Modelling and Verification of Real-Time Systems with Alvis

Marcin Szpyrka, Łukasz Podolski and Michał Wypych

Abstract Alvis is a formal language specifically intended for modelling systems consisting of concurrently operating units. By default, the time dependencies of the modelled system are taken into account, what is expressed by the possibility of determining the duration of each statement performed by the model components. This makes Alvis suitable for modelling and verification of real-time systems. The paper focuses on Alvis time models. The article outlines the syntax and semantics of such models, and discusses the main issues related to the generation of Labelled Transition Systems for time models. Particular attention was paid to tools that support the verification process.

1 Introduction

The development of concurrent systems for which we want to guarantee a high level of reliability can be a tedious and difficult task. High degree of concurrency makes system more flexible but increases the risk of leaving in such a system significant bugs that cannot be detected during the system testing stage. Combination of concurrency and time dependencies makes the task even more difficult. The verification and validation process for such systems cannot be based on classical approaches like peer reviewing and testing [5]. Application of formal methods in the develop-

Marcin Szpyrka
AGH University of Science and Technology, Department of Applied Computer Science, Al. Mickiewicza 30, 30-059 Krakow, Poland e-mail: mszpyrka@agh.edu.pl

Łukasz Podolski
AGH University of Science and Technology, Department of Applied Computer Science, Al. Mickiewicza 30, 30-059 Krakow, Poland e-mail: podolski@agh.edu.pl

Michał Wypych
AGH University of Science and Technology, Department of Applied Computer Science, Al. Mickiewicza 30, 30-059 Krakow, Poland e-mail: mwypych@agh.edu.pl

ment process may significantly reduce the costs and affects the product quality [2]. However, the use of formal methods in the development process requires additional effort in learning new skills and spending more time on analysis and design stages of a software development cycle.

The most popular formal languages that can be used for modelling real-time systems include selected classes of Petri nets [9], [13], [14], time automata [3], and time process algebras [1]. Due to their specific mathematical syntax, these languages are usually treated as the ones suitable only for scientists. In contrast to these languages Alvis [16], [15] is being developed for making the modelling and verification process simpler and more accessible to software developers. The heavy mathematical foundations are hidden from users without compromising the capabilities and expressive power of the formalism. Alvis is equipped with a graphical language [15] for modelling communication channels between the considered system units (called *agents* in Alvis) and a high level programming language for defining agents's behaviour [18]. The language is supported by a set of tools called *Alvis Toolkit*. The software can be used for designing Alvis models, for generating executable Haskell files, for generating Labelled Transition Systems [2] (LTS graphs), and for exporting the LTS graphs to DOT, Aldebaran or CSV formats. This makes possible to verify Alvis models with the most popular model checkers including nuXmv [6] and CADP [7].

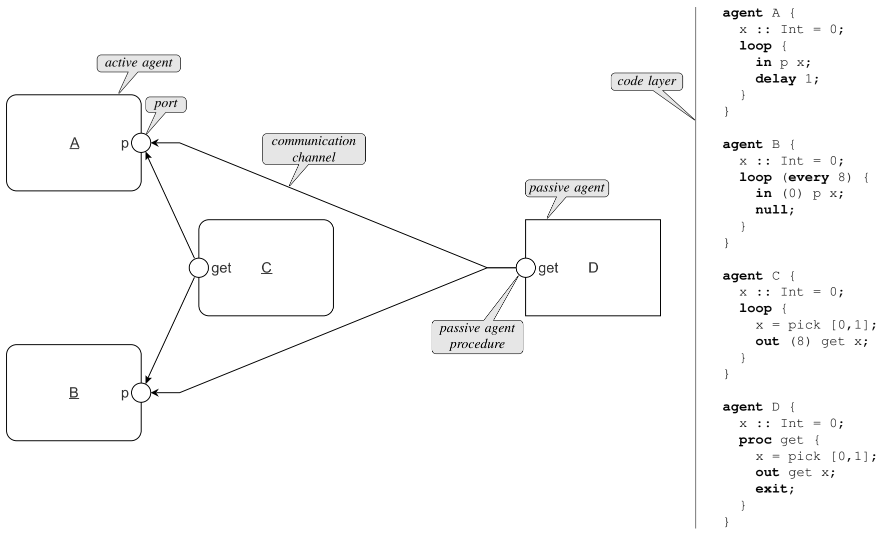
The paper is organised as follows. Section 2 provides a short introduction to the Alvis language. Section 3 deals with semantics of time Alvis models. The concept of LTS graphs for time models is discussed in Sec. 4. Conclusions and future works are presented in the final section.

2 Alvis Language in a Nutshell

An Alvis model is a system of *agents* that usually run concurrently, communicate with each other, compete for shared resources, etc. The set of agents can be divided into two subsets *active agents* and *passive agents*. Active agents can be treated as processes. Current version of the Alvis models (time and non-time models) supports only so-called α^0 system layer. It means that each active agent has access to its own processor and can perform its statements in parallel with other agents. There is also α^1 system layer under development. The later layer is based on the assumption that there is only one processor and all active agents compete for access to the processor.

Passive agents are used to represent shared resources. They provide a set of services for other agents and prevent simultaneous access to the data they store. Passive agents' services do not have their own thread of control but always work in the context of an active agent.

The set of agents of a given model is described with two description layers. The graphical layer is called *communication diagram*. It takes the form of the directed graph with nodes representing agents and edges representing communication channels between ports of agents. Alvis communication diagrams allow users to group a set of agents into a subsystem that is represented as a hierarchical agent at the higher level. Hierarchical communication diagrams are used to simplify modelling of more complex systems, but they do not influence the model semantics. Thus we will consider only flat models in the remainder part of the article. For more information on hierarchical communication diagrams see [15]. The second level contains the Alvis code that defines the behaviour of active and passive agents. The small set of Alvis statements is supported by the Haskell functional programming language [12]. Haskell is used to define parameters, data types and data manipulation functions.



```
agent A {
  x :: Int = 0;
  loop {
    in p x;
    delay 1;
  }
}

agent B {
  x :: Int = 0;
  loop (every 8) {
    in (0) p x;
    null;
  }
}

agent C {
  x :: Int = 0;
  loop {
    x = pick [0,1];
    out (8) get x;
  }
}

agent D {
  x :: Int = 0;
  proc get {
    x = pick [0,1];
    out get x;
    exit;
  }
}
```

Fig. 1 Example of Alvis model

An example of Alvis model is shown in Fig. 1. The model is composed of three active agents and one passive agent. Agents *A* and *B* compete for data provided by agent *C* and for access to the agent *D* procedure. The behaviour of these agents was defined for illustrating typical Alvis statements suitable for modelling real-time systems.

- Agent *A* – The agent collects an integer via port *p*. It uses the blocking communication [11] i.e. after initialisation of a communication it waits until agent *C* provides the value or procedure *D.get* is accessible. After collecting an integer the agent is postponed for 1 time-unit. This behaviour is repeated inside the infinite loop.
- Agent *B* – The agent uses the *loop every* statement. It means that the contents of the loop is repeated every 8 time-units. The agent uses the non-blocking communication with argument 0. It means that the statement finalises a communication with agent *C* or calls *D.get* procedure that must be accessible. If it is not possible, the communication is abandoned. The *null* statement is necessary at the end of a *loop every* statement.

- Agent *C* – The agent randomly selects a value from the given list, assigns it to parameter *x*, and sends it via port *get*. If the agent initialises a communication it waits at most 8 time-units for finalisation. Otherwise, the communication is abandoned. This behaviour is repeated inside the infinite loop.
- Agent *D* – The passive agent is equipped with one procedure that provides a randomly selected value from the given list.

Both the communication diagram and the code layer can be developed using *Alvis Editor* software. The semantics of models is presented in the next section. For more details about the Alvis syntax see the project website.[1]

3 Model Semantics

A *state of an agent X* is a tuple $S(X) = (am(X), pc(X), ci(X), pv(X))$, 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. A *state of an Alvis model* is a sequence of such four-tuples as shown in Fig. 2 [15].

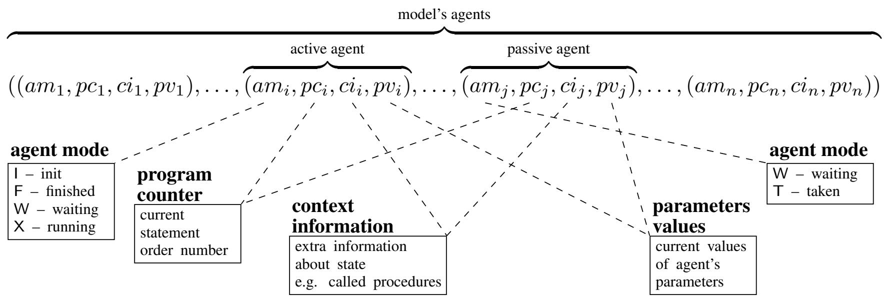

model's agents
active agent
passive agent
$((am_1, pc_1, ci_1, pv_1), \ldots, (am_i, pc_i, ci_i, pv_i), \ldots, (am_j, pc_j, ci_j, pv_j), \ldots, (am_n, pc_n, ci_n, pv_n))$
agent mode
I – init
F – finished
W – waiting
X – running
program counter
current statement order number
context information
extra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

Fig. 2 Representation of an Alvis model state

An active agent can be in one of the following modes:

- *Finished* (F) – It means that the agent has finished its work.
- *Init* (I) – This is the default mode for agents that are inactive in the initial state.
- *Running* (X) – It means that the agent is performing one of its statements.
- *Waiting* (W) – It means that the agent is waiting for an event e.g. releasing a currently inaccessible procedure

A passive agent is in the *waiting* mode if it is inactive and waits for another agent to call one of its accessible procedures or in the *taken* (T) mode if it is executing one of its procedures.

[1] `http://alvis.kis.agh.edu.pl`

The program counter points out the current statement of the corresponding agent. The context information list contains additional information about the current state of the corresponding agent e.g. if the agent is in the *waiting* mode, *ci* contains information about events the agent is waiting for. In case of passive agents in the *waiting* mode, the context information contains list of accessible procedures. The *parameters values tuple* contains the current values of the agent parameters. The initial state for the model from Fig. 1 is as follows:

$$((\mathsf{X},1,[\,],0),(\mathsf{X},1,[\,],0),(\mathsf{X},1,[\,],0),(\mathsf{W},0,[out(get)],0)$$

Table 1 Alvis transitions (α^0 system layer)

Transition	**Arguments**	**Description**
TDelay	*Agent Int*	*delay* statement execution
TExec	*Agent Int*	*exec* statement execution
TExit	*Agent Int*	*exit* statement execution
TIn	*Port Int*	initialisation of communication, *in* statement
TInAP	*Port Port Int*	calling procedure by active agent, *out* statement
TInPP	*Port Port Int*	calling procedure by passive agent, *out* statement
TInF	*Port Port Int*	finalisation of communication with active agent, *in* statement
TJump	*Agent Int*	*jump* statement execution
TLoop	*Agent Int*	entering *loop* statement
TLoopEvery	*Agent Int*	entering *loop every* statement
TNull	*Agent Int*	*null* statement execution
TOut	*Port Int*	initialisation of communication, *out* statement
TOutAP	*Port Port Int*	calling procedure by active agent, *out* statement
TOutPP	*Port Port Int*	calling procedure by passive agent, *out* statement
TOutF	*Port Port Int*	finalisation of communication with active agent, *out* statement
TSelect	*Agent Int*	entering *select* statement
TStart	*Agent Int*	*start* statement execution
STInAP	*Port Port Int*	system version of *TInAP* – for wake up purposes
STInPP	*Port Port Int*	system version of *TInPP* – for wake up purposes
STOutAP	*Port Port Int*	system version of *TOutAP* – for wake up purposes
STOutPP	*Port Port Int*	system version of *TOutPP* – for wake up purposes
STDelayEnd	*Agent Int*	termination of agent's suspension
STLoopEnd	*Agent Int*	termination of current *loop every* run
STInEnd	*Port Int*	abandonment of communication, non-blocking *in* statement
STOutEnd	*Port Int*	abandonment of communication, non-blocking *out* statement
STTime	*Int*	passage of time.

Execution of Alvis statements is described using the *transition* idea. For example execution of the *loop* statement i.e. entering a loop if its guard is true (or there is no guard) is represented by the *TLoop* transition. The activity of a transition is always considered for a given agent and statement. The set of Alvis transitions for models with α^0 system layer is given in Tab. 1.

The transitions *TInAP, TInPP TInF* and *TIn* represent the *in* statement. The use of four transitions for single statement is associated with the variety of situations in which the statement can be used. Let us focus on the model from Fig. 1. When agent *A* executes the statement `in p x;` it means that *A* wants to collect a value via port *p* and assign it to parameter *x*. The agents does not know whether the value will be provided by the active agent *C* or the passive agent *D*. On the other hand, from the *Alvis Compiler* point of view these situations must be distinguished and they are represented by different transitions. Similarly, four transitions are used to represent the *out* statement. For more information about different Alvis communication modes see [11].

Despite the set of transitions that directly represent execution of some statements, names of these transitions start with capital T, there are transitions that represent some activities of the model *runtime environment* (so-called *system transitions*) – names of these transitions start with capital S. These transitions represent waking up of an agent that, from some reasons, is in the *waiting* mode. There is one exception, the *STTime* transition, that represents the passage of time. It is used when there are no transitions available in the current moment (e.g. all active agents are in the *waiting* mode) but at least one of them will be enabled in some future moment. It is used to shift the value of the global clock.

Table 1 contains the list of all Alvis transitions for time α^0 models. The set of transitions used in a given model depends on the statements used in the code layer. For example the following transitions can be enabled for agent *A* from the considered example: *TLoop A* 1, *TInF A.p C.get* 2, *TInAP A.p D.get* 2, *TIn A.p* 2, *STInAP A.p D.get* 2, *TDelay A* 3, *STDelayEnd A* 3. It should be stressed that we have four different transitions for the second statement.

3.1 Enable Rules

For each of the transitions from Tab. 1 we can define *enable* and *firing* rules. Enable rules define conditions when the given transition is enabled. The firing rules define how the given transition influences on the change of the current model state. For active agents and transitions: *TDelay*, *TExec*, *TExit*, *TJump*, *TLoop*, *TLoopEvery*, *TNull*, *TSelect*, and *TStart* the given transition is *enable* iff the agent is in the *running* mode and the corresponding statement is the current statement, what is indicated by the agent program counter. In case of communication transitions additional conditions must be fulfilled:

- *TInAP* – There exists an accessible procedure connected with the considered port of the active agent and none procedure is executed for the agent currently.
- *TInF* – There exists an active agent that already initialised a communication (*out* statement) via a port connected with the considered port of the active agent, that executes the transition, and none procedure is executed for the agent currently.
- *TIn* – Transitions *TInAP*, *TInF* are not enabled and no procedure is executed for the agent currently – If it is not possible to call a procedure or finalise a communication, the transition initialises a communication and moves the agent

to the *waiting* mode. If it waits for a currently inaccessible procedure then the *STInAP* transition wakes up the agent when the procedure is accessible.

Enable conditions for *TOut*, *TOutAP*, *TOutF*, and *STOutAP* transitions are defined analogously. In case of passive agents the conditions are similar but the given passive agent must be in the *taken* mode and the active agent in which context it works must be in the *running* mode. For passive agents, transitions *TInPP*, *TOutPP*, *STInPP*, and *STOutPP* are used instead of corresponding ... *AP* transitions.

The system transitions *STDelayEnd*, *STInEnd*, *STOutEnd* are enabled if the agent is in the *waiting* mode after executing the corresponding statements (in case of passive agent the agent is in the *taken* mode and the context agent is in the *waiting* mode) and the waiting time has elapsed. Finally, the *STLoopEnd* transition is enabled if the agent has finished executing the contents of the corresponding *loop every* statement and the period of the loop has expired.

3.2 Firing Rules

As it is presented in Sec. 4, in case of time models a few transitions can be executed in parallel. However, to describe the *firing rules* we consider the results of executing of individual transitions in a given state s. Let $\mathit{nextpc}(n)$ denote the next program counter determined on the basis of the code structure for the considered agent and the current program counter n. For example, if we consider a *TLoop* transition then $\mathit{nextpc}(n)$ is equal to the number of the first statement inside the loop if the guard is satisfied (or there is no guard) or the number of the first statement after the loop otherwise. It is assumed that $\mathit{nextpc}(n) = 0$ if there is no next statement.

Assume we consider the result of a transition firing for agent X, n denotes the number of the statement the considered transition refers to, and $\mathit{context}(X)$ denotes the active agent in which context X works if X is a passive agent. Firing of the *TExec*, *TJump*, *TLoop*, *TLoopEvery*, *TNull*, *TSelect* or *TStart* transition sets $pc(X) = \mathit{nextpc}(n)$. Moreover, the *TExec* transition updates the value of the parameter used as the left-hand side of the assign operator; *TLoopEvery* transition adds $\mathit{timer}(n,d)$ entry to $ci(X)$, where d represents the number of time-units to the end of the current loop run; and the *TStart* transition set its argument (agent) to the *running* mode and its program counter to 1 if the agent is in the *init* mode. If X is an active agent and $\mathit{nextpc}(n) = 0$ then any of these transitions (except *TJump* and *TLoopEvery*) sets $am(X) = \mathsf{F}$ and $ci(X) = [\,]$. The *null* statement is also used to point out the end of the contents of a *loop every* statement. In such a case, the corresponding *TNull* transition sets $am(X) = \mathsf{W}$ (or $am(\mathit{context}(X)) = \mathsf{W}$ if X is a passive agent), and $pc(X)$ to the number of the corresponding *loop every* statement.

Firing of the *TDelay* transition sets $am(X) = \mathsf{W}$ (or $am(\mathit{context}(X)) = \mathsf{W}$ if X is a passive agent) and adds $\mathit{timer}(n,d)$ entry to $ci(X)$, where d represents the number of time-units of the suspension.

If X is an active agent then firing of the *TExit* transition sets $pc(X) = 0$, $am(X) = \mathsf{F}$, and $ci(X) = [\,]$. If X is a passive agent then firing of the *TExit* transition ends the current procedure (let us denote it by $X.p$) i.e. sets $am(X) = \mathsf{W}$, $pc(X) = 0$, and $ci(X)$ to the set of X procedures accessible in the new state. Moreover, if the procedure has been called by an agent Y then the $proc(X.p)$ entry is removed from $ci(Y)$ and $pc(Y)$ is set to its next value (if it is 0 and Y is an active agent then also $am(Y) = \mathsf{F}$, and $ci(Y) = [\,]$).

Firing of the transition *TInAP* $X.p$ $Y.q$ n (or *TInPP*, *TOutAP*, *TOutPP* with the same arguments) inserts $proc(Y.q)$ entry to $ci(X)$ and sets $am(Y) = \mathsf{T}$, $ci(Y) = [\,]$, and $pc(Y)$ to the number of the first statement in $Y.q$ procedure. System transitions *STInAP* and *STInPP* additionally changes $am(X)$ ($am(context(X))$ if X is a passive agent) from W to X and removes $in(p)$, $timer(n,d)$ entries from $ci(X)$ (the $timer(n,d)$ entry is used only for non-blocking communication). System transitions *STOutAP* and *STOutPP* work similarly but it removes $out(p)$ entry instead of $in(p)$. The result of the *TIn* $X.p$ n transition firing depends on the type of communication:

1. *Non-blocking with time* $d = 0$: sets $pc(X) = nextpc(n)$ (if it is 0 and X is an active agent then also $am(X) = \mathsf{F}$, and $ci(X) = [\,]$).
2. *Non-blocking with time* $d > 0$: sets $am(X) = \mathsf{W}$ ($am(context(X)) = \mathsf{X}$ if X is a passive agent), and inserts $in(p)$, $timer(n,d)$ entries into $ci(X)$.
3. *Blocking, X is an active agent*: sets $am(X) = \mathsf{W}$ and inserts $in(p)$ entry into $ci(X)$.
4. *Blocking, X is a passive agent, X.p is non-procedure port*: sets $am(context(X)) = \mathsf{W}$ and inserts $in(p)$ entry into $ci(X)$.
5. *Blocking, X is passive agent, X.p is procedure port*: sets $pc(X) = nextpc(n)$, and updates value of the corresponding parameter (if a value has been sent).

The *TOut* transition works similarly but $out(p)$ entry is inserted instead of $in(p)$ and in case of a procedure port there is no parameter update.

Firing of the *TInF* $X.p$ $Y.q$ n transition updates value of the corresponding parameter of agent X (if a value has been sent), sets $pc(X) = nextpc(n)$, $am(Y) = \mathsf{X}$, $pc(Y) = nextpc(m)$ (where m is the current value of $pc(Y)$), and removes $out(q)$ and $timer(n,d)$ entries from $ci(Y)$. If $nextpc(n) = 0$ or $nextpc(m) = 0$ then for the corresponding agent the mode is set to F and the context list to $[\,]$. The *TOutF* transition works similarly but $in(q)$ entry is removed instead of $out(q)$ and a parameter of Y agent is potentially updated.

Firing of the *STLoopEnd* transition sets $am(X) = \mathsf{X}$ ($am(context(X)) = \mathsf{X}$ if X is a passive agent), and removes $timeout(n)$ entry from $ci(X)$. Firing of a *STDelayEnd* transition additionally sets $pc(X) = nextpc(n)$ (if it is 0 and X is an active agent then also $am(X) = \mathsf{F}$, and $ci(X) = [\,]$).

Firing of the *STInEnd* transition sets $am(X) = \mathsf{X}$ ($am(context(X)) = \mathsf{X}$ if X is a passive agent), removes $in(p)$ and $timeout(n)$ entries from $ci(X)$, and sets $pc(X) = nextpc(n)$ (if it is 0 and X is an active agent then also $am(X) = \mathsf{F}$, and $ci(X) = [\,]$). A *STOutEnd* works similarly but it removes $out(p)$ entry instead of $in(p)$.

The presented *enable* and *firing* rules show how the firing of a transition affects the changes of states. More precise definitions of the rules but in the context of the Intermediate Haskell Representation (IHR) of Alvis models are presented in the language manual [18]. The next section describes how a new state is determined if we take into account a set of transitions executed in parallel and the passage of time.

4 LTS Graphs for Time Models

To verify an Alvis model's properties using model checking techniques [2] it is necessary to generate the model state-space first. We use Labelled Transition Systems (LTS graphs) to represent such state-spaces. Nodes of an LTS graph represent reachable model states. Labels of arcs provide two pieces of information: a set of transitions that are executed in parallel and lead from the corresponding arc source state to the arc destination state and the time that elapsed between these two consecutive states. The initial part of the LTS graph for the model from Fig. 1 is shown in Fig. 3. This section focuses on the most important parts of the LTS generation algorithm implemented in Alvis Toolkit.

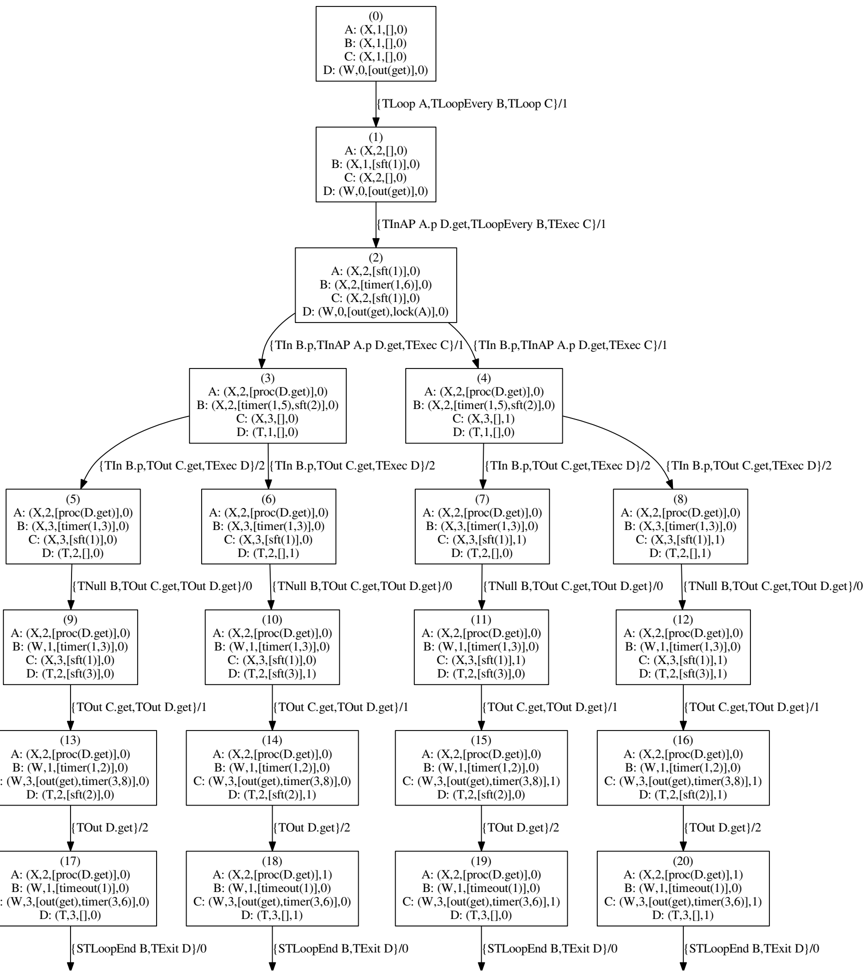

(0) A: (X,1,[],0) B: (X,1,[],0) C: (X,1,[],0) D: (W,0,[out(get)],0)
{TLoop A,TLoopEvery B,TLoop C}/1
(1) A: (X,2,[],0) B: (X,1,[sft(1)],0) C: (X,2,[],0) D: (W,0,[out(get)],0)
{TInAP A.p D.get,TLoopEvery B,TExec C}/1
(2) A: (X,2,[sft(1)],0) B: (X,2,[timer(1,6)],0) C: (X,2,[sft(1)],0) D: (W,0,[out(get),lock(A)],0)
{TIn B.p,TInAP A.p D.get,TExec C}/1
{TIn B.p,TInAP A.p D.get,TExec C}/1
(3) A: (X,2,[proc(D.get)],0) B: (X,2,[timer(1,5),sft(2)],0) C: (X,3,[],0) D: (T,1,[],0)
(4) A: (X,2,[proc(D.get)],0) B: (X,2,[timer(1,5),sft(2)],0) C: (X,3,[],1) D: (T,1,[],0)
{TIn B.p,TOut C.get,TExec D}/2
{TIn B.p,TOut C.get,TExec D}/2
{TIn B.p,TOut C.get,TExec D}/2
{TIn B.p,TOut C.get,TExec D}/2
(5) A: (X,2,[proc(D.get)],0) B: (X,3,[timer(1,3)],0) C: (X,3,[sft(1)],0) D: (T,2,[],0)
(6) A: (X,2,[proc(D.get)],0) B: (X,3,[timer(1,3)],0) C: (X,3,[sft(1)],0) D: (T,2,[],1)
(7) A: (X,2,[proc(D.get)],0) B: (X,3,[timer(1,3)],0) C: (X,3,[sft(1)],1) D: (T,2,[],0)
(8) A: (X,2,[proc(D.get)],0) B: (X,3,[timer(1,3)],0) C: (X,3,[sft(1)],1) D: (T,2,[],1)
{TNull B,TOut C.get,TOut D.get}/0
{TNull B,TOut C.get,TOut D.get}/0
{TNull B,TOut C.get,TOut D.get}/0
{TNull B,TOut C.get,TOut D.get}/0
(9) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,3)],0) C: (X,3,[sft(1)],0) D: (T,2,[sft(3)],0)
(10) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,3)],0) C: (X,3,[sft(1)],0) D: (T,2,[sft(3)],1)
(11) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,3)],0) C: (X,3,[sft(1)],1) D: (T,2,[sft(3)],0)
(12) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,3)],0) C: (X,3,[sft(1)],1) D: (T,2,[sft(3)],1)
{TOut C.get,TOut D.get}/1
{TOut C.get,TOut D.get}/1
{TOut C.get,TOut D.get}/1
{TOut C.get,TOut D.get}/1
(13) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,2)],0) C: (W,3,[out(get),timer(3,8)],0) D: (T,2,[sft(2)],0)
(14) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,2)],0) C: (W,3,[out(get),timer(3,8)],0) D: (T,2,[sft(2)],1)
(15) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,2)],0) C: (W,3,[out(get),timer(3,8)],1) D: (T,2,[sft(2)],0)
(16) A: (X,2,[proc(D.get)],0) B: (W,1,[timer(1,2)],0) C: (W,3,[out(get),timer(3,8)],1) D: (T,2,[sft(2)],1)
{TOut D.get}/2
{TOut D.get}/2
{TOut D.get}/2
{TOut D.get}/2
(17) A: (X,2,[proc(D.get)],0) B: (W,1,[timeout(1)],0) C: (W,3,[out(get),timer(3,6)],0) D: (T,3,[],0)
(18) A: (X,2,[proc(D.get)],1) B: (W,1,[timeout(1)],0) C: (W,3,[out(get),timer(3,6)],0) D: (T,3,[],1)
(19) A: (X,2,[proc(D.get)],0) B: (W,1,[timeout(1)],0) C: (W,3,[out(get),timer(3,6)],1) D: (T,3,[],0)
(20) A: (X,2,[proc(D.get)],1) B: (W,1,[timeout(1)],0) C: (W,3,[out(get),timer(3,6)],1) D: (T,3,[],1)
{STLoopEnd B,TExit D}/0
{STLoopEnd B,TExit D}/0
{STLoopEnd B,TExit D}/0
{STLoopEnd B,TExit D}/0

Fig. 3 Initial part of the LTS graph for the model from Fig. 1

The generation of an LTS graph for time models starts with the initial state that has the serial number 0 and is the initial node of the LTS graph. The initial state is the first *current* node. For any current node the set of outgoing arcs and finally direct successors are generated. Let *s* denote the model state represented by the current node. We start with generation of the set of transitions enabled in state *s* according to the enable rules presented in Sec. 3. For example the set of transitions enabled in the initial state of the considered model contains three elements: *TLoop A* 1, *TLoopEvery B* 1, *TLoop C* 1. If such a set does not contain any communication transition then all the transitions can be executed in parallel as shown in Fig. 3

By default duration of each statement (transition) is equal to 1 time-unit. One can define the individual value of duration for each model statement. This is done using the *duration* function implemented in Haskell. In case of the considered model the function is defined as follows:

```
duration :: Agent -> Int -> Int
duration A 1 = 1
duration A 2 = 2
duration A 3 = 1
duration B 1 = 2
duration B 2 = 3
duration B 3 = 0
duration C 1 = 1
duration C 2 = 2
duration C 3 = 3
duration D 1 = 2
duration D 2 = 3
duration D 3 = 1
duration _ _ = 1
```

This means that the three enabled transitions have different durations (1, 2, and 1 respectively) and we cannot move from the initial state directly to a state where all the transitions are finished. In this case, we can move only 1 time-unit forward. Thus, the new state describes a situation when one of the transitions is still under execution (see state 1, Fig. 3). The $sft(n)$ (*step finish time*) entry used in $ci(B)$ points out the number of time-units necessary to finish the current transition.

The set of transitions executed in parallel is called *multi-step*. The LTS graph generation algorithm determines the maximal *time shift* for each multi-step. This value is selected so as not to lose any information about the changes of states of the analysed system. The algorithm takes into account not only the duration of each transition in the multi-step but also the arguments of context entries such as *sft* or *timer*.

The result of a multi-step execution consists not only of the effects of single transitions execution but also the results of the time shift i.e. the arguments of all timers and *sft* entries are reduced by the value of the time shift, even if an agent does not execute a transition in the given multi-step. Moreover, if the time argument of a timer is reduced to 0, then the entry is replaced with *timeout* one (e.g. see state 17 Fig. 3).

If the set of enabled transitions contains communication transitions then *conflicts* may arise e.g. two transitions are enabled but they cannot be executed in parallel. This is the case with the state 437:

$$(\mathsf{X},2,[\,],0),(\mathsf{X},2,[timer(1,6)],0),(\mathsf{X},3,[sft(1)],0),(\mathsf{W},0,[out(get)],0))$$

The set of enabled transitions contains the following elements: *TInAP A.p D.get* 2, *TInAP B.p D.get* 2, and *TOut C.get* 3. Thus, we have two agents *A* and *B* that compete for the access to the same passive agent. Of course these three transitions cannot be executed in parallel. The set of enabled transitions must be divided into two multi-steps.

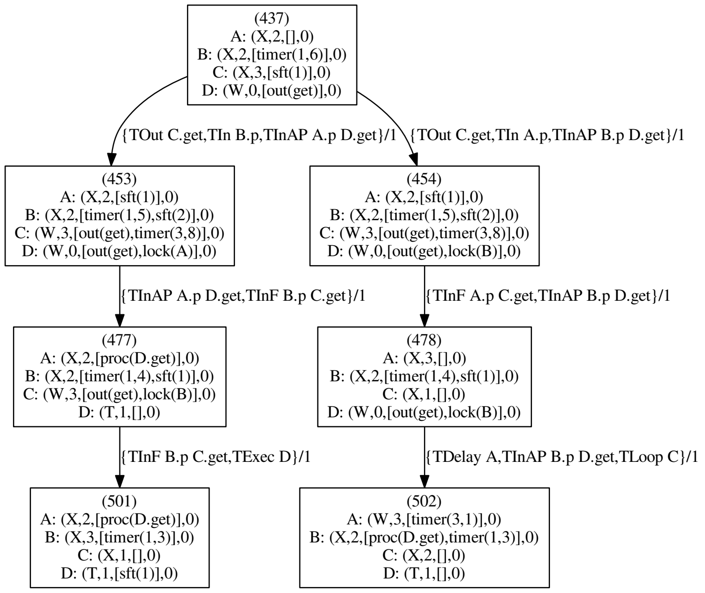

(437)
A: (X,2,[],0)
B: (X,2,[timer(1,6)],0)
C: (X,3,[sft(1)],0)
D: (W,0,[out(get)],0)
{TOut C.get,TIn B.p,TInAP A.p D.get}/1
{TOut C.get,TIn A.p,TInAP B.p D.get}/1
(453)
A: (X,2,[sft(1)],0)
B: (X,2,[timer(1,5),sft(2)],0)
C: (W,3,[out(get),timer(3,8)],0)
D: (W,0,[out(get),lock(A)],0)
(454)
A: (X,2,[sft(1)],0)
B: (X,2,[timer(1,5),sft(2)],0)
C: (W,3,[out(get),timer(3,8)],0)
D: (W,0,[out(get),lock(B)],0)
{TInAP A.p D.get,TInF B.p C.get}/1
{TInF A.p C.get,TInAP B.p D.get}/1
(477)
A: (X,2,[proc(D.get)],0)
B: (X,2,[timer(1,4),sft(1)],0)
C: (W,3,[out(get),lock(B)],0)
D: (T,1,[],0)
(478)
A: (X,3,[],0)
B: (X,2,[timer(1,4),sft(1)],0)
C: (X,1,[],0)
D: (W,0,[out(get),lock(B)],0)
{TInF B.p C.get,TExec D}/1
{TDelay A,TInAP B.p D.get,TLoop C}/1
(501)
A: (X,2,[proc(D.get)],0)
B: (X,3,[timer(1,3)],0)
C: (X,1,[],0)
D: (T,1,[sft(1)],0)
(502)
A: (W,3,[timer(3,1)],0)
B: (X,2,[proc(D.get),timer(1,3)],0)
C: (X,2,[],0)
D: (T,1,[],0)

Fig. 4 Part of the LTS graph for the model from Fig. 1

The transitions *TInAP A.p D.get* 2 and *TInAP B.p D.get* 2 cannot be placed in the same multi-step, but both agents *A* and *B* can execute a transition in each of this two multi-steps. If we chose the *TInAP A.p D.get* 2 transition then the port *D.get* is already inaccessible for agent *B*, but the agent can execute the *TIn B.p* 2 transition. Thus both multi-sets contain three transitions as shown in Fig. 4 (see labels for node 437 outgoing arcs). States 453, 454 and 478 illustrate a situation when agent *D* is still in the *waiting* mode, because the corresponding *TInAP* is not finished yet, but it is not accessible for other agents, because the *TInAP* is already started. This is indicated by the *lock* entry included into the context information list.

Stages like determining the list of enabled transitions, dividing the set into multi-steps, determining time shift for each multi-step are performed for each reachable state. The final LTS graph contains a node for each reachable state and an arc for each executed multi-step. The LTS graph is generated automatically. The Alvis language is supported by computer tools called *Alvis Toolkit*. Models can be designed with *Alvis Editor* that provides essential editing features, such as: diagram edition, basic tools for alignment and colouring, automatic creation and removal (flattening) of hierarchical pages [15], textual layer addition with syntax colouring and code folding. An Alvis model stored in an XML file is then processed by *Alvis Compiler* that generates the IHR for the model. The generated Haskell file can be modified by the user. For example user-defined verification algorithms, a priority management algorithm, user-defined *duration* function (assignment of duration to each model statement), user-defined *main* function, etc. can be included into the file before compilation. Finally, depending on the used *Alvis Compiler* options and user's optional code modifications, as a result of the Haskell program execution, the LTS graph in various textual represtation that can be directly passed into standard model checkers like e.g. *CADP*, simulation logs and/or result of user-defined verification procedures are provided.

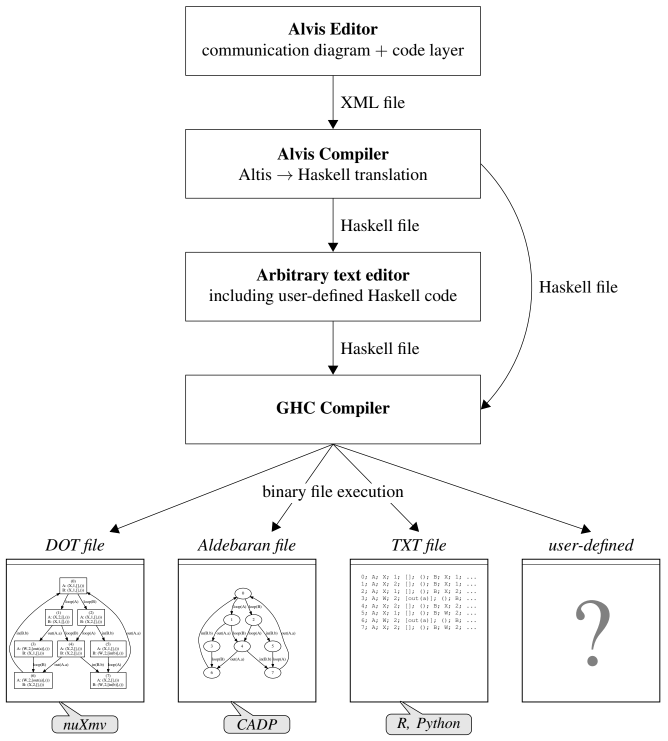
5 Conclusions and Future Work

The most important features of Alvis time models that are essential for modelling real-time systems have been presented in the paper. Both the Alvis language and the Alvis Toolkit were developed for providing comfortable and flexible formal tools for engineers. The language provides statements for modelling of phenomena typical for real-time systems modelling like: concurrent executing of processes, synchronisation of processes, periodic processes, priorities of processes, relative delays, timeouts, etc.

The verification methods for Alvis models are mainly based on the LTS graph, generated for the given model automatically, and model checking techniques [2]. Users can choose the preferred output format that the generated program will deliver, compiler expose *-dot*, *-ald* and *-csv* options for *Graphviz DOT*, *CADP Aldebaran* and popular *CSV* formats respectively. Thus, Alvis models can be verified using popular model checkers like nuXmv [6], [4] and CADP [7] and languages for statistical data analysis like Python [8] and R [10]. Possible processing paths of Alvis models are shown in Fig. 5.

It is worth emphasizing yet another advantage of the IHR use. The Alvis *exec* statement is represented by the assignment operator (=) that takes an agent's parameter as its left-hand side argument and a Haskell expression as the right-hand side argument. The statement evaluates the expression and assigns its result to the parameter. This is represented as a single step in the LTS graph, but the Haskell expression may contain any user-defined function and may represent complex operations on some data. It is an easy way to include, for example, artificial intelligence systems like rule-based systems, decision trees, neural networks, etc. into Alvis models. An example of including a decision support system into an Alvis model of a railway traffic management system is presented in [17].

The presented version of Alvis time models supports only so-called α^0 system layer. There is also α^1 system layer under development. This layer is based on the assumption that there is only one processor and all active agents compete for access to it. This will allow Alvis to be used for modelling of software for single-processor embedded systems.



```
0; A; X; 1; []; (); B; X; 1; ...
1; A; X; 2; []; (); B; X; 1; ...
2; A; X; 1; []; (); B; X; 2; ...
3; A; W; 2; [out(a)]; (); B; ...
4; A; X; 2; []; (); B; X; 2; ...
5; A; X; 1; []; (); B; W; 2; ...
6; A; W; 2; [out(a)]; (); B; ...
7; A; X; 2; []; (); B; W; 2; ...
```

Fig. 5 Possible processing paths of Alvis models

References

1. Aceto, L., Ingófsdóttir, A., Larsen, K., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge, UK (2007)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, London, UK (2008)
3. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. Lecture Notes on Concurrency and Petri Nets **3098** (2004)
4. Biernacki, J.: Alvis models of safety critical systems state-base verification with nuXmv. In: Proceedings of the Federated Conference on Computer Science and Information Systems, pp. 1701–1708 (2016)
5. Bozzano, M., Villafiorita, A.: Design and Safety Assessment of Critical Systems. CRC Press (2011)
6. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 8559, pp. 334–342. Springer (2014)
7. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Computer Aided Verification (CAV'2007), *LNCS*, vol. 4590, pp. 158–163. Springer, Berlin, Germany (2007)
8. Idris, I.: Python Data Analysis. Packt Publishing Ltd. (2014)
9. Jensen, K., Kristensen, L.: Coloured Petri nets. Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
10. Lee, A., Ihaka, R., Triggs, C.: Advanced Statistical Modelling. Course notes for University of Auckland Paper STATS 330 (2012)

11. Matyasik, P., Szpyrka, M., Wypych, M., Biernacki, J.: Communication between agents in Alvis language. In: Proc. of Mixdes 2016, the 23nd International Conference Mixed Design of Integrated Circuits and Systems, pp. 448–453. Łódź, Poland (2016)
12. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. O'Reilly Media, Sebastopol, CA, USA (2008)
13. Samolej, S., Rak, T.: Simulation and performance analysis of distributed internet systems using TCPNs. Informatica (Slovenia) **33**(4), 405–415 (2009)
14. Szpyrka, M., Biernacki, J., Biernacka, A.: Tools and methods for RTCP-nets modelling and verification. Archives of Control Sciences **26**(3), 339–365 (2016). DOI 10.1515/acsc-2016-0019
15. Szpyrka, M., Matyasik, P., Biernacki, J., Biernacka, A., Wypych, M., Kotulski, L.: Hierarchical communication diagrams. Computing and Informatics **35**(1), 55–83 (2016)
16. Szpyrka, M., Matyasik, P., Mrówka, R.: Alvis – modelling language for concurrent systems. In: P. Bouvry, H. Gonzalez-Velez, J. Kołodziej (eds.) Intelligent Decision Systems in Large-Scale Distributed Environments, *Studies in Computational Intelligence*, vol. 362, chap. 15, pp. 315–341. Springer-Verlag (2011)
17. Szpyrka, M., Matyasik, P., Podolski, L., Wypych, M.: Simulation of multi-agent systems with Alvis Toolkit. In: L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. Zadeh, Z. J. (eds.) Artificial Intelligence and Soft Computing. ICAISC 2017, *LNCS*, vol. 10246, pp. 599–608. Springer-Verlag (2017). DOI 10.1007/978-3-319-59060-8 54
18. Szpyrka, M., Matyasik, P., Wypych, M., Biernacki, J., Podolski, L.: Alvis Modelling Language (2017). URL http://alvis.kis.agh.edu.pl