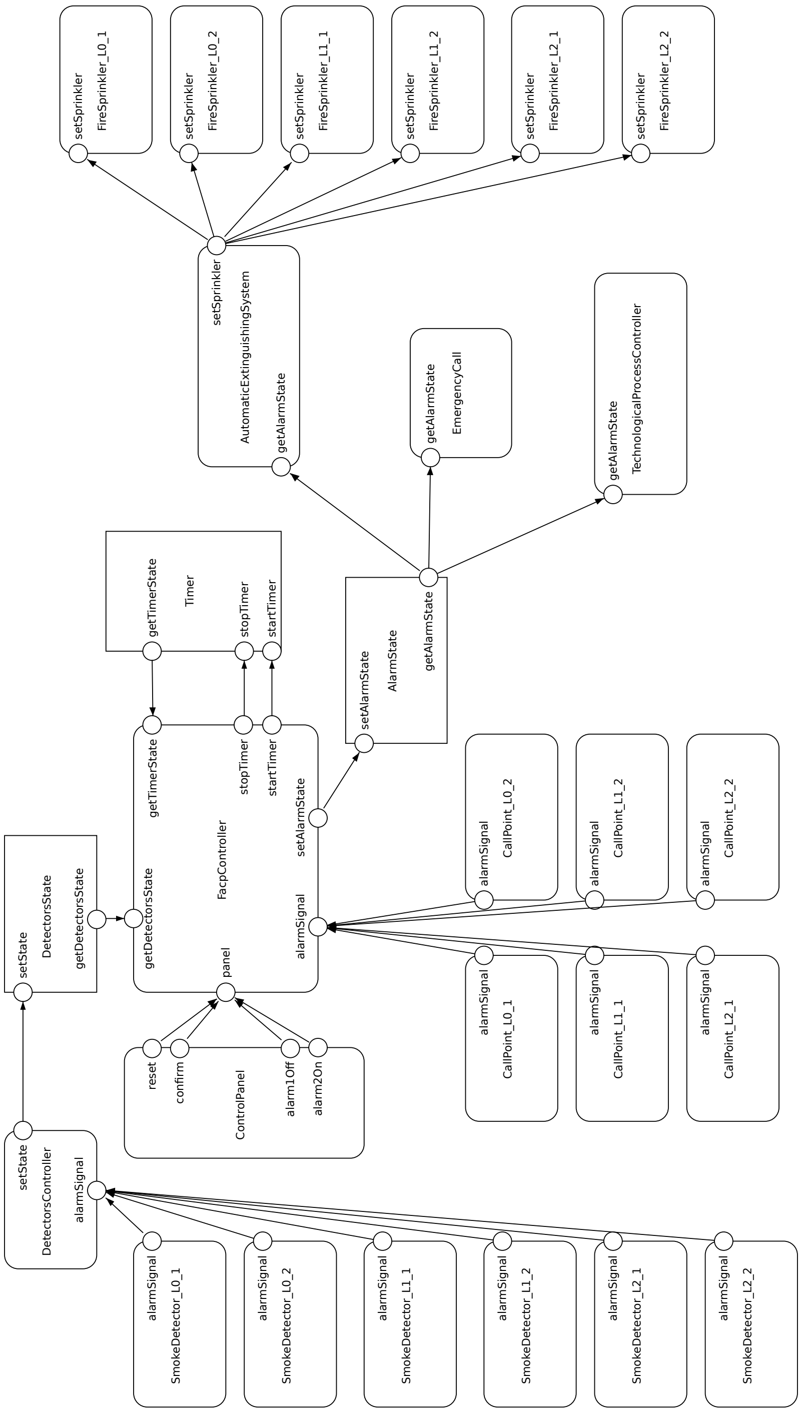


Figure 18. Fire alarm control panel non-hierarchical model

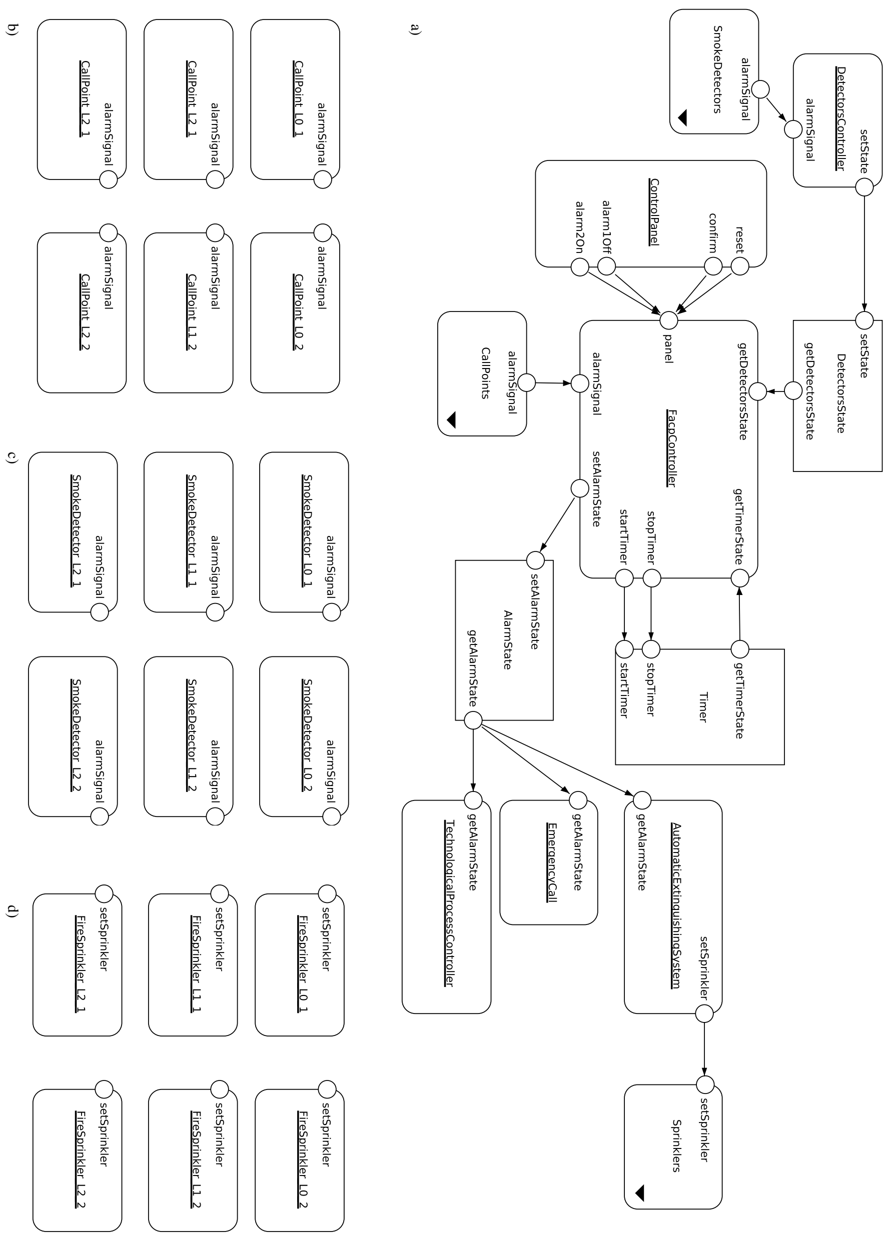


Figure 19. a) Primary page of FACP model (Figure 18) after replacing multiple agent instances; b) CallPoints' agent subpage; c) SmokeDetectors' subpage; d) Sprinklers' subpage

the compactness of its graphical representation. The same system modelled in any class of Petri nets would be much more complicated and take at least a couple of times the space it has taken in this case.

Nonetheless, the first signs of illegibility are visible in the diagram. The point that draws attention is the repetition of instances of agents representing detectors, call points and sprinklers. However, these multiple instances can be represented by hierarchical agents, according to the rule presented in Section 6.3. The results of performing operation on the initial, non-hierarchical communication diagram are shown in Figure 19.

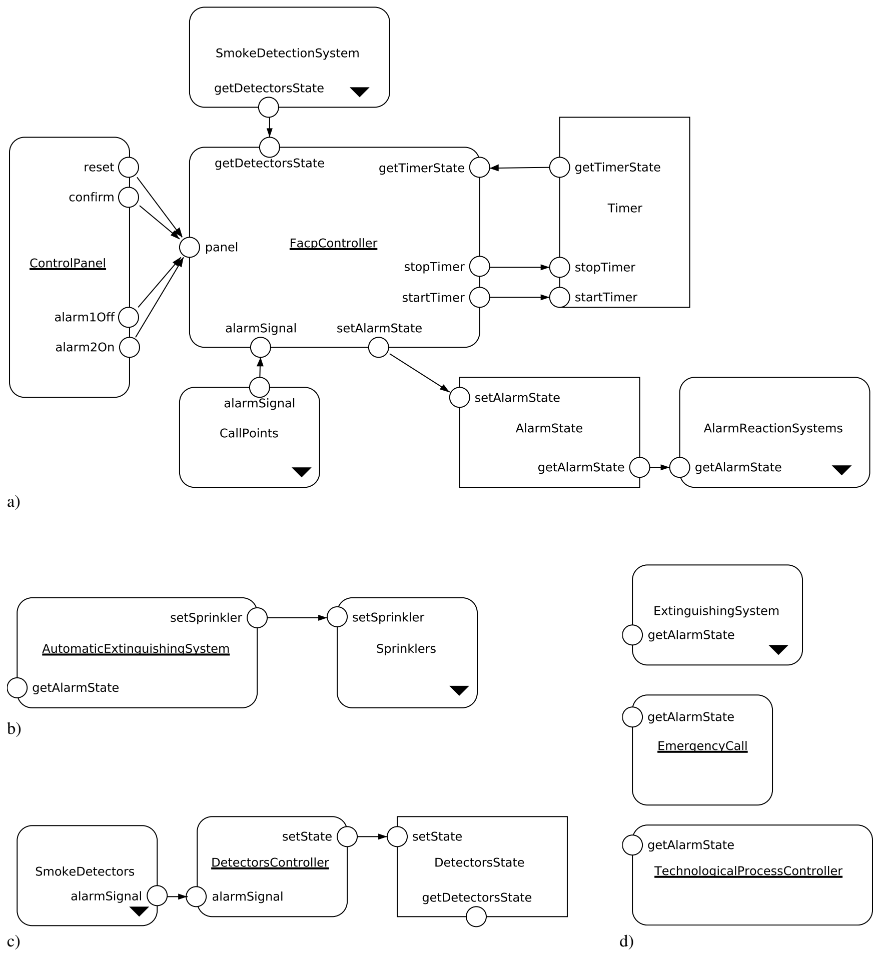


Figure 20. a) Primary page of FACP model (Figure 19) after grouping multiple agents and components with a common interface; b) ExtinguishingSystem's subpage; c) SmokeDetectionSystem's subpage; d) AlarmReactionSystem's subpage

The obtained hierarchical communication diagram is more readable. Moreover, adding new instances of the moved agents will not affect its readability. Nonetheless, the model can still be improved. Extinguishing and smoke detection subsystems are actually independent of the fire alarm control panel. They can be modelled and verified separately and therefore should be grouped into components (Figure 20). This would allow to switch between different versions of these subsystems without the need to change the structure of the base model. For similar reasons, agents with a common interface and related functionality should be grouped. Figure 20 d) presents page containing *ExtinguishingSystem*, *EmergencyCall* and *Technological-ProcessController* agents grouped into a single representation.

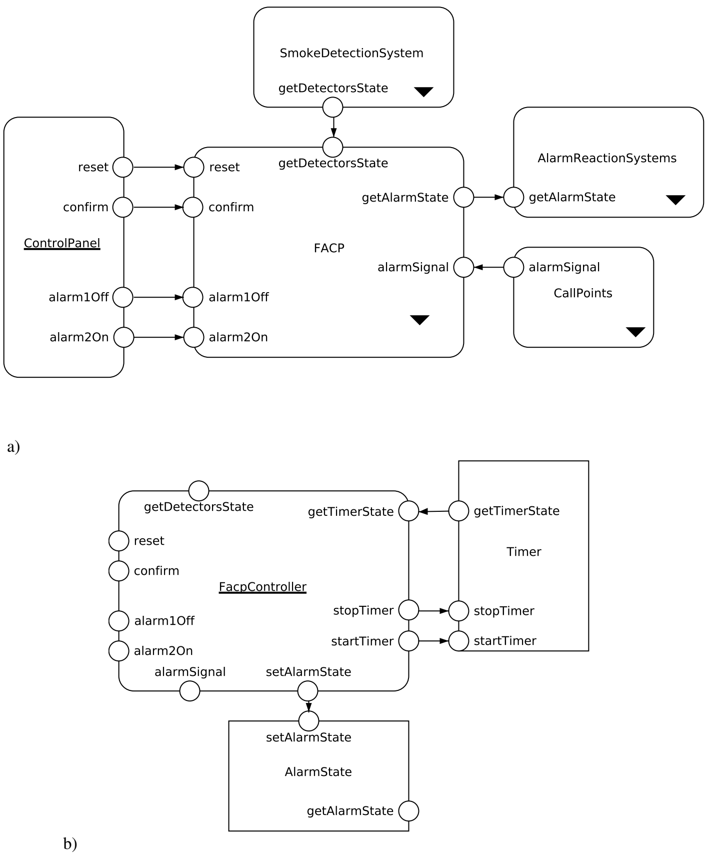


Figure 21. a) Primary page of FACP model (Figure 20) after creating a new module encapsulating a core of the fire alarm control panel; b) FACP's subpage

The last is the operation of encapsulation of fire alarm control panel's core. It required a small change in the *FACPController* agent – the *panel* port had been divided into 4 separate ports with the same labels as the ports it communicates with. This allowed moving this agent to a lower level. The final result is presented in Figure 21. The structure of the model (page hierarchy graph) is shown in Figure 22.

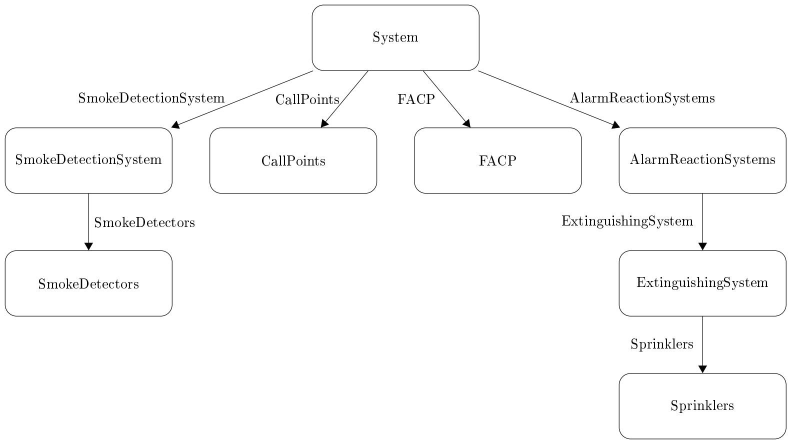


Figure 22. Page hierarchy graph for the final model

When comparing the original non-hierarchical communication diagram to the one that is obtained after a few operations of analysis, the difference is evident. The concluding model is certainly more legible and compact. Furthermore, it is much easier to modify or expand because of its modular structure. Certain components can be effortlessly replaced and the number of repeating agent instances can be freely modified. Given that the additional work needed to perform analysis operation in the Alvis model is minimal, the presented practices are highly recommended when modelling complex systems.

8 RELATED APPROACHES

As it was already mentioned Alvis LTS graphs may be verified using the CADP toolbox [12] and the nuXmv (NuSMV) model checker [6]. Developed tools provide functions to translate an Alvis LTS graph to an SMV finite state machine or the Aldebaran format automatically. Thus, Alvis may be treated as an input language for these mainstream model checkers. This section provides a comparison of Alvis and a few popular formal languages used as input language for tools such as nuXmv, LTSA, SPIN, Prism or CPN Tools.

The nuXmv (previously NuSMV) tool [6] is one of the most popular model checkers for temporal logic. Given a finite state model and a formula, nuXmv can be used to check automatically whether or not the model satisfies the formula. Formulae can be treated as a specification of requirements for a given model and can be expressed using LTL or CTL [9] temporal logics. The nuXmv tool is equipped with a dedicated modelling language which is used to define finite state transition systems [7]. The language describes a finite state system as a directed graph with nodes representing states and arcs representing transitions between states. Each state is determined by values of the system variables. Most of an SMV code describes the structure of the graph and values of variables for individual states. Compared to the SMV language Alvis describes a model at a higher level. The syntax of an Alvis model is a combination of a high level programming language and visual modelling. For such a model an LTS graph is generated which is then translated to the NuXmv representation. The corresponding SMV model preserves the structure of the LTS graph and uses a set of variables to represent states of Alvis agents.

The LTSA (Labelled Transition System Analyser [21]) is another popular model checker. A model in LTSA is composed of a set of interacting finite state machines and the requirements are also given as finite states machines. To avoid explicit description of an LTS in terms of its states, action labels and transition relation, LTSA uses a process algebra notation (FSP). The notation is similar to other process algebra formalisms like CCS [23] or CSP [13]. As it was already said, Alvis has its origins in the CCS process algebra. However, from the engineers point of view, description of a system component behaviour with a high level programming language is more convenient than using FSP calculus.

PRISM (probabilistic model checker [20]) is fairly well known model checker for several types of probabilistic models like discrete and continuous-time Markov chains, Markov decision processes, probabilistic automata and probabilistic time automata. It provides a simple, state-based language, based on the Reactive Modules formalism. The fundamental components of the PRISM language are modules and variables. A model is composed of a number of interacting modules. Every module holds its local variables which constitute the state of the module. The global state of the whole model is a sum of the local states. This approach is very similar to the Alvis one. However, the behaviour of the module is represented by a set of rules which explicitly introduce probability to the model. PRISM commands take predicate like form where user defines how variables are changing and specifies probability of such events. Alvis model behaviour is represented by imperative language. It allows user to model indeterminism but was not designed for it. Model composition and synchronization in PRISM is achieved with CSP like syntax in textual form, in Alvis it is represented as a graph. PRISM introduces a language for analysing model properties. It subsumes several temporal logics, including PCTL, CSL, PLTL and PCTL*. Alvis relays on external tools to verify generated models.

Another well established model checker is SPIN [14]. An input model is specified in Promela (Process Meta Language). Promela is an imperative language similar to Alvis. It allows for specification of concurrent processes communicating with each other through channels. Both languages support the rendezvous mechanism and the asynchronous communication through buffers. The later case has to be explicitly modelled in Alvis with passive agents. The approach to verify model properties is similar in both languages. SPIN generates a specialised model checker in the form of C source code, which after compilation is used to verify model properties. In case of Alvis a Haskell representation is used to perform on-the-fly verification using Haskell user-defined function or to generate input models for CADP or nuXmv. SPIN supports only on-the-fly model checking. On-the-fly model checking has drawback of recompilation for not only every change in the model but in requirements as well. On the other hand, the explicit LTS graph representation does not need a recreation for the new requirements check. Finally, Spin has additional GUI tools like jSpin or tau, but none of them allows for hierarchical modelling of concurrent systems.

Petri nets is the most popular formalism and coloured Petri nets (CP-nets [16]) are one of the most popular classes of Petri nets. They provide a discrete-event modelling language combining capabilities of Petri nets with the capabilities of a high-level programming language that gives the primitives for the definition of data types, variables, expressions for describing data manipulation, etc. CP-nets are supported by a modelling and verification environment called CPN Tools [17].

One of the main advantages of CP-nets is the possibility of hierarchical modelling. The idea of transitions substitution has been adopted for Alvis communication diagrams. The main difference between CP-nets and Alvis is the modelling language and the form of a model states' representation. A Petri net is a bipartite graph composed of two disjoint sets: set of places and set of transitions connecting by directed arcs. Places usually represent parts of the modelled system, while transitions its activities. A distribution of tokens in net places represents the model state. In case of CP-nets the tokens may belong to different data types. Elements of the net are labelled by expressions which describe the tokens flow. While designing a CP-net model user decides how elements of the net are interpreted. Compared to CP-nets an Alvis model usually resembles the structure of modelled system and we do not need extra hints to understand it. Moreover, the Alvis method of models states' description, which is similar to information provided by software debuggers and it is easy to understand. Equipped with a time model CP-nets may be used to model real-time systems [15, 25]. A time version of Alvis is under development [29] so verification of real-time systems with Alvis will be possible in future.

9 SUMMARY

The issue that currently inhibits the popularisation of formal methods is the fact that existing formalisms are difficult to understand for an average software engineer. In most cases, the amount of time and experience required to create a formal model of even simple system is too high to accept it in the industrial software development process. Alvis language, where the fundamental focus is to ease the use and optimization of time required to create a model, is allowing to bypass this barrier.

This paper is focused on hierarchy in Alvis models. A formal definition of hierarchical communication diagrams and methods of their transformation are introduced. Modelling with Alvis is supported by the *Alvis Editor* tool. It provides essential editing features, such as: diagram edition, basic tools for alignment and colouring, automatic creation and removal (flattening) of hierarchical pages, textual layer adition with syntax colouring and code folding. Alvis Editor is also integrated with the Alvis Compiler which allows user to create executable models directly from the editor. Alvis editor is written in Java language and the Swing graphical library. For graph rendering and management the jGraph library is used. Textual code edition is supported by the RichTextEdit library. For documentation purposes, it is possible to export diagrams into PNG, EPS and SVG formats. More information about the practical usage of the Alvis language and the tools can be found at the project web page `http://alvis.kis.agh.edu.pl`.

As regards directions for future developments, some extensions of the software are considered. First of all, a compiler for time version of Alvis will be developed. This task requires also some research on efficient algorithms for generating LTS graphs for models with time. Moreover, one of our planned future endeavours is using Alvis to explore the features of agent-based computing, e.g. such as EMAS [5], both on the algorithmic and implementation level. Some application of the agent approach can be found in [18] and [11]. In our opinion, formal verification of such systems with Alvis should provide an essential enhancement to the approach.

REFERENCES

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

[2] Baier, C.—Katoen, J.-P.: Principles of Model Checking. The MIT Press, London, UK, 2008.

[3] Balicki, K.—Szpyrka, M.: Formal Definition of XCCS Modelling Language. Fundamenta Informaticae, Vol. 93, 2009, No. 1–3, pp. 1–15.

[4] Burns, A.—Wellings, A.: Concurrent and Real-Time Programming in Ada 2005. Cambridge University Press, 2007.

[5] Byrski, A.: Tuning of Agent-Based Computing. Computer Science, Vol. 14, 2013, No. 3, pp. 491–512.

[6] Cavada, R.—Cimatti, A.—Dorigatti, M.—Griggio, A.—Mariotti, A.—Micheli, A.—Mover, S.—Roveri, M.—Tonetta, S.: The nuXmv Symbolic Model Checker. Computer Aided Verification, Springer-Verlag, LNCS, Vol. 8559, 2014, pp. 334–342.

[7] Cimatti, A.—Clarke, E.—Giunchiglia, F.—Roveri, M.: NUSMV: A new Symbolic Model Checker. International Journal on Software Tools for Technology Transfer, Vol. 2, 2000, No. 4, pp. 410–425.

[8] Ciszewski, J.—Kunecki, K.—Markowski, W.—Sawicki, J.—Sobecki, M.: SITP Guideline WP-02:2010. Fire Alarm Systems. The design, 2010.

[9] Clarke, E. M.—Grumberg, O.—Peled, D. A.: Model Checking. The MIT Press, Cambridge, Massachusetts, 1999.

[10] Emerson, E. A.: Model Checking and the Mu-Calculus. In: Immerman, N., Kolaitis, P. G. (Eds.): Descriptive Complexity and Finite Models, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Vol. 31, 1997, pp. 185–214.

[11] Faber, Ł.: Agent-Based Data Integration Frameworks. Computer Science, Vol. 15, 2014, No. 4, pp. 389–410.

[12] Garavel, H.—Lang, F.—Mateescu, R.—Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. Computer Aided Verification, Springer-Verlag, LNCS, Vol. 4590, 2007, pp. 158–163.

[13] Hoare, C. A. R.: Communicating Sequential Processes. Prentice-Hall, 1985.

[14] Holzmann, G. J.: The Model Checker SPIN. IEEE Transactions on Software Engineering, Vol. 23, 1997, No. 5, pp. 279–295.

[15] Jamro, M.—Rzonca, D.—Rząsa, W.: Testing Communication Tasks in Distributed Control Systems with SysML and Timed Colored Petri Nets Model. Computers in Industry, Vol. 71, 2015, pp. 77–87.

[16] Jensen, K.—Kristensen, L.: Coloured Petri Nets. Modelling and Validation of Concurrent Systems. Springer, Heidelberg, 2009.

[17] Jensen, K.—Kristensen, L.—Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer, Vol. 9, 2007, No. 3–4, pp. 213–254.

[18] Kisiel-Dorohinicki, M.: Evolutionary Multi-Agent Systems in Non-Stationary Environments. Computer Science, Vol. 14, 2013, No. 4, pp. 563–575.

[19] Kozen, D.: Results on the Propositional μ-Calculus. Theoretical Computer Science, Vol. 27, 1983, No. 3, pp. 333–354.

[20] Kwiatkowska, M.—Norman, G.—Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11), Snowbird, USA, 2011, pp. 585–591.

[21] Magee, J.—Kramer, J.: Concurrency: State Models & Java Programs. Wiley, 2006.

[22] Mateescu, R.—Sighireanu, M.: Efficient On-the-Fly Model-Checking for Regular Alternation-Free μ-Calculus. Tech. Rep. No. 3899, INRIA, 2000.

[23] Milner, R.: Communication and Concurrency. Prentice-Hall, 1989.

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

[25] Szpyrka, M.: Analysis of VME-Bus Communication Protocol – RTCP-Net Approach. Real-Time Systems, Vol. 35, 2007, No. 1, pp. 91–108.

[26] Szpyrka, M.—Matyasik, P.: Formal Modelling and Verification of Concurrent Systems with XCCS. Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008), Krakow, Poland, July 1–5, 2008, pp. 454–458.

[27] Szpyrka, M.—Matyasik, P.—Mrówka, R.: Alvis – Modelling Language for Concurrent Systems. In: Bouvry, P., Gonzalez-Velez, H., Kołodziej, J. (Eds.): Intelligent Decision Systems in Large-Scale Distributed Environments, Studies in Computational Intelligence, Springer-Verlag, Vol. 362, 2011, pp. 315–341.

[28] Szpyrka, M.—Matyasik, P.—Mrówka, R.—Kotulski, L.: Formal Description of Alvis Language with α^0 System Layer. Fundamenta Informaticae, Vol. 129, 2014, No. 1–2, pp. 161–176.

[29] Szpyrka, M.—Matyasik, P.—Wypych, M.: Alvis Language with Time Dependence. Proceedings of the Federated Conference on Computer Science and Information Systems, Krakow, Poland, 2013, pp. 1607–1612.

Marcin Szpyrka is Full Professor at AGH University of Science and Technology in Krakow, Poland (Department of Applied Computer Science). He is the author of over 120 publications, from the domains of formal methods, software engineering and knowledge engineering. His fields of interest also include theory of concurrency, systems security and functional programming. He is the Alvis Project leader. He is a member of the IEEE Computer Society.

Piotr Matyasik is Assistant Professor at AGH University of Science and technology, Department of Applied Computer Science. He has M.Sc. in automatics and Ph.D. in computer science. His interest covers formal methods, robotics, artificial intelligence and programming languages. Currently he is involved in Alvis project. He is the author of publications on artificial intelligence, formal methods, embedded systems and software engineering.

Jerzy Biernacki received his Bachelor's and Master's degrees in computer science from the Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, AGH University of Science and Technology, Poland, in 2013 and 2014, respectively. Currently he is a Ph.D. student at the AGH UST, Department of Applied Computer Science. His research focuses on formal methods and model checking.

Agnieszka Biernacka received her Bachelor's and Master's degrees in computer science from the Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, AGH University of Science and Technology, Poland, in 2013 and 2014, respectively. She is currently pursuing her Ph.D. degree at the AGH UST, Department of Applied Computer Science. Her research interests include formal methods and model checking.

Michał Wypych received his Master's degree in computer science from the Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, AGH University of Science and Technology, Poland, in 2012. Currently he is Ph.D. candidate and assistant at the AGH UST, Department of Applied Computer Science. His research focuses on formal methods and model checking. He is the chief developer of Alvis Compiler project.

Leszek Kotulski is Full Professor and Head of Department of Applied Computer Science at AGH-UST. His interests focus on distributed computing, graph transformations systems, data warehouses. He is the author/co-author of over 150 papers in the above-mentioned areas. He served as co-chair and PC member of many conferences and workshops worldwide. He is a member of the IEEE Computer Society, ACM, and FIPA.