PRACTICAL APPROACH TO MODELLING AND VERIFICATION OF CONCURRENT SYSTEMS WITH ALVIS*

Marcin Szpyrka, Piotr Matyasik, Rafał Mrówka
AGH University of Science and Technology
Department of Automatics
Al. Mickiewicza 30
30-059 Krakow, Poland
Email: {mszpyrka,ptm,Rafal.Mrowka}@agh.edu.pl

KEYWORDS

Alvis modelling language, concurrent systems modelling, formal verification, CADP

ABSTRACT

The paper presents a practical introduction to the modelling and verification of concurrent systems with the Alvis modelling language using the α^0 system layer. This version of Alvis is the most universal one. It is assumed that each active agent has access to its own processor and all agents perform their statements concurrently. All layers of an Alvis models are shortly described in the paper and possibilities of a formal verification are also discussed. A classical problem of dining philosophers is presented to illustrate Alvis features and methods of an automatic model verification with the CADP toolbox.

INTRODUCTION

Standard techniques, such as peer reviewing or testing are very often insufficient to guarantee an expected level of software quality in case of concurrent systems. Formal methods included into the design process may provide more effective verification techniques, and may reduce the verification time and system costs. Unfortunately, formal methods are very seldom used in real IT projects, due to their specific mathematical syntax.

Alvis (Szpyrka et al., 2011a) is a novel modelling language designed by our team especially for concurrent systems. Alvis has its origins in the CCS process algebra (Milner, 1989), (Aceto et al., 2007), and the XCCS modelling language (Balicki and Szpyrka, 2009), (Matyasik, 2009). In contrast to process algebras, Alvis uses a high level programming language based on the Haskell syntax, instead of algebraic equations, and provides a hierarchical graphical modelling for defining interconnections among agents.

The aim of the paper is to provide a practical introduction to the modelling and verification of concurrent systems with Alvis. The subsequent sections provides information about:

- comparison of Alvis with other languages used for embedded systems development;
- layers of Alvis models;
- basic information about states of Alvis models, transitions among states and LTS graphs used for verification purposes;
- methods of a formal verification with the CADP toolbox.

COMPARISON WITH OTHER LANGUAGES

Alvis has its origins in the CCS process algebra (Milner, 1989), (Fencott, 1995), (Aceto et al., 2007) and the XCCS language (Balicki and Szpyrka, 2009), (Matyasik, 2009). The main result of the fact is the communication model used in Alvis that is similar to the one used in CCS and the rendes-vous mechanism used in Ada (Barnes, 2006). However, Alvis uses a simplified rendez-vous mechanism with equal agents without distinguishing servers and clients. In contrast to Ada, Alvis does not support asynchronous procedure calling, a procedure uses always an active agent context.

A few constructs in Ada were an inspiration while developing Alvis language. For example, protected objects have been used to define passive agents and the Ada *select statement* has been used to define the Alvis *select statement*. An Alvis model composed of few agents that work concurrently is similar to an Ada distributed system. Active agents can be treated as processing nodes, while passive agents as storage ones.

Alvis has many features in common with E-LOTOS – an extension of the LOTOS modelling language (ISO, 1989). First of all, Alvis as E-LOTOS is derived from process algebras. Alvis, like E-LOTOS, was intended to allow a formal modelling and verification of distributed real-time systems. In contrast to E-LOTOS, Alvis provides graphical modelling language. Moreover, Alvis Toolkit supports a LTS graph generation, which significantly simplifies the formal verification of models.

Alvis has also many features in common with System Modelling Language (SysML)(Sys, 2008) – a general purpose modelling language for systems engineering applications. It contains concepts similar to SysML

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

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

Proceedings 25th European Conference on Modelling and Simulation ©ECMS Tadeusz Burczynski, Joanna Kolodziej Aleksander Byrski, Marco Carvalho (Editors)
ISBN: 978-0-9564944-2-9 / ISBN: 978-0-9564944-3-6 (CD)

ports, property blocks, communication among the blocks and hierarchical models. Unlike SysML, Alvis combines structure diagrams (block diagrams) and behaviour (activity diagrams) into a single diagram. In addition, Alvis defines formal semantics for the various artifacts, which is not the case in SysML.

Due to the use of Ada origins, VHDL (Ashenden, 2008) and Alvis have a similar syntax for the communication and parallel processing. The concept of agent in Alvis is also similar to a design entity in VHDL and both languages use ports for a communication among system components. It should be noted, however, that Alvis is closely linked with its graphical model layer. Graphical composition allows for easier identification of the system hierarchy and components. The main purpose of VHDL is the specification of digital electronic circuits and it focuses on systems hardware. However, Alvis integrates the hardware and software views of an embedded system.

In contrast to synchronous programming languages like Esterel (Berry, 2000), (Palshikar, 2001) or SCADE (SCA, 2007), Alvis does not use the broadcast communication mechanism. Only agents connected with communication channels can communicate one with another.

MODELS

An Alvis model is composed of three layers:

Graphical layer – is used to define data and control flow among distinguished parts of the system under consideration that are called *agents*. The layer takes the form of a hierarchical graph with nodes representing agents.

Code layer – is used to describe the behaviour of individual agents. It uses both Haskell functional programming language (O'Sullivan et al., 2008) and original Alvis statements.

System layer – depends on the model running environment i.e. the hardware and/or operating system. The layer is the predefined one and it is necessary for a model simulation and verification.

To present the most important features of Alvis the well-known problem of dining philosophers has been chosen. Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. There is a large bowl of spaghetti in the centre of the table. There are also five plates at the table and five forks set between the plates. Eating the spaghetti requires the use of two forks. Each philosopher thinks. When he gets hungry, he picks up the two forks that are closest to him. If a philosopher can pick up both forks, he eats for a while. After a philosopher finishes eating, he puts down the forks and starts thinking.

System layer

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

User can choose one of a few versions of the layer and it affects the developed model semantic. System layers differ about the scheduling algorithm and system architecture mainly. There are two approaches to the scheduling problem considered. System layers with α symbol provide a predefined scheduling function that is called after each step automatically. On the other hand, system layers with β symbol do not provide such a function. A user must define a scheduling function himself.

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

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

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

Graphical layer

The graphical layer takes the form of a communication diagram (Szpyrka et al., 2011b) i.e. a hierarchical graph whose nodes represent agents. Agents are divided into *active* (rounded boxes) and *passive* ones (rectangles). *Active agents* are treated as threads of control in a concurrent system, while passive agents represent shared resources with mutual exclusion access. Communication diagrams are the only way, in Alvis, to point out agents that communicate one with another. Moreover, the diagrams allow programmers to combine sets of agents into modules that are also represented as agents (called *hierarchical ones*).

Agents communicate one with another using *ports* drawn as circles placed at the edges of the corresponding rounded box or rectangle. A communication is possible only through defined communication channels drawn as lines (or broken lines) between ports. An arrowhead points out the input port for the particular connection.

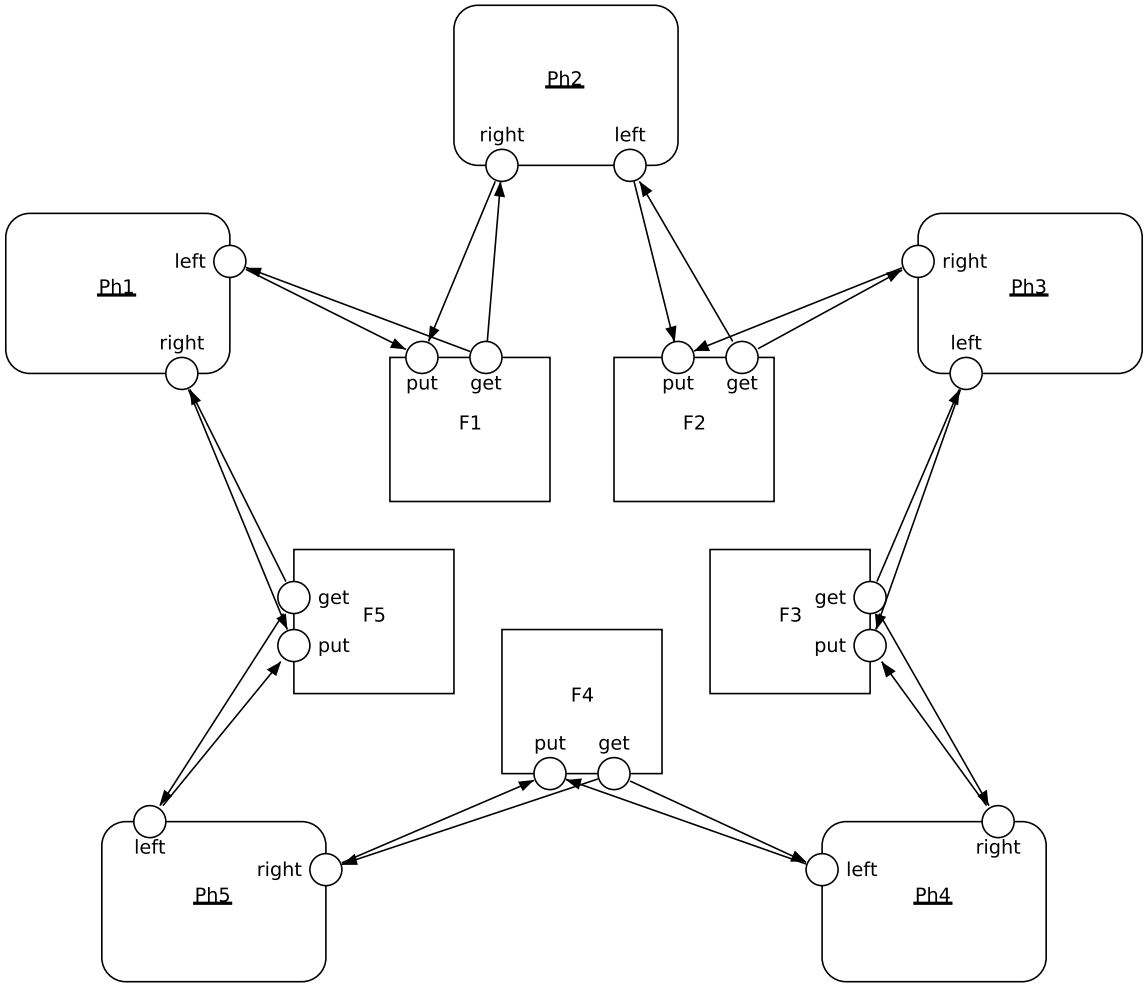

Figure 1: Dining philosophers – communication diagram

Communication channels without arrowheads represent pairs of connections with opposite directions.

The communication diagram for the considered model of dining philosophers is shown in Fig. 1. It contains 5 active ($Ph1, \ldots, Ph5$) and 5 passive ($F1, \ldots, F5$) 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.

Code layer

Code layer is used to describe the behaviour of individual agents. The layer uses both Haskell functional language and original Alvis statements. The list of Alvis statements used with the α^0 system layer is presented in Table 1. We have omitted statements explicitly related to time e.g., *loop every* or *delay* that are used for real-time programming. Discussing time dependences is out of the scope of the paper. For more information and a detailed formal description of all presented statements see (Szpyrka et al., 2011a) or (Szpyrka et al., 2011b).

```
agent Ph1, Ph2, Ph3, Ph4, Ph5 {
  loop {                              -- 1
    in right;                         -- 2
    in left;                          -- 3
    out right;                        -- 4
    out left;                         -- 5
  }
}

agent F1, F2, F3, F4, F5 {
  taken :: Bool = False;
  proc (taken == False) get {
    taken = True;                     -- 1
    out get; }                        -- 2
  proc (taken == True)  put {
    taken = False;                    -- 3
    in put; }                         -- 4
  }
}
```

Listing 1: Dining philosophers – code layer

The code layer for the considered model of dining philosophers is presented in Listing 1. In this approach

Table 1: Alvis statements used with the α^0 system layer (time statements omitted)

Statement	Description
exec x = expression	Evaluates the *expression* and assigns the result to the parameter; the *exec* keyword can be omitted.
exit	If an active agent performs the statement, it is terminated. If a passive agent performs the statement, its current procedure is terminated.
if (g1) {...} **elseif** (g2) {...} ... **else** {...}	Conditional statement.
in p **in** p x	Collects a signal (without value) via the port p. Collects a value via the port p and assigns it to the parameter x.
jump label	Transfers the control to the line of code identified with the *label*.
loop (g) {...} **loop** {...}	Repeats execution of the contents while the guard if satisfied, the guard is checked everytime before entering the loop contents. – It is similar to the while loop in most languages. Infinite loop.
null	Empty statement.
out p **out** p x	Sends a signal (without value) via the port p. Sends the value of the parameter x via the port p; a literal value can be used instead of a parameter.
proc (g) p {...}	Defines the procedure for the port p of a passive agent. The guard is optional.

select {
alt (g1) {...}
alt (g2) {...}
...
}
	Selects one of the alternative choices. Guards $g1, g2, \ldots$ decide which alternatives can be chosen after entering the *select* statement.

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

philosophers try to take their right fork before the left one. All agents $Ph1, \ldots, Ph5$ share the same behaviour definition. Forks modelled as passive agents provide two procedures – *get* for taking a fork, and *put* for putting them back. The *taken* parameter is used to control the procedures accessibility.

We consider behaviour of Alvis models at the level of detail of single steps. Statements such as *exec, exit, in, jump, null, out* and *start* are *single-step* statements. On the other hand, *if, loop* and *select* are *multi-step* statements. We use recursion to count the number of steps for multi-step statements. For each of these statements, the first step enters the statement interior. Then, we count steps of statements put inside curly brackets. Comments included into the considered code layer contain step numbers. For example, one cycle of a philosopher activity consists of 5 steps: 1) entering the loop, 2) *in* statement (port *right*), etc.

MODELS DYNAMIC

States

A state of a model is represented as a sequence of agents' states. To describe the current state of an agent we need a tuple with four pieces of information: *agent mode* (am), *program counter* (pc), *context information list* (ci) and *parameters values tuple* (pv).

A passive agent is always in one of two modes: *waiting* or *taken*. The former one means that the agent is inactive and waits for another agent to call one of its accessible procedures. In such a situation its pc is equal to zero and ci contains names of accessible procedures. The *taken* mode means that one of the passive agent procedures has been called and the agent executes it. In such a case, ci contains the name of the called procedure (i.e. the name of the port used for current communication). The pc points out the index of the next statement to be executed or the current statement if the corresponding active agent is *waiting*.

If α^0 system layer is considered, an active agent can be in one of the following modes: *finished, init, running, waiting*. The *init* mode means that an agent has not started its activity yet, while the *finished* one means that it has already finished its work. The *waiting* mode means that an active agent is waiting either for a synchronous communication with another active agent or for a currently inaccessible procedure of a passive agent, and the *running* mode means that an agent is performing one of its steps. In case of the *waiting* or *running* mode, ci contains additional information about the events an agent is waiting for, or about a passive agent that uses the considered active agent context. For any agent, pv contains the current values of the agent parameters.

A detailed description of agents states can be found

in (Szpyrka et al., 2011b).

Transitions

Transitions describe execution of single steps in an Alvis model. The transitions list for models with the α^0 system layer is given in Table 2.

Table 2: Set of transitions

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

```
0:
Ph1: (running,1,[],())
Ph2: (running,1,[],())
Ph3: (running,1,[],())
Ph4: (running,1,[],())
Ph5: (running,1,[],())
F1: (waiting,0,[out(get)],(False))
F2: (waiting,0,[out(get)],(False))
F3: (waiting,0,[out(get)],(False))
F4: (waiting,0,[out(get)],(False))
F5: (waiting,0,[out(get)],(False))
```

Listing 2: Structure of the code layer

The initial state for the considered model of dining philosophers is presented in Listing 2. All active agents are running and are about to execute the t_{loop} transition. All passive agents are waiting for a communication through their *get* port. Suppose the $Ph2$ agent executes its first step. Its state changes into

```
Ph1: (running,2,[],())
```

while states of other agents remain unchanged. Then, if the same agent executes the t_{in} transition, states of $Ph1$ and $F5$ agents change into:

```
Ph1: (running,2,[proc(F5.get)],())
F5:  (taken,1,[out(get)],(False))
```

It means that:

- $F5$ is running in the $Ph1$ agent's context ($am(Ph1) = \mathit{running}$ and $\mathit{proc}(F5.get) \in ci(Ph1)$);
- the *get* procedure of $F5$ has been called ($am(F5) = \mathit{taken}$ and $ci(F5) = [out(get)]$);
- the first step of the procedure is about to execute ($pc(F5) = 1$);

LTS graphs

States of an Alvis model and transitions among them are represented using a labelled transition system (LTS graph for short). An LTS graph is an ordered graph with nodes representing states of the considered system and edges representing transitions among states.

Due to practical reasons, such an LTS graph generated automatically for an Alvis model takes the textual form. Then it is converted into the *Binary Coded Graphs* (BCG) format and used as input data for the CADP toolbox (Garavel et al., 2007). CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking.

The LTS graph generated for the five philosophers example has the following properties (as reported by CADP tool) :

- 111486 states,
- 447735 transitions,
- 31 labels,
- 1 deadlocked state,
- provides deterministic behaviour for all labels.

The detailed analysis of the generated LTS graph is discussed in the next section.

VERIFICATION WITH CADP

Deadlocks belong to basic properties in the formal verification domain. As it was shown before the verified system has one deadlock. The CAPD tool can provide not only the number of deadlocks but also paths leading to them. Finding deadlocks in this tool can be achieved by running a built-in function via the Ecalyptus graphical environment or by checking a property specified in the regular alternation-free $\mu - calculus$ (Garavel et al. (2007)) (In fact CADP allows users to check models with other formalisms but only the alternation-free $\mu - calculus$ is used in this article). The following formula is used to check whether each state has at least one successor.

```
[ true* ] < true > true
```

Obviously, this property does not hold in the presented example. As a proof CADP returns the following path leading to the deadlocked state:

```
"loop(Ph1)" -> "loop(Ph2)" ->
"loop(Ph3)" -> "loop(Ph4)" ->
"loop(Ph5)" -> "in(Ph1)"   ->
"in(Ph2)"   -> "in(Ph3)"   ->
"in(Ph4)"   -> "in(Ph5)"   ->
"exec(F1)"  -> "exec(F2)"  ->
"exec(F3)"  -> "exec(F4)"  ->
"exec(F5)"  -> "out(F1)"   ->
"in(Ph2)"   -> "out(F2)"   ->
"in(Ph3)"   -> "out(F3)"   ->
"in(Ph4)"   -> "out(F4)"   ->
"in(Ph5)"   -> "out(F5)"   ->
"in(Ph1)"
```

The `"loop(Ph1)"` label stands for a *loop* statement executed by the first philosopher, `"out(F3)"` stands for an *out* statement executed by the passive agent $F1$, etc.

Liveness is another commonly used system property. It can be expressed as "something good eventually happens". Let us define "something good" in terms of the presented example:

```
< true* . "in(Ph1)" .
  true* . "in(Ph1)" > true
```

In this property specification "something good" takes the form of eating. Eating requires taking both forks which is represented by executing in sequence methods of passive agents that represent forks. Calling a passive agent procedure is performed by executing the *in* statement. In other words, the question is if there is a path where the philosopher number one executes the *in* statement twice. This property is true and the CADP's verification engine can provide a trace that proves it.

```
@ ( true* . "in(Ph1)" .
    true* . "in(Ph1)"
    and not ( "in(Ph2)" or
              "in(Ph3)" or
              "in(Ph4)" or
              "in(Ph5)"
            )
  )
```

The above property is true and the proof is presented in Fig. 2. To save space most of the states leading to the interesting circle was omitted.

Another interesting situation in system modelling is a livelock. It takes a place when a system is consequently processing some statements but without doing any useful work. As it was assumed before, a desirable behaviour of an agent can be represented by picking a fork. Thus, an example of a livelock is a situation when there exists a cycle where no *in* statement is executed – forks remain on the table all the time. Let us consider the following formula:

```
@ ( not ( "in(Ph1)" or
          "in(Ph2)" or
          "in(Ph3)" or
          "in(Ph4)" or
          "in(Ph5)"
        )
  )
```

According to the system specification (see Fig. 1 and Listing 1) the above property is false. After executing at most five steps an *in* statement has to be performed. However, a livelock can be easily introduced by using the *select* statement with a *delay* branch or the *ready* statement in a guard (Szpyrka et al. (2011b)).

SUMMARY

The article presents an example of the practical application of the Alvis language and a basic informal knowledge about its syntax and semantic. The included example presents how an Alvis model can be designed and verified. Alvis evolved from process algebras and is in fact a formal modelling language but it takes the form of an imperative programming language. Such a representation seems to be more convenient from the engineering point of view. It should be underlined that the example presented in this paper refers to the α^0 system layer, when Alvis behaves similar to the CCS process algebra. Other system layers provide ability to check a developed system in exactly specified circumstances like the number of processors or a scheduling algorithm. The results obtained from such a verification may differ from the presented ones. Moreover, they will represent the system behaviour taking into account a chosen computer environment, not an abstract parallel execution.

References

(2007). *Welcome to SCADE 6.0*. Esterel Technologies SA.

(2008). *OMG Systems Modeling Language (OMG SysML)*. Object Management Group.

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

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

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

Barnes, J. (2006). *Programming in Ada 2005*. Addison Wesley.

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

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

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

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

Matyasik, P. (2009). *Design and analysis of embedded systems with XCCS process algebra*. PhD thesis, AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Kraków, Poland.

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

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

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

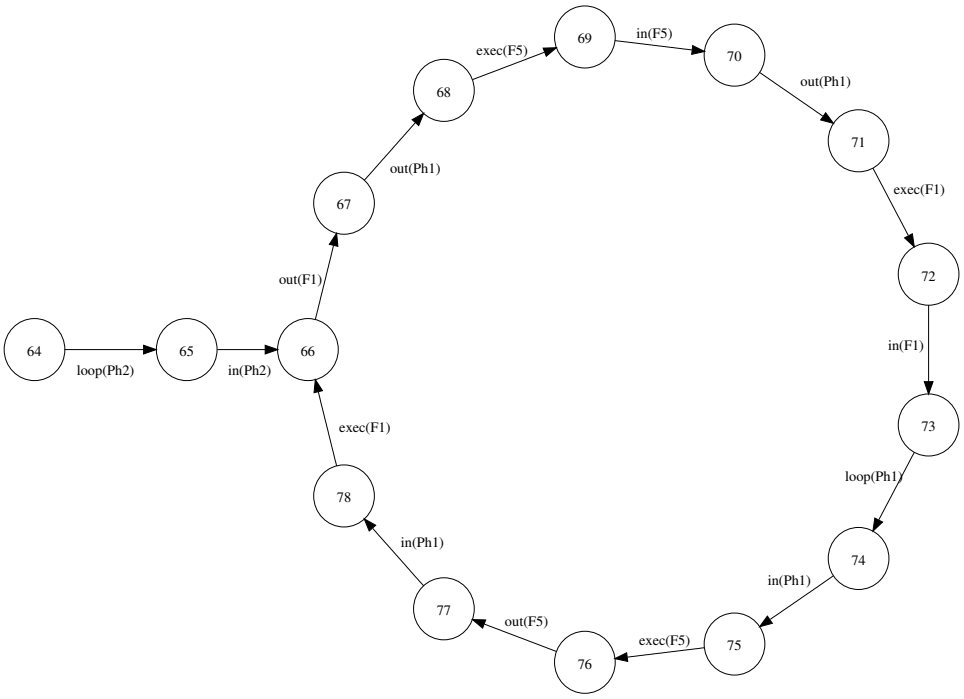

Figure 2: First philosopher eating only – execution path

Szpyrka, M., Matyasik, P., and Mrówka, R. (2011a). Alvis – modelling language for concurrent systems. In Bouvry, P., Gonzalez-Velez, H., and Kołodziej, J., editors, *Intelligent Decision Systems in Large-Scale Distributed Environments*, Studies in Computational Intelligence. Springer-Verlag. (to appear).

Szpyrka, M., Matyasik, P., Mrówka, R., Kotulski, L., and Balicki, K. (2011b). Formal introduction to alvis modelling language. *International Journal of Applied Mathematics and Computer Science*. (to appear).

AUTHOR BIOGRAPHIES

MARCIN SZPYRKA holds a position of associate professor in AGH-UST in Krakow, Poland, Department of Automatics. He has a MSc in Mathematics and PhD and DSc (habilitation) in Computer Science. He is the author of over 70 publications, from the domains of formal methods, software engineering and knowledge engineering. Among other things, he is author of 3 books on Petri nets. His fields of interest also include theory of concurrency and functional programming. He is currently leader of the Alvis project. He also worked out the idea of RTCP-nets (real time coloured Petri nets) for modelling real-time embedded systems. His email is `mszpyrka@agh.edu.pl` and his personal webpage is `http://home.agh.edu.pl/mszpyrka`.

PIOTR MATYASIK holds a position of assistant professor in AGH-UST in Krakow, Poland, Department of Automatics. He has MSc in Automatics and PhD in Computer Science. His interest covers formal methods, robotics, artificial intelligence and programming languages. Currently he is involved in the Alvis project. He is the author of publications on artificial intelligence, formal methods, embedded systems and software engineering. His email is `ptm@agh.edu.pl` and his personal webpage is `http://home.agh.edu.pl/ptm`.

RAFAŁ MRÓWKA holds a position of assistant professor in AGH-UST in Krakow, Poland, Department of Automatics. He has MSc in Automatics and PhD in Computer Science. His interest covers virtual machines, embedded systems, software estimation, formal methods and artificial intelligence. Currently he is a memeber of the Alvis project team. He've published papers on software engineering, formal methods and embedded systems. His email is `Rafal.Mrowka@agh.edu.pl`.