A Survey of Alvis Communication Modes

Marcin Szpyrka, Jerzy Biernacki, Piotr Matyasik, and Michał Wypych

***Abstract*—Concurrent systems appear natural and intuitive solution for many real IT problems. However, designing a more complex concurrent system is a difficult task. The main problem is that for systems that have more than several subsystems it becomes difficult to control their properties at the design stage. Applications of formal methods in the development process may remarkable reduce the problem. An important issue is to choose a suitable formal modelling language, that supports the required methods of communication between subsystems. The paper provides a survey of communication modes introduced to the Alvis modelling language and discusses how the communication modes may be used while modelling concurrent systems.**

***Index Terms*—Alvis language, communication modes, concurrent systems, real-time systems**

I. Introduction

HIGH degree of concurrency makes a system more flexible, but makes difficult its verification and validation. Standard techniques, such as peer reviewing or testing are very often insufficient to guarantee the expected level of software quality in case of concurrent systems. Formal methods included into the design process may provide more effective verification techniques, reduce the verification time and system costs [1]. The key issue of concurrent systems is communication among their components. Communication makes subsystems mutually dependent and influences the correctness of the whole system.

This is the reason why communication specification mechanism is one of the most important aspects that should be considered when choosing a suitable formalism to model a given system. Every one of the most popular formal modelling languages provides such a mechanism, each having its own strengths and weaknesses.

Petri nets [2], [3], [4] are the most popular group of formal modelling languages. The very rich set of varieties of Petri net classes allows user to choose the one that suites the given problem. Petri nets do not provide specific concepts for communication modelling but are flexible enough to allow to model wide range of concurrent systems behaviours. Unfortunately, it requires a quite extensive training to achieve decent skills in using it. Moreover, Petri net models are usually very large (for more complex systems) and illegible for inexperienced users. This effectively blocks incorporating Petri nets in the standard software development process.

The second popular group of formal languages are process algebras. The most popular languages of this group are CCS (Calculus of Communicating Systems, [5], [6]), CSP (Communicating Sequential Processes, [7], [8]) and LOTOS [9]. In case of CCS and CSP calculus the synchronous communication (handshaking) between processes is considered. Processes performing the same actions are treated as communicating with each other. Although this approach is relatively simple to apply, it requires a certain amount of discipline on the part of the designer to implement. It is easy to communicate some parts of the system unintentionally. The problem is becoming a serious risk, especially when modelling complex systems.

In LOTOS communication is purely synchronous. Processes interact with each other using a concept known as gates. Even though there were attempts to add asynchronous composition to this formal language [10], the main problem with LOTOS process specification is that it is hard to understand not only for the beginners but even intermediate users. Therefore, LOTOS is not likely to be popularised among software developers.

Another popular formalism is time automata [11]. Complex concurrent systems are modelled as a product of timed automata. Parallel execution is based on synchronous execution of actions with the same labels attached.

Considering all the above, the Alvis language [12], [13], [14] seems to be a good alternative. It is formal, yet it has a concise syntax which resembles procedural languages. It supports synchronous and asynchronous communication schemes out of the box. Moreover, Alvis is not only focused on communicational part of concurrent system but also provides tools to express processing aspects of a program execution. Compared with the other formal languages, it includes the syntax that is more user-friendly from engineers' point of view, and the practical visual modelling language [14] used to define communication between concurrent subsystems.

The paper discusses communication modes supported by the current version of the Alvis language. This is an extended version of the conference paper [15]. The paper is organised as follows. Section II provides basic information about the language and tools. Section III deals with selected aspects of models semantics. Survey of communication modes is studied in Section IV. A short summary is given in the final section.

II. Alvis Language and Tools

The key notion of Alvis is the concept of *agent* borrowed from CCS [5], [16] that denotes a distinguished part of the system with its own state. 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 and mimic, to some degree, tasks and protected objects in the Ada programming

M. Szpyrka, J. Biernacki, P. Matyasik and M. Wypych are with AGH University of Science and Technology, Department of Applied Computer Science, Al. Mickiewicza 30, 30-059 Krakow, Poland (e-mail: {mszpyrka,jbiernac,ptm,mwypych}@agh.edu.pl).

ISSN 2080-8755

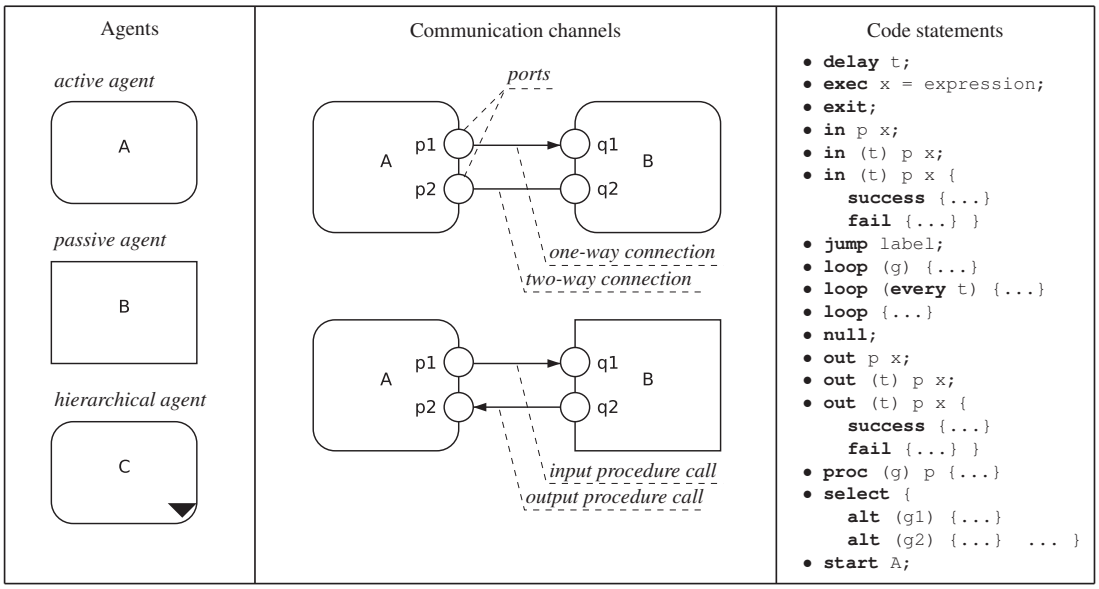


Figure 1. Elements of Alvis language

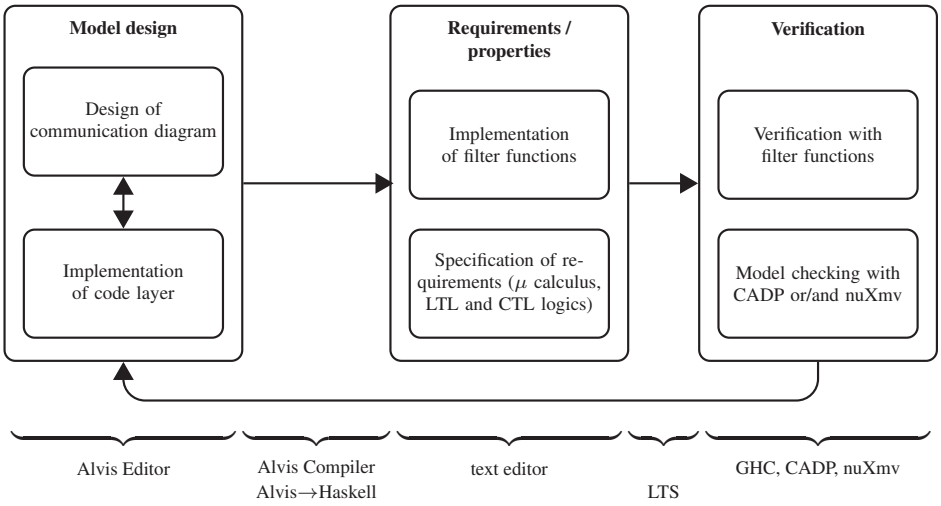


Figure 2. Modelling and verification process with Alvis

language [17]. *Active agents* may perform some activities and are treated as threads of control in a concurrent system. *Passive agents* provide a mechanism for the mutual exclusion and data synchronization. Each agent may be equipped with a set of *ports* used to communicate with other agents. Communication channels must be defined explicitly and connect two ports belonging to two agents. From the point of view of the control and data flow, an Alvis model structure is represented as a directed graph whose nodes may represent both kinds of agents, and edges represent communication channels. Moreover, to cope with complex systems a hierarchical communication diagram can be used [14]. In such a case, a node (called hierarchical agent) represents a subsystem described in details at the lower level. Behaviour of each active and passive agent is defined using a few Alvis statements supported by the Haskell functional programming language [18]. A survey of Alvis graphical components and code statements is given in Fig. 1. For more details see the manual at the project website (*http://alvis.kis.agh.edu.pl*).

The Alvis language is supported by a modelling and verification environment composed of *editor* and *compiler*. *Alvis Editor* provides the following functionality: basic hierarchy editor/viewer, visual diagram editing, textual editor with syntax highlighting, code folding and code completion. A screenshot of the editor is shown in Fig. 3. *Alvis Compiler* is used to translate the XML file into its Haskell representation. This Haskell middle-stage representation is a complete program that may be used to generate the LTS graph (*labelled transition system*) or to simulate the model behaviour.

Alvis LTS graphs can be verified using external tools supporting model checking techniques [1]. The current version of the tools cooperates with the CADP toolbox [19] and nuXmv tool [20], [21]. The former approach is action oriented and model properties are specified using the μ-calculus formulas [22]. The latter approach is states oriented and model properties are specified using the LTL and CTL temporal logics [23]. The scheme of modelling and verification process with Alvis is shown in Fig. 2.

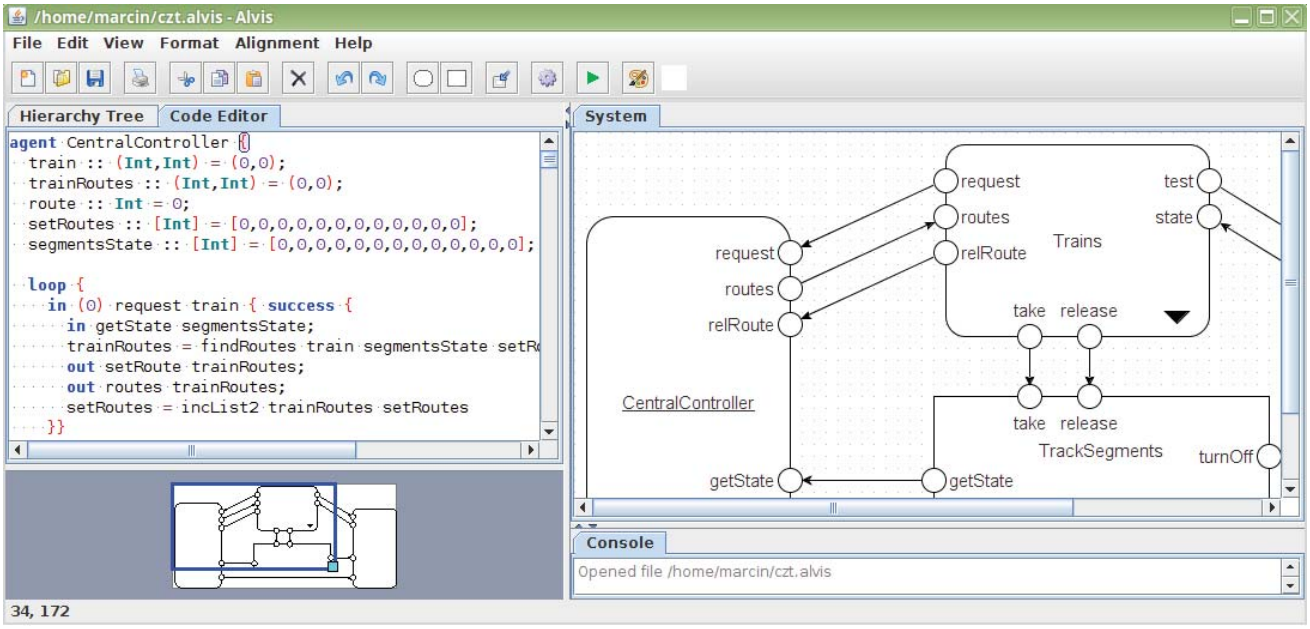


Figure 3. Alvis Editor

III. Model Semantics

A state of an Alvis model is represented as a list of states of its active and passive agents. A state of an agent is a four-tuple (agent's mode, program counter, context information list, values of parameters). The four-tuple contains all necessary information to describe an agent state unambiguously.

For active agents the agent's mode takes one of the following values: *init* (I) – the default mode for agents that are inactive in the initial state, *running* (X) – an agent is performing one of its steps (statements), *waiting* (W) – an agent is waiting for an event, e.g. for a communication with another active agent, or for a currently inaccessible procedure of a passive agent, *finished* (F) – an agent has finished its work. For passive agents the mode takes one of the two values, *waiting* (W) – an agent is inactive and waits for another agent to call one of its accessible procedures, or *taken* (T) – one of the passive agent's procedures has been called and the agent is executing it.

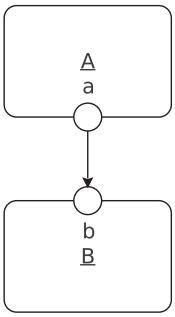
The program counter points at the next statement to be executed or the statement that has been already executed but to be completed, it needs a feedback from another agent (e.g. a communication between agents). For active agents in the *init* or *finished* mode and for passive agents in the *waiting* mode, the program counter is equal to 0.

The context information list contains additional information about the current state. Some of possible entries are given in Table I. The parameters values list contains the current values of the agent parameters.

An example of a very simple Alvis model is given in Fig. 4. The model contains two agents. Agent A sends signals (in a loop) via port a and agent B collects the signals via port b. The comments contain numbers of steps (statements). Execution of any language statement is expressed as a transition between formally defined states. All reachable states and transitions between them are represented using an LTS graph. Nodes of such a graph represent reachable states of the given model. In case of non-time models edges are labelled with single statements (see Fig. 5). In case of time models edges are labelled with a set of simultaneously executed statements. Moreover, they label provides information about the passage of time (see Fig. 6).

TABLE I
Context information list entries

Entry	Description
$in(a)$	input procedure $X.a$ is accessible (for passive agents in *waiting* mode)
$in(a)$	X waits for finalization of a communication via port $X.a$ ($X.a$ is the input port for this communication)
$out(a)$	output procedure $X.a$ is accessible (for passive agents in *waiting* mode)
$out(a)$	X waits for finalization of a communication via port $X.a$ ($X.a$ is the output port for this communication)
$proc(Y.b)$	X has called the $Y.b$ procedure and this procedure is being executed in the X agent context
$timeout(s)$	a timer signal for the statement number s has been generated and waits for serving
$timer(s,n)$	a timer signal for the statement number s will be generated in n time-units
$sft(n)$	the current step needs n time-units to be finished




```
agent A {
  loop {        -- 1
    out a;      -- 2
  }
}

agent B {
  loop {        -- 1
    in b;       -- 2
  }
}
```

Figure 4. Example of simple Alvis model

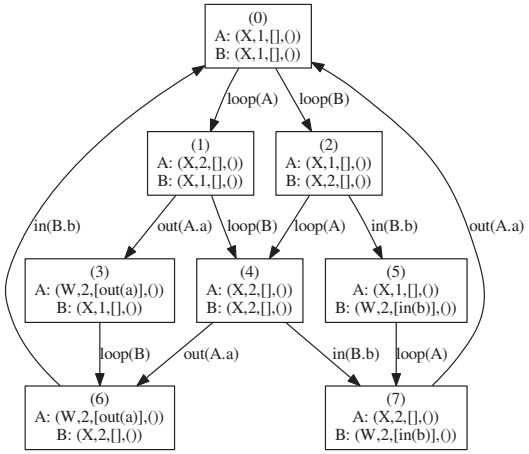


Figure 5. LTS for model from Fig. 4

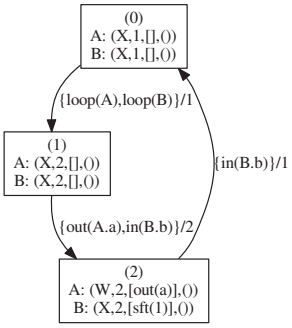


Figure 6. LTS for model from Fig. 4 (time version)

IV. COMMUNICATION

Communication modes in Alvis can be grouped according to various criteria. First of all, we can distinguish communication between two active agents and communication with a passive agent (the second side of such a communication may be both active or passive agent). When two active agents communicate with each other, they synchronise their execution. It is required for both agents to reach an appropriate state to be able to execute communication protocol. In the case of a communication with a passive agent, the communication is treated as a procedure call. The passive agent performs a service for the second agent using its context.

On the other hand, any communication may be a *pure communication* or a *value passing communication*. In case of a pure communication, a signal (without specified value) is sent between agents, while in case of a value passing communication, a value is sent. The value can be a composed type. The only difference between these communication modes is the new state of the agent that collects the value/signal. In the case of a value passing communication, value of one agent's parameter is updated. In the following part, we will focus on the pure communication only. Nevertheless, all presented concepts and remarks can also be applied to the value passing communication.

Finally, any communication may be a *blocking* or *non-blocking* one. In the instance of blocking communication, the agent that initiates the communication waits until another agent finalises it. On the other hand, when a non-blocking communication is used, the agent that initiates the communication may abandon it when the second side is not ready to finalise it within the given period of time.

In the following paragraphs all communication modes and popular constructs will be explained on abstract examples.

A. Blocking communication between active agents

Let us focus on a blocking communication between two active agents. Alvis uses two statements for the communication. The in statement for receiving data and out for sending. Each of them takes a port name as its first argument and optionally a parameter name as the second. Parameters are not used for the pure communication (value passing communication is presented in subsection IV-F). Let us focus on the example shown in Fig. 4. A communication between two active agents may be initiated by any of them. The agent that initiates it, performs the out statement to provide some information and waits for the second agent to take it, or performs the in statement to express its readiness to collect some information and waits until the second agent provides it.

Let us focus on state 4 shown in Fig. 5. Both considered agents are ready to initiate a communication between them. The A agent initiates the communication by performing the out statement. As the result of this action, A moves to the $waiting$ mode and its context list contains information about the port involved in the communication. The A agent remains in the $waiting$ mode until another agent finalises the communication.

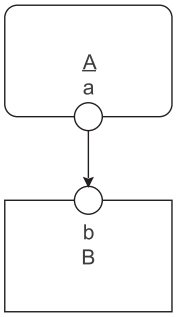
Alvis agents do not know the partners with which they communicate. Thus, the communication may be finalised by any agent connected with port $A.a$ that may collect a valueless signal or value of the suitable type in case of value passing communication. In the considered example agent B is the only one connected with the $A.a$ port and ready to finalise the communication. The edge from state 6 to 0 represents the result of performing the in statement by agent B.

The result of communication between these two agents is the same when the agent B initiates the communication (in statement between states 4 and 7) and the agent A finalises it (out statement between states 7 and 0).

B. Blocking communication between active and passive agent

Passive agents are used to store data shared among agents and to avoid the simultaneous use of such data by two or more agents. They provide a set of procedures that can be called by other agents. Each procedure has its own port attached and a communication with a passive agent via that port is treated as the corresponding procedure call. Depending on the communication direction, such a procedure may be used to send or collect some data from the passive agent. Moreover, a passive agent may also contain internal non-procedure ports. Such ports are connected with another passive agents and are used to call other procedures inside procedures of the considered agent.

Communication between active and passive agents is explained on model presented in Fig. 7 (an LTS graph for this




```
agent A {
  out a;      -- 1
}

agent B {
  proc b {
    in b;     -- 1
    exit;     -- 2
  }
}
```

Figure 7. Communication between active and passive agent

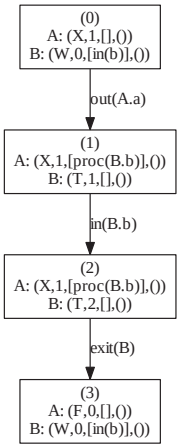


Figure 8. Communication between active and passive agent – LTS

model is shown in Fig. 8). Such a communication must be initiated by the active agent and is treated as a procedure (service) call. Let us focus on state 0. The passive agent B is in *waiting* mode and its context information list contains the list of procedures accessible in the current state. Agent A calls the a procedure using its out statement. This moves the system to state 1, where the $B.b$ procedure is executed in A context (B mode is $taken$, and A contains $B.b$ in its context information list).

A passive agent that has been taken is unavailable for all other agents. Even if it has more procedural ports, all of them are inaccessible. By definition only one active agent is allowed to operate "inside" a passive one. In the next step, the first statement of the passive agent is executed. Then communication is finished by performing $exit(B)$ statement. The agent B returns to the $waiting$ mode and b is back on the list of accessible procedures.

Generally, all communication mechanisms are analogical to the ones between two active agents. The main difference is "communication readiness" of a passive agent. It is willing to communicate when it is in the *waiting* mode and a given port is on the list of accessible procedures. Although it has not been shown in the presented example, in general, if a passive agent has more than one port, not all of them might be ready for communication at a given state. User has a possibility to associate *guards* (logical expressions) with ports to limit their accessibility.

C. Non-blocking communication

In case of a non-blocking communication, the communication may be abandoned if the second agent is not ready to finalise it. The in and out statements for non-blocking communication contain additional parameter that represents how long the corresponding agent may wait for finalisation of the communication. In case of non-time Alvis models, the time parameter is always treated as 0. Thus, for a non-timed model, a non-blocking communication may be used to finalise the communication only.

Let us consider the model structure from Fig. 4 with the following code layer:

```
agent A {
  out (0) a;    -- 1
}
```

```
agent B {
  in b;      -- 1
}
```

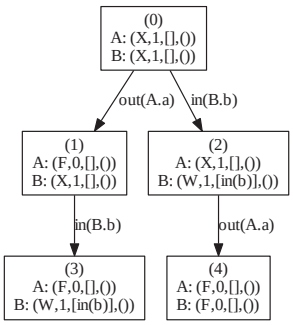


Figure 9. Model with a non-blocking communication – LTS graph

The parameter 0 in the out statement means that agent A cannot wait for finalisation of the communication – communication must be finalised immediately or is abandoned. In other words, if this statement does not finalise the communication, agent A abandons it and executes further statements. In our example, executing of the out statement in state 0 from Fig. 9 leads to state 1, where agent A is already in the $finish$ mode and the communication between A and B will never be finalised. On the other hand, if agent B has already initiated a communication (state 2), agent A can finalise it (state 4). The non-blocking in statement works similarly.

D. Non-blocking communication in timed model

Let us consider the model structure from Fig. 4 with the following code layer:

```
agent A {
  null;         -- 1
  out (3) a;    -- 2
}
```

```
agent B {
   null;      -- 1
   in b;      -- 2
}
```

If we consider models with time, the first argument of a non-blocking communication statement denotes how long the corresponding agent may wait for the finalisation of the communication.

Suppose, that duration of all statements is equal to 1 except the $null(A)$ statement that lasts 3 time-units. Thus, the B agent will finish its in statement before A will start its out statement (see Fig. 10a). This time the out statement finalises the communication immediately.

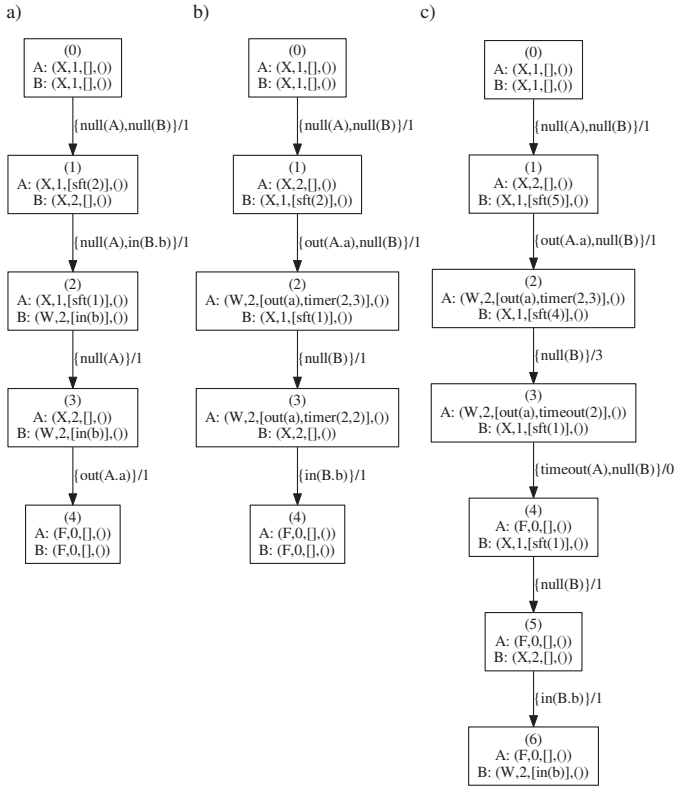


Figure 10. Three variants of the LTS graph of the considered timed model depending on the duration of individual statements

Assume now that the $null(B)$ statement lasts 3 time-units and other statements last 1 time-unit. After executing the out statement A moves to the $waiting$ mode (see Fig. 10b) and waits no more than 3 time-units for another agent to finalise the communication (more precisely, to start finalising). This time agent B manages to finalise the communication before the deadline. If the $null(B)$ statement lasts 6 time-units, no agent finalises the communication, it is abandoned and A performs its further statements if any (see Fig. 10c).

E. Fail and Success clause in non-blocking communication

The in and out non-blocking statements may be equipped with $success$ and $fail$ clauses (both are optional). The $success$ clause is executed upon successful communication, while the $fail$ clause is executed, if the communication has been abandoned. Let us consider the model structure from Fig. 4 with the following code layer:

```
agent A {                          agent B {
  out (0) a{            -- 1         in b;     -- 1
    fail {out (0) a;} -- 2         }
  }
}
```

The effect of addition the *fail* clause is visible in Fig. 11. Agent A instead of abandoning efforts to send signal to agent B after the first failure (state 1) is trying to send it once again. This gives agent B opportunity to initiate the communication (state 4) and the communication to be finalised (state 5).

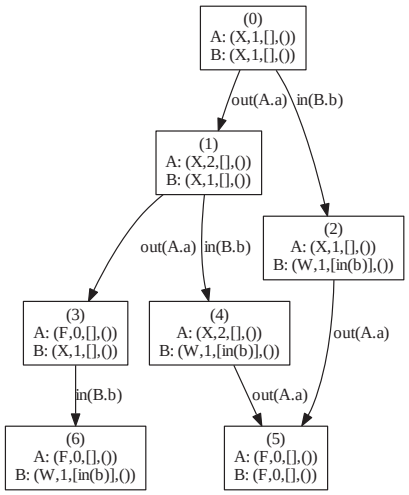


Figure 11. Result of application of a fail clause – LTS

F. Value passing communication

All models presented in this section so far were examples of pure communication that means they did not send values of any type, just signals. Such choice was not enforced by the limitation of the language but by the need of model simplicity, which would allow to focus on specific aspects of communication. Every one of these models could use a value passing mode instead. There is no significant difference between these two modes other than the fact that model designer has to remember about ensuring the values sent and received are of the same type. The Alvis language offers a number of data types that can be used during the modelling. The list of selected, basic Haskell types recommended to be used in Alvis is as follows:

- *Char* – Unicode characters.
- *Bool* – Values in Boolean logic (True and False).
- *Int* – Fixed-width integer values – The exact range of values represented as Int depends on the system's longest native integer.
- *Double* – Float-point numbers typically 64 bits wide and uses the system's native floating-point representation.

The basic types can be combined into the composite data types. The most common ones are *lists* and *tuples*. A *list* is a sequence of elements of the same type, with the elements being enclosed in square brackets and separated by commas. A *tuple*, on the other hand, is a sequence of elements of possibly different types, with the elements being enclosed in parentheses and separated by commas. There is also a *String* data type, which is in fact a list of *Char* values.

Let us once again consider the model structure from Fig. 4 with the following code layer:

```
agent A {                          agent B {
  v :: Int = 1;                      v :: Int = 0;
  out (0) a v;    -- 1               in b v;          -- 1
}                                  }
```

The first line of both agent blocks contains a declaration of the v parameter of type *Int* with its initial value. Declarations of the parameters are not a part of the agent body and are not regarded as statements by the program counter. The result of

using value passing communication is shown in Fig. 12. In case of successful finalisation of the communication between the agents (state 4), the v parameter of agent B changes value from 0 to 1.

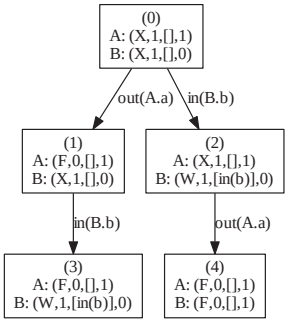
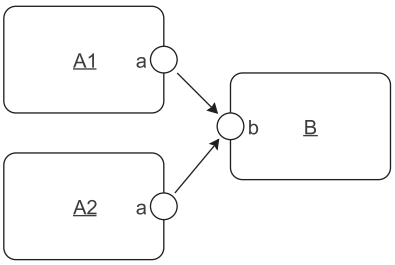


Figure 12. Result of application of value passing communication – LTS

G. Indeterminism in communication

Let us consider the example presented in Fig. 13 and its LTS graph shown in Fig. 14. In the given example agents $A1$ and $A2$ have identical behaviour specifications. In Alvis language it is possible to share one definition among a few agents and describe multiple agents in a single agent block. In such a case, a few agents' names are placed after the keyword agent separated by commas.




```
agent A1, A2 {
  out a;     -- 1
}
```

```
agent B {
in b;      -- 1
}
```

Figure 13. Example of indeterminism in communication

Let us focus on the state 3 of the LTS graph. Agent B is *waiting* after execution of a blocking *in* statement. Because both $A1$ and $A2$ agents are ready to finalise the communication the system behaviour is indeterministic. The signal provided by the B agent can be collected by either $A1$ or $A2$ (see states 5 and 6).

To eliminate indeterminism, priorities are being introduced into Alvis [24]. The basic concept is based on a simple principle of selecting for execution an agent with the highest priority. Therefore, user is able to determine which agents should be winning data races. One of the latest enhancements introduced in the Alvis language is the possibility to apply custom, user-defined priority management algorithms.

Moreover, Alvis allows to introduce fairness. Considering the latest example, to achieve fairness in providing data from both A agents system should be redesigned. One of the most

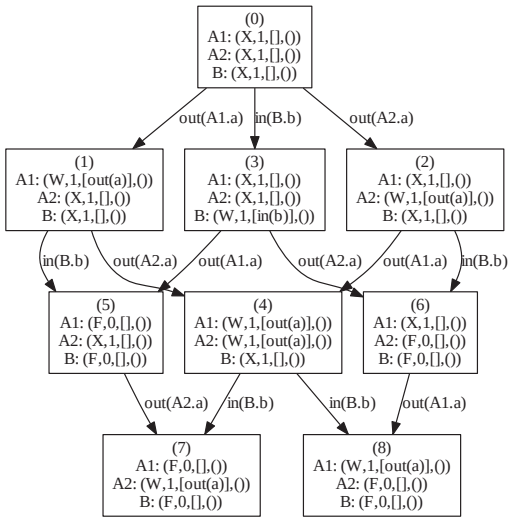
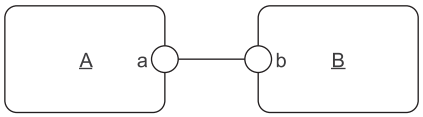


Figure 14. Indeterminism in communication – LTS

obvious solutions is to equip B with dedicated ports for receiving data from all providers. Then it would be able to internally choose, from which agent it should receive the next data package to process. Another solution may introduce an additional agent to provide fair data supply from both sources.

H. Send value and wait for response construct

Having learned the basic communication modes one can easily model complex behaviour of concurrent systems. This and the following subsections will focus on presenting two examples of common, yet not complicated constructs used in concurrency modelling. They aim to explain how the presented communication modes can be utilised in order to achieve certain effects.




```
agent A {
  v :: Int = 1;
  out a v;        -- 1
  in a v;         -- 2
}
```

```
agent B {
  v :: Int = 0;
  in b v;          -- 1
  v = v * 2;       -- 2
  out b v;         -- 3
}
```

Figure 15. Send value and wait for response construct

Fig. 15 contains a simple model of a *send and wait for response* construct. The LTS graph of this model is presented in Fig. 16. Agent A sends to agent B a synchronous request to process its parameter v. When agent B receives a value (state 3) it doubles it (state 5 or 6) and sends back to agent A. It is worth noticing that the communication channel between the agents is a two-way connection, which allows the communication to flow in both directions. Agent A is waiting for the response until the communication is eventually finalised in state 8. This final state of the LTS graph shows that the initial value of agent A parameter was doubled as expected.

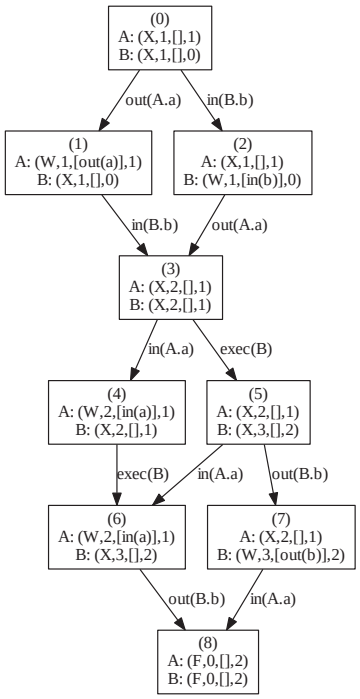


Figure 16. Send value and wait for response construct – LTS

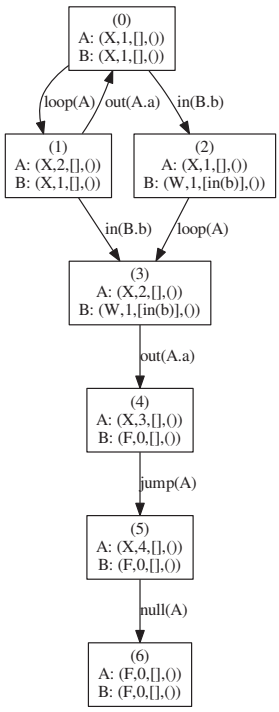


Figure 17. Busy waiting construct – LTS

I. Busy waiting construct

Busy waiting is a common technique especially in multiprocessor systems. It suspends a process in a tight loop without relinquishing CPU until a specified condition is satisfied, e.g contested resource is released or a device status is changed. It offers a very good response time to one particular signal, at the expense of fully occupying the processor and wasting CPU cycles that other process might use productively.

Let us consider the model structure from Fig. 4 with the following code layer:

```
agent A {                          agent B {
  loop{                  -- 1        in b;   -- 1
    out (0) a{           -- 2      }
      success{
        jump off;}}}     -- 3
  off:
  null;                  -- 4
}
```

The LTS graph for this model is shown in Fig. 17. Agent A contains a tight loop in which it repeatedly checks if anybody wants to collect a signal from port a (states 0 and 1). When agent B eventually initialises the communication, agent A executes the *success* clause connected to its *in* statement (state 4) and exits the loop (state 5).

V. Conclusion

A survey of communication techniques available in Alvis has been presented in the paper. They make Alvis suitable for modelling concurrent or distributed systems with different methods of communication between components. Moreover, what distinguishes Alvis among other formal modelling languages is the straightforward syntax both for visual modelling and defining behaviour of agents. Alvis equally supports the modelling of communication between components of a complex system and describing of data processing. The Alvis language allows for the construction of models at a high level of abstraction, as well as at level close to their target implementation. These features give Alvis a potential to be popularised and commonly used in the design and development process of complex concurrent and distributed systems.

References

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

[2] I. Grobelna, M. Wiśniewska, R. Wiśniewski, M. Grobelny, and P. Mróz, "Decomposition, validation and documentation of control process specification in form of a Petri net," in *7th International Conference on Human System Interactions (HSI)*, Lisbon, Portugal, 2014, pp. 232–237.

[3] M. Jamro, D. Rzonca, and W. Rząsa, "Testing communication tasks in distributed control systems with SysML and timed colored Petri nets model," *Computers in Industry*, vol. 71, pp. 77–87, 2015.

[4] M. Szpyrka, J. Biernacki, and A. Biernacka, "Tools and methods for RTCP-nets modelling and verification," *Archives of Control Sciences*, vol. 26, no. 3, pp. 339–365, 2016.

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

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

[7] C. A. R. Hoare, *Communicating sequential processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

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

[9] ISO, "Information processing systems, open systems interconnection LOTOS," Tech. Rep. ISO 8807, 1989.

[10] N. Maneerat, R. Varakulsiripunth, D. Seki, K. Yoshida, K. Takahashi, Y. Kato, B. Bista, and N. Shiratori, "Composition method of communication system specifications in asynchronous model and its support system," in *Proceedings of 9th IEEE International Conference on Networks*, 2001, pp. 64–69.

[11] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," *Lecture Notes on Concurrency and Petri Nets*, vol. 3098, 2004.

[12] M. Szpyrka, P. Matyasik, and R. Mrówka, "Alvis – modelling language for concurrent systems," in *Intelligent Decision Systems in Large-Scale Distributed Environments*, ser. Studies in Computational Intelligence, P. Bouvry, H. Gonzalez-Velez, and J. Kołodziej, Eds. Springer-Verlag, 2011, vol. 362, ch. 15, pp. 315–341.
[13] M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski, "Formal description of Alvis language with α^0 system layer," *Fundamenta Informaticae*, vol. 129, no. 1-2, pp. 161–176, 2014.
[14] M. Szpyrka, P. Matyasik, J. Biernacki, A. Biernacka, M. Wypych, and L. Kotulski, "Hierarchical communication diagrams," *Computing and Informatics*, vol. 35, no. 1, pp. 55–83, 2016.
[15] P. Matyasik, M. Szpyrka, M. Wypych, and J. Biernacki, "Communication between agents in Alvis language," in *Proc. of Mixdes 2016, the 23nd International Conference Mixed Design of Integrated Circuits and Systems*, Łódź, Poland, June 23–25 2016, pp. 448–453.
[16] M. Szpyrka and P. Matyasik, "Formal modelling and verification of concurrent systems with XCCS," in *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008)*, Krakow, Poland, July 1-5 2008, pp. 454–458.
[17] A. Burns and A. Wellings, *Concurrent and real-time programming in Ada 2005*. Cambridge University Press, 2007.
[18] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. Sebastopol, CA, USA: O'Reilly Media, 2008.
[19] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A toolbox for the construction and analysis of distributed processes," in *Computer Aided Verification (CAV'2007)*, ser. LNCS, vol. 4590. Berlin, Germany: Springer, 2007, pp. 158–163.
[20] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *Computer Aided Verification*, ser. LNCS. Springer-Verlag, 2014, vol. 8559, pp. 334–342.
[21] J. Biernacki, "Alvis models of real-time safety critical systems state-base verification with nuXmv," in *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, ser. ACSIS, vol. 8, 2016, pp. 1701–1708.
[22] E. A. Emerson, "Model checking and the Mu-calculus," in *Descriptive Complexity and Finite Models*, ser. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, N. Immerman and P. G. Kolaitis, Eds. American Mathematical Society, 1997, vol. 31, pp. 185–214.
[23] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.
[24] M. Szpyrka, J. Biernacki, A. Biernacka, and M. Wypych, "Priority management in Alvis language," in *Proc. of Mixdes 2016, the 23nd International Conference Mixed Design of Integrated Circuits and Systems*, Łódź, Poland, June 23–25 2016, pp. 464–468.

Marcin Szpyrka is a 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 leader of the Alvis Project. He is a member of the IEEE Computer Society and the Polish Artificial Intelligence Society (PSSI).

Jerzy Biernacki received his Bachelor and Master 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, a PhD student at the AGH UST, Department of Applied Computer Science. His research focuses on formal methods and model checking.

Piotr Matyasik is Assistant Professor at AGH University of Science and technology, Department of Applied Computer Science. 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.

Michał Wypych received his Master 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, PhD candidate and assistant at the AGH UST, Department of Applied Computer Science. His research focuses on formal methods and model checking. He is chief developer of Alvis Compiler project.