Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1553–1558

Alvis Language with Time Dependence

Marcin Szpyrka, Piotr Matyasik, Michał Wypych
AGH University of Science and Technology
Department of Applied Computer Science
Al. Mickiewicza 30, 30-059 Krakow, Poland
Email: {mszpyrka,ptm,mwypych}@agh.edu.pl

***Abstract*—The paper presents the semantics for the time version of the Alvis modelling language. Alvis combines possibilities of formal models verification with flexibility and simplicity of practical programming languages. The considered time Alvis language is suitable for formal verification of real-time systems. The paper contains description of: the Alvis time model, states and transitions between states and snapshot reachability graphs that represent models state spaces in the form of directed graphs.**

I. Introduction

COSTS of creating and maintaining embedded software draw attention of producers to formal methods. There are more and more attempts to provide methods and tools to improve the concurrent systems development [6], [9]. Alvis combines a formal approach with engineering-like look and style. It is hiding most of the formal side from users but not losing any part of it. Alvis is a modelling language being developed at AGH-UST in Krakow, Department of Applied Computer Science (*http://fm.kis.agh.edu.pl*).

Previous research on Alvis has been mainly concerned with the untimed version of the language with α^0 system layer (multiprocessor environments). The syntax of Alvis which is common for all language versions can be found in [21]. Formal semantics of the untimed version of Alvis has been presented in [22]. This version of Alvis has been successfully used for formal verification of concurrent systems e.g. for BPMN models [23] which may include rule-based systems [18] designed with the XTT2 method [11], [15] or as D-nets [24].

The aim of the paper is to present a draft of semantics for the time version of Alvis with α^0 system layer which allows users to assign to every model statement its duration. Then the set of reachable states of such a model is represented in the form of SR-graph and is used for its verification with model checking techniques [2]. SR-graphs provide the possibility of formal verification of real-time requirements. In contrast to other formalisms like time automata [1], Petri nets with time [10], [17] or multi-agent systems [5], Alvis syntax is very similar to procedural programming languages and the method of model states description is similar to information provided by software debuggers. The idea of SR-graphs has been shortly introduced in [19]. This paper contains formalised and more detailed description of it.

II. Alvis at a Glance

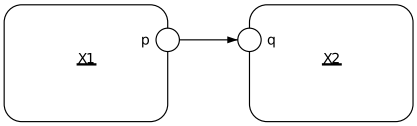
Alvis combines advantages of high level programming languages with a graphical language for modelling interconnections between subsystems (called agents) of a concurrent system. Agents are divided into *active* and *passive*. *Active agents* perform some activities and are similar to tasks in Ada programming language [4]. By contrast, *passive agents* do not perform any individual activities and are similar to protected objects (shared variables). Passive agents provide other agents with a set of procedures (services). An Alvis model is composed of three layers. A *communication diagram* (graphical layer) is used to describe a modelled system from the control and data flow point of view. Examples of such diagrams are given in Fig. 1 and 6. Active agents are drawn as rounded boxes while passive ones as rectangles. Ports used for communication are drawn as circles placed at the edges of the corresponding figures. Alvis agents communicate with each other using communication channels drawn as lines. The code layer is used to define behaviour of agents. It uses a set of Alvis statements and some elements of the Haskell functional programming language [16]. Despite of the fact that Alvis has its origin in the CCS process algebra [14] and the XCCS language [3], [20], it does not use algebraic equations to describe the behaviour of agents but a high level programming language. The *system layer* is predefined and defines the hardware environment for a model. In this paper we consider models with the α^0 *system layer* that denotes that each active agent has access to its own processor and if possible agents perform their steps concurrently. For more details see [21] or the project website.

An Alvis model semantics find expression in a *labelled transition system* (LTS graph). Execution of any language statement is expressed as a transition between formally defined states of such model. An LTS graph is an ordered graph with nodes representing states of the considered system and edges representing transitions among states. Examples of Alvis LTS graphs are given in Fig. 2, 3, 5 and 7. Alvis LTS graphs can be verified using the CADP toolbox [8]. We use CADP *evaluator* tool to check whether the model satisfies requirements given as regular alternation-free μ-calculus formulas [7], [13].

III. Alvis Time Model

The Alvis time model is based on the idea of a *global clock* used to measure the duration of model steps. The language provides carefully selected set of statements sufficient to describe the behaviour of individual agents. Each of them can have duration assigned which is provided by a user as

978-1-4673-4471-5/$25.00




```
agent X1 { loop { out p; } }
agent X2 { loop { in q; } }
```

Figure 1. Communication between active agents

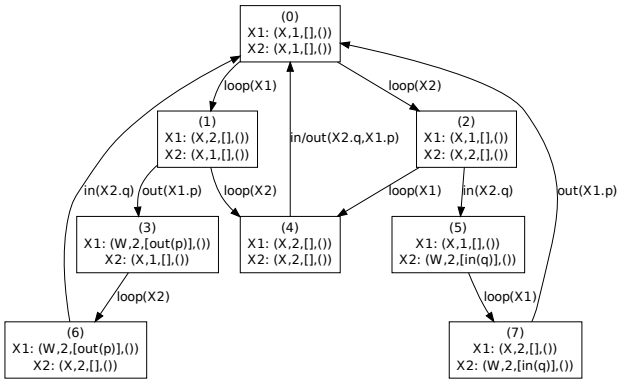


Figure 2. LTS graph for model in Fig. 1

a verification parameter. The time units used in a given model are strictly connected with the model interpretation.

Let us consider the simple model of two communicating active agents shown in Fig. 1. Each agent performs two steps: entering a loop and a communication. Agent $X1$ sequentially sends signals via port $X1.p$ ($X1.p$ denotes port p of agent $X1$), while agent $X2$ sequentially collects signals via port $X2.q$. If an untimed Alvis language is considered, the LTS graph represents all possible execution paths as shown in Fig. 2. The LTS graph labels point out steps performed by agents.

Definition 1: A *state of an agent* X is a tuple

$$S(X) = (am(X), pc(X), ci(X), pv(X)) \tag{1}$$

where $am(X)$, $pc(X)$, $ci(X)$ and $pv(X)$ denote *agent mode*, *program counter*, *context information list* and *parameters values* of the agent X respectively.

The following modes are possible. *Finished* (F) means that an agent has finished its work. *Init* (I) is the default mode for agents that are inactive in the initial state. *Running* (X) means that an agent is performing one of its statements. *Taken* (T) means that one of the passive agent procedures has been called and the agent is executing it. For passive agents *waiting* (W) means that the corresponding agent is inactive and is waiting for another agent to call one of its accessible procedures. For active agents the mode means that the corresponding agent is waiting either for a communication with another active agent or for a currently inaccessible procedure of a passive agent.

The *program counter* points out the current statement of an agent. The context information list contains additional

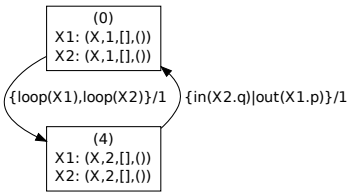


Figure 3. LTS graph for model in Fig. 1 – timed version

information about the current state e.g. if an agent is in the *waiting* mode, ci contains information about events the agent is waiting for. The set of admissible entries used in ci lists is given in Table II. The *parameters values list* contains the current values of the corresponding agent parameters, if such parameters (variables) have been defined in the agent code.

A state of a model is represented as a sequence of agents states [22], [12]. We will use letter S with possible index to denote states. If necessary am, pc, ci, pv will be indicated by indexes S, S' etc. to point out the state they refer to.

If durations of steps are taken under consideration, we cannot consider states of a system in the same way as previously. For example, state 3 in the untimed LTS graph shown in Fig. 2 represents the situation when agent $X1$ has already finished two of its steps, while agent $X2$ still remains in its initial state. Such situation is not possible in time models. Assume steps durations for all steps in the considered model are equal to 1. It means that both agents start execution of their first steps in the same time, so after 1 time-unit the system changes its state from 0 to 4. Finally, the LTS graph for the model is reduced to the one shown in Fig. 3. Labels of edges in the presented graph are of the form $steps/t$, where t stands for the duration of the steps performed simultaneously. The change of the state from 4 to 0 is the result of synchronous communication between agents which is denoted by symbol `|` used instead of a comma.

Alvis uses three statements that use time explicitly:

- **`delay`** `t` – postpones an agent for a given time;
- **`alt`**`(`**`delay`** `t) {...}` – defines a branch of the *select* statement that is open after the given time;
- **`loop`** `(`**`every`** `t) {...}` – repeats loop contents every specified number of time-units.

Let us focus on the *step* idea. It is necessary to distinguish between code statements and steps. Most of the Alvis statements e.g. *exec*, *exit*, etc. are *single-step* statements. By contrast, *if*, *loop* and *select* are *multi-step* statements. We use recursion to count the number of steps for multi-step statements. For each of them, the first step enters the statement interior. Then we count steps of statements put inside curly brackets. From theoretical point of view steps are described as transitions. The formal description of Alvis provides definitions of results of any transition execution. Such formal semantics for untimed models is presented in [22]. The time aspect of transitions is considered in Section IV.

Suppose the code layer for the communication diagram in Fig. 1 is implemented as shown in Fig 4. Agent $X1$ starts its

```
agent X1 {
  loop (every 10) { out p; }}        -- 1, 2

agent X2 {
  loop {                             -- 1
    select {                         -- 2
      alt (ready [in(q)]) {
        in q; delay 1; }             -- 3, 4
      alt (delay 2) { null; }}}}     -- 5
```

Figure 4. Communication between active agents version 2 – new code layer for communication diagram in Fig. 1

Table I
STEP DURATION FOR MODEL IN FIG. 4

Agent $X1$	Step duration	Agent $X2$	Step duration
loop every	1	*loop*	1
out	3	*select*	2
		in	2
		delay	1
		null	1

loop every 10 time-units and sends a signal via port p inside the loop. Behaviour of agent $X2$ is defined as an infinite loop with a **select** statement inside. The statement contains two branches. First one is open (can be performed) if port q can be immediately used to collect a signal (i.e. agent $X1$ has already sent a signal via port p which is connected with q). Inside the branch agent $X2$ collects a signal via port q and is postponed for 1 time-unit. Second branch is open 2 time-units after entering the **select** statement. Inside the branch agent $X2$ performs the empty statement.

Assume steps durations for all steps performed by agents $X1$ and $X2$ are defined as given in Table I. Let us focus on the initial state $S_0 = ((\mathsf{X}, 1, [\,], ()), (\mathsf{X}, 1, [\,], ()))$. When the α^0 system layer and timed Alvis language are considered it is assumed that agents execute their steps as soon as possible. Thus, both agents are running their first steps ($loopevery(X1)$ and $loop(X2)$) concurrently and after one time-unit the state $S_1 = ((\mathsf{X}, 2, [timer(1, 9)], ()), (\mathsf{X}, 2, [\,], ()))$ is received. The $timer(1, 9)$ entry used in $X1$ agent context information list points out that the next loop course can start after 9 time-units. There are two steps $out(X1.p)$ and $select(X2)$ enabled in the state 1. Because step $out(X1.p)$ takes 3 time-units, while $select(X2)$ takes 2 time-units, we cannot present the result of these transitions execution as a state similar to state 1. After 2 time-units the step $out(X1.p)$ is still under execution and after 3 time-units when step $out(X1.p)$ is finished, agent $X2$ could be executing another step – this is not the case in this model due to the lack of an open branch for the **select** statement. The solution for the problem is a *snapshot* [19] i.e. a state that presents the considered system with some steps under execution. We can take a snapshot every 1 time-unit but we are interested only in such snapshots when at least one step has finished its execution.

An LTS graph with snapshots will be called *snapshot reachability graph* or SR-graph for short. A part of the SR-graph for the model in Fig. 4 is shown in Fig. 5. State 2 represents the time point when agent $X2$ has finished step 2 and is waiting for an open branch of the **select** statement, while agent $X1$ is still performing step $out(X1.p)$. The $sft(n)$ (*step finish time*) entry used in $X1$ context information list points out the number of time-units necessary to finish the current step. State 4 represents the time point when agent $X1$ is waiting for a timer event to restart the loop. The event will be generated in 4 time-units. State 5 represents the time point when both agents are waiting for timers' events. Agent $X2$ is waiting for the end of the postpone time. The $time$ label of the edge from state 5 to 6 denotes the passage of time.

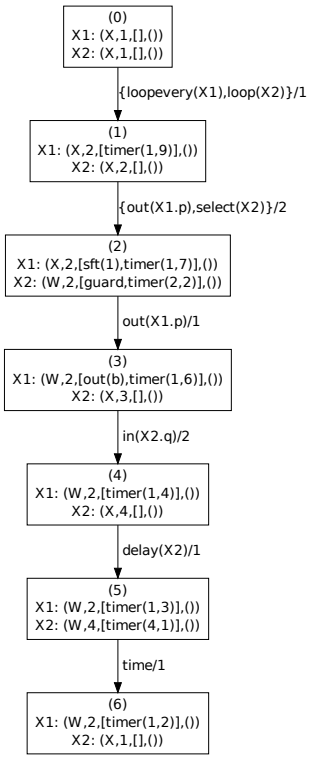


Figure 5. Part of SR-graph for model in Fig. 4

IV. TRANSITIONS

Let $\mathcal{P}$ denote the set of all model ports. For this paper we define Alvis models as follows [22].

Definition 2: A *communication diagram* is a triple $D = (\mathcal{A}, \mathcal{C}, \sigma)$, where: $\mathcal{A} = \{X_1, \ldots, X_n\}$ is the set of *agents* consisting of two disjoint sets, $\mathcal{A}_A$, $\mathcal{A}_P$ such that $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$, containing *active* and *passive* agents respectively; $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ is the *communication relation*, such that: (1) a connection cannot be defined between ports of the same agent; (2) procedure ports are either input or output ones i.e. ports defined as procedures are used to transfer signals (values) either to or from a passive agent; (3) a connection between an active and a passive agent must be a procedure call; (4) a connection between two passive agents must be a procedure call from a non-procedure port. Function $\sigma\colon \mathcal{A}_A \to \{False, True\}$ is the

start function that points out initially activated agents.

Definition 3: An Alvis *model* is a triple $\mathbf{A} = (D, B, \alpha^0)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ is a *communication diagram*, B is a syntactically correct *code layer*, and α^0 is the α^0 *system layer*. Moreover, each agent X belonging to the diagram D must be defined in the code layer and each agent defined in the code layer must belong to the diagram.

Definition 4: A *state* of a model $\mathbf{A} = (D, B, \alpha^0)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ and $\mathcal{A} = \{X_1, \ldots, X_n\}$ is a tuple $S = (S(X_1), \ldots, S(X_n))$. The *initial state* is defined as follows:

- $am(X) = \mathsf{X}$, for any active agent X such that $\sigma(X) = True$; $am(X) = \mathsf{I}$, for any active agent X such that $\sigma(X) = False$; $am(X) = \mathsf{W}$, for any passive agent X;
- $pc(X) = 1$ for any active agent X in the *running* mode and $pc(X) = 0$ for other agents.
- $ci(X) = [\,]$ for any active agent X; and $ci(X)$ contains names of accessible procedures for any passive agent X.
- For any agent X, $pv(X)$ contains X parameters with their initial values.

Table II contains all possible entries that can be included into a context information list and the relationships between the entries and an agent mode.

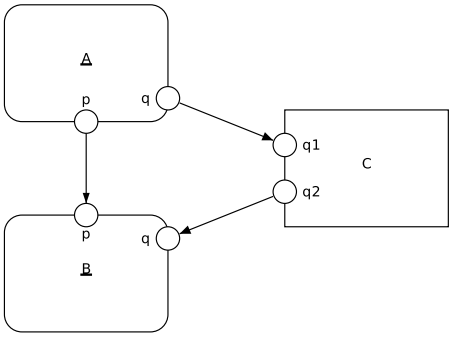
Let $B(X)$ denote an agent X code, $card(B(X))$ denote the number of *steps* in $B(X)$, $B_i(X) \in \{delay, exec, exit, if, in, jump, loop, loopevery, null, out, select, start\}$ denote the name of the agent X i-th step, and $\mathcal{N}(t)$ denote the name of the transition t (possible values are the same as for steps). The set of all transitions available for a particular model will be denoted by $\mathcal{T}$. Moreover, let $\Delta(X, k)$ denote the duration of the k-th step of agent X.

Let us consider the model given in Fig. 6. It contains two active agents A and B that can communicate directly or using passive agent C. Agent A inside the infinite loop performs a **select** statement and waits at most 3 time-units for a communication via port p. In case of a timeout, agent A sends a signal via port q. Agent B inside its periodic loop picks 0 or 1 at random and depending on the result collects a signal via port q or p. Agent C provides two procedures that are accessible depending on the value of parameter n. It works like a buffer for a single signal. The comments included into the code contain steps numbers and durations.

Definition 5: Assume $\mathbf{A} = (D, B, \alpha^0)$ is an Alvis model with the current state S and $X \in \mathcal{A}_A$. A transition $t \in \mathcal{T}$ is *enable* in the state S *with respect to* X if and only if X is in the *running* mode, the program counter points out step t, X has not called a procedure and the step t is not already in progress. The fact that a transition t is enabled in a state S with respect to an agent X and that a state S' is the result of executing t in S will be denoted by $S{-}t(X){\rightarrow}S'$.

The paper [22] contains formal description of all possible transitions for untimed Alvis models. In this section we will focus on description of the differences between untimed and time versions of the language.

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




```
agent A {
  loop {                                        -- 1/1
    select {                                    -- 2/1
      alt (ready [out(p)]) { out p; } -- 3/2
      alt (delay 3) {out q; } } } }    -- 4/2

agent B {
  i :: Int = 0;
  loop (every 6) {                              -- 1/1
    i = pick [0,1];                             -- 2/1
    if(i == 1) { in q; }          -- 3/1, 4/3
    else { in p; } } }                          -- 5/2

agent C {
  n :: Bool = False;
  proc q1 (n == False) {
    in q1; n = True; }            -- 1/2, 2/1
  proc q2 (n == True) {
    out q2; n = False; } }        -- 3/2, 4/1
```

Figure 6. A time model with a passive agent

assign a value w with the corresponding *exec* statement, then for an untimed model the state S' is defined as folows: $S'(X) = (\mathsf{X}, nextpc(S(X)), ci_S(X), pv_S(X)|_{x=w})$, if $nextpc(S(X)) \neq 0$, and $S'(X) = (\mathsf{F}, 0, [\,], pv_S(X)|_{x=w})$ otherwise, where $nextpc$ function determines the next program counter for an agent [22]. Moreover, $S'(Y) = S(Y)$ for any other agent Y.

The transition is defined in a similar way for the time Alvis language. The basic difference concerns ci list with entries referring to time. Let Δ denote the duration of the considered step. Then, in case of $nextpc(S(X)) \neq 0$, we have $S'(X) = (\mathsf{X}, nextpc(S(X)), update(ci_S(X), \Delta), pv_S(X)|_{x=w})$, where the function $update$ replaces entries $timer(s, n)$ with $timer(s, n - d)$ if $n > d$ and with $timeout(s)$ otherwise.

It should be stressed that the $update$ function must be applied to context information lists of all agents in the considered model but it is not enough to determine the new state for the model. If after an ci update the list contains a $timeout(s)$ entry and the agent is in the *waiting* mode in the current state, then the corresponding agent may change its mode (to *running*) and program counter. For example, after execution of the **delay** d statement, agent switches to the *waiting* mode. Then after d time-units (if the statement is not the last one in the main block or a procedural block) the agent switches back to the

Table II
RELATIONSHIPS BETWEEN THE MODE AND THE CONTEXT INFORMATION LIST OF AN AGENT

agent X	$am(X)$	$ci(X)$ entry	description
active passive	X T	$sft(n)$	the current step will be finished in n time-units
active	X, W	$proc(Y.b, a)$	X has called the $Y.b$ procedure via port a and this procedure is being executed in the X agent context
active passive	X, W T	$timer(n, t)$ $timeout(n)$	a time event for the step number n will be generated in t time-units a time event for the step number n has been generated but it has not yet been served
active passive	W T	$in(a)$, $in(a\|T)$ $out(a)$, $out(a\|T)$ $guard$	X waits for a communication via port a (a is the input port for the communication); T is the type of the expected value X waits for a communication via port a (a is the output port for this communication) X waits for an open branch of a *select* statement
passive	T	$proc(Y.b, a)$	X has called the $Y.b$ procedure via port a and this procedure is being executed in the same context as X
passive	W	$in(a)$ $out(a)$	input procedure a is accessible output procedure a is accessible

running mode, its program counter is set to the next value and the $timeout(s)$ entry associated with the considered statement is removed from the ci list. When the $timeout(s)$ entry is *consumed* immediately then it does not appear at all in the agent state in the SR-graph.

The process of a new state determination is complex due to necessity of consideration of all concurrent steps and optimisation of the number of states in the SR-graph. The optimisation refers to skipping snapshots that differ from their predecessors only in parameters of sft and $timer$ entries in the corresponding context information lists. We can distinguish the following stages of a new SR-graph node generation:

- determination of the set $\mathcal{T}_1$ of all transitions that are in progress;
- determination of the set $\mathcal{T}_2$ of all transitions that start performing a new step;
- determination of the new state S' on the assumption that all steps from $\mathcal{T}_1 \cup \mathcal{T}_2$ are performed concurrently.

A state S is called *dead*, iff sets $\mathcal{T}_1$ and $\mathcal{T}_2$ are empty in S and does not exist an agent with ci list containing a $timer$ entry.

Assume all steps have assigned non-zero durations. Firstly, we determine the new state S' as the state 1 time-unit later than S. If state S' differs from S only in parameters of sft and $timer$ entries (parameters are decreased by 1) then we skip that state and calculate the new state 2 time-units later than S, etc. Otherwise, the state S' is a new node in the SR-graph.

If we allow zero duration for at least one step then as additional state *separators* are used changes of agents program counters values. In other words, a label in an SR-graph cannot contain two steps performed by the same agent.

Let us focus on t_{delay} and $t_{loopevery}$ transitions. Suppose, $X \in \mathcal{A}_A$, $S{-}t_{delay}(X){\rightarrow}S'$, $d > 0$ is the argument of the **`delay`** statement and Δ is the duration of the considered step. Then: $S'(X) = (\mathsf{W}, pc_S(X), update(ci_S(X), \Delta) \oplus timer(pc_S(X), d), pv_S(X))$, where $\oplus$ adds the $timer$ entry at the end of the list.

Suppose, $X \in \mathcal{A}_A$, $S{-}t_{loopevery}(X){\rightarrow}S'$ and $d > 0$ is the loop period. Then: $S'(X) = (\mathsf{X}, nextpc(S(X)), update(ci_S(X) \oplus timer(pc_S(X), d), \Delta), pv_S(X))$.

Activity of passive agents is defined similarly as for active ones but a passive agent context (i.e. the active agent that called the procedure in progress) must be taken under consideration [22].

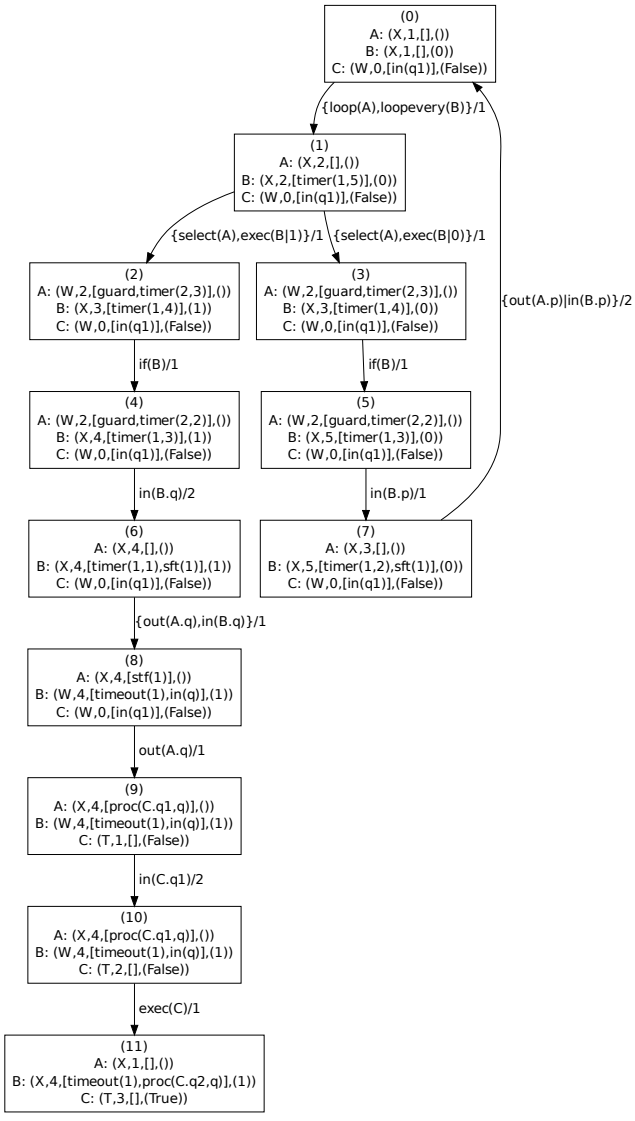


Figure 7. Time model – part of SR-graph

To illustrate presented definitions let us consider a part of the SR-graph for the considered model that is shown in Fig. 7. Let us consider sample states and transitions between them.

- Edge $0 \rightarrow 1$: Steps $loop(A)$ and $loopevery(B)$ are executed simultaneously.
- State 1: The ci list of agent B contains entry $timer(1,5)$ referring to the periodic loop.
- States 2 and 3: The states differ in the value of the parameter of agent B. After execution of **`select`** step, agent A switches to *waiting* mode, because all branches are closed; ci list contains $guard$ and $timer(2,3)$ entries, because A waits either for guard satisfaction or timeout.
- Edge $4 \rightarrow 6$: The duration of $in(B.q)$ step is 3 time-units, but state 6 is present in the SR-graph, because of agent A state change. After lapse of 2 time-units (referring to state 4) entry $timer(2,2)$ was updated to $timeout(2)$ and the agent switched mode to *running* and its program counter was set to 4. At the same time ci list of agent B contains $sft(1)$ entry.
- Edge $5 \rightarrow 7$: In the case of time models readiness of a port for a communication is stated just after commencement (rather than completion) of a communication via this port. After the lapse of 1 time-unit from starting executing $in(B.p)$ step, the condition of the first branch of **`select`** statement is satisfied, so agent A switches to *running* mode and performs steps from the branch.
- Edge $7 \rightarrow 0$: Steps $out(A.p)$ and $in(B.p)$ are performed at the same time as a synchronous communication. In time models communication is considered as synchronous, when intervals of execution of steps in and out overlap partially at least. Such communication is completed when both steps are finished.
- State 8: The $timer(1,1)$ entry in agent B context information list was updated to $timeout(1)$. The periodic loop cannot be restarted because agent B still waits for availability of the called procedure.
- Edge $10 \rightarrow 11$: The execution of agent C $exec$ step finishes procedure $q1$. Because procedure $q2$ has been already called agent C starts it immediately.

V. SUMMARY

The formal description of time Alvis models and the set of transition rules for such models have been considered in the paper. The transition rules provide in fact an algorithm for SR-graphs generation that represent state spaces for such models. It should be stressed that an SR-graph is strictly dependent on the steps duration. For example, if we change the integers presented in Table I we will receive another SR-graph with possibly another paths. An SR-graph enables to check whether a given path (a sequence of steps) is possible to be executed for a given steps durations. We can also determine the minimal and maximal times of passing between two given states, i.e. we can, for example, determine the maximal time of reaction of our system to an event. Moreover, SR-graphs enable us to verify all classic properties like live-locks, deadlocks, process starvation etc. What is more important, the verification of these properties takes time dependencies under consideration. The future work will focus on implementation of algorithms for verification time requirements automatically.

REFERENCES

[1] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

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

[3] K. Balicki and M. Szpyrka, "Formal definition of XCCS modelling language," *Fundamenta Informaticae*, vol. 93, no. 1-3, pp. 1–15, 2009.

[4] J. Barnes, *Programming in Ada 2005*. Addison Wesley, 2006.

[5] A. Byrski and M. Kisiel-Dorohinicki, "Agent-based model and computing environment facilitating the development of distributed computational intelligence systems," in *Computational Science – ICCS 2009*, ser. LNCS, Springer-Verlag, 2009, vol. 5545, pp. 865–874.

[6] A. M. K. Cheng, *Real-time Systems. Scheduling, Analysis, and Verification*. Wiley Interscience, 2002.

[7] E. Emerson, "Model checking and the mu-calculus," in *DIMACS Series in Discrete Mathematics*. Amer. Math. Soc., 1997, pp. 185–214.

[8] 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*, ser. LNCS, vol. 4590. Springer-Verlag, 2007, pp. 158–163.

[9] S. Gnesi and T. Margaria, Eds., *Formal Methods for Industrial Critical Systems. A Survey of Applications*. Hoboken, John Wiley & Sons, 2013.

[10] K. Jensen and L. Kristensen, *Coloured Petri nets. Modelling and Validation of Concurrent Systems*. Springer-Verlag, 2009.

[11] K. Kluza, T. Maślanka, G. Nalepa, and A. Ligęza, "Proposal of representing BPMN diagrams with XTT2-based business rules," in *Intelligent Distributed Computing V – IDC 2011*, ser. Studies in Computational Intelligence, Springer-Verlag, 2011, vol. 382, pp. 243–248.

[12] L. Kotulski, M. Szpyrka, and A. Sędziwy, "Labelled transition system generation from Alvis language," in *Knowledge-Based and Intelligent Information and Engineering Systems – KES 2011*, ser. LNCS, Springer-Verlag, 2011, vol. 6881, pp. 180–189.

[13] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free μ-calculus," INRIA, Tech. Rep. 3899, 2000.

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

[15] G. Nalepa, A. Ligęza, and K. Kaczor, "Formalization and modeling of rules using the XTT2 method," *International Journal on Artificial Intelligence Tools*, vol. 20, no. 6, pp. 1107–1125, 2011.

[16] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. O'Reilly Media, 2008.

[17] M. Szpyrka, "Analysis of VME-Bus communication protocol – RTCP-net approach," *Real-Time Systems*, vol. 35, no. 1, pp. 91–108, 2007.

[18] ——, "Exclusion rule-based systems – case study," in *International Multiconference on Computer Science and Information Technology*, vol. 3, Wisła, Poland, 2008, pp. 237–242.

[19] M. Szpyrka and L. Kotulski, "Snapshot reachability graphs for Alvis models," in *Knowledge-Based and Intelligent Information and Engineering Systems – KES 2011*, ser. LNAI, Springer-Verlag, 2011, vol. 6881, pp. 190–199.

[20] M. Szpyrka and P. Matyasik, "Formal modelling and verification of concurrent systems with XCCS," in *Proc. of the 7th Int. Symposium on Parallel and Distributed Computing (ISPDC 2008)*, Krakow, Poland, 2008, pp. 454–458.

[21] 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. Springer-Verlag, 2011, vol. 362, ch. 15, pp. 315–341.

[22] M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski, "Formal description of Alvis language with α^0 system layer," *Fundamenta Informaticae*, 2013, (to appear).

[23] M. Szpyrka, J. Nalepa, A. Ligęza, and K. Kluza, "Proposal of formal verification of selected BPMN models with Alvis modeling language," in *Intelligent Distributed Computing V – IDC 2011*, ser. Studies in Computational Intelligence, Springer-Verlag, 2011, vol. 382, pp. 249–255.

[24] M. Szpyrka and T. Szmuc, "Decision tables in Petri net models," in *Rough Sets and Intelligent Systems Paradigms*, ser. LNAI, Springer-Verlag, 2007, vol. 4585, pp. 648–657.