Received 9 August 2022, accepted 15 September 2022, date of publication 26 September 2022, date of current version 6 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3210191

RESEARCH ARTICLE

Alvis Approach to Modeling and Verification of Real-Time Systems Running on Single-Processor Environment

MARCIN SZPYRKA, (Senior Member, IEEE), JAROSŁAW BANIEWICZ, AND ANDREI KARATKEVICH
Department of Applied Computer Science, Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, AGH University of Science and Technology, 30-059 Kraków, Poland

Corresponding author: Marcin Szpyrka (mszpyrka@agh.edu.pl)

This work was supported by the Akademia Gorniczo-Hutnicza Krakow (AGH).

ABSTRACT Alvis is a formal modelling language developed primarily for modelling concurrent systems including real-time systems. The prepared model is compiled, and the resulting runnable model allows, e.g., to generate the state space and to verify the model using model checking techniques. The default version assumes that each system component (agent) executes its computations in parallel and that the hardware platform is not a shared resource. This paper describes an alternative approach in which agents remain concurrent, but compete for access to a single shared processor. This situation significantly affects the correctness of the system, in particular from the perspective of the timing properties of real-time systems. The key element of the presented solution is the fact that changing the runtime environment comes down to changing compiler options and does not require changes to the original model. This paper presents a method of designing Alvis language models for a single-processor hardware platform. The presented concepts are illustrated with examples.

INDEX TERMS Alvis language, discrete-time systems, formal languages, single-processor hardware, system analysis and design, system verification.

I. INTRODUCTION

The methods of formal modelling and verification of real-time systems have been developed for a long time; however, their application in engineering practise is still very limited. The main obstacles are insufficient knowledge and understanding of such models by the engineers and a gap between the models used by engineers and the formal models, making practical application of the last ones difficult. For this reason, efforts have been made to design models that can be both practical enough to allow the modelling of real systems and formal enough to allow the application of mathematical verification methods [1], [2]. One of the attempts to create such a modelling framework is Alvis.

The Alvis language [3], [4] is designed for modelling concurrent systems. The syntax typical of high-level programming languages and a graphical language [5] for modelling communication connections between agents are the key elements that distinguish it from popular formal methods such as Petri nets [6], [7], process algebras [8] or timed automata [9].

All previous papers on Alvis discussed its multiprocessor version. It assumes that each agent performs its computations in parallel, i.e., it has access to its own processor, regardless of the number of agents in the model. The assumption of unlimited parallelism is typical of the formal methods mentioned above. In practical applications, especially when we consider embedded systems, a system built based on such a model and running on an architecture with a limited number of processors may not possess the properties that the model possesses. One solution to this problem is the modelling of the hardware architecture inside the model itself, which can significantly affect its complexity and design time.

The associate editor coordinating the review of this manuscript and approving it for publication was Mauro Gaggero.

This paper presents a method for designing Alvis language models for a single-processor hardware platform. Such models are based on the assumption that only one processor is available and all agents compete for access to that processor. The key element of the presented solution is the fact that changing the runtime environment does not require changing the model at all. From the theoretical point of view, an Alvis model is a triple. The last element of the triple called *system layer* describes the runtime environment. A system layer is not defined by the user but is chosen from the predefined set of such layers. The selection of the system layer is realised by indicating the appropriate option of the *Alvis Compiler*. This means that the same model can be verified assuming that it will be executed in either a multiprocessor environment or a single-processor environment. To our knowledge, Alvis is the only formal language that allows this approach to formal model verification.

In this paper, the models are described with a system layer denoted by α^1_{FPPS} (*Fixed Priority Preemptive Scheduling*), which is an alternative to the only system layer previously available α^0 (unlimited multiprocessor layer). The relatively complete description of the timed version of Alvis with the α^0 system layer is given in [4]. The main contributions of the paper can be summarised as follows:

- we show a new single-processor system layer for Alvis language;
- we provide an updated definition of a model state that takes into account the hierarchical priority queue and system interrupts;
- we describe the algorithm for LTS generation for the new system layer;
- the presented idea is illustrated with simple case studies to show the influence of the choice of a system layer on the model properties.

The paper is organised as follows. Section II provides a comparison of Alvis with other formal methods. Section III contains a short description of the Alvis modelling and verification process and the related tools. The formal definition of Alvis models and a description of the α^1_{FPPS} system layer are provided in Section IV. Section V deals with the states of a model, transitions between states, and the LTS graph generation algorithm. Models of concurrent systems used to illustrate the approach are presented in Section VI. A short summary is given in the final section.

II. RELATED APPROACHES

One of the popular formalisms used for modelling of concurrent systems are the Petri nets. Their classical form is rather abstract, but there are numerous extensions of the model that allow us to catch many behavioural details, such as coloured Petri nets [6]. Such enriched Petri nets are represented, e.g., by high-level Petri nets supported by CPN Tools [10], [11], [12]. Comparing such nets with Alvis, one can see that Alvis has richer possibilities in data proceeding because the Haskell language, used in Alvis, is more powerful than the subset of the standard ML language used in CPN Tools. The structure of the system can be more complex in CPN nets, as far as the Petri nets make possible the different ways of forking and joining of parallel processes; on the other hand, the module structure of Alvis models makes them more regular and easier to handle. The communication between agents via ports in Alvis is similar to the transition firings in CPN, and the sent and received values can be modelled by the coloured tokens. However, such Alvis features as the non-blocking instructions cannot be directly expressed in CPN. Besides, there is no direct distinction between active and passive agents in CPN, as it is in Alvis. Such kinds of agents can be modelled with CPN, but the distinction in Alvis makes the models more understandable.

Other known Petri net based formalisms used for the modelling of concurrent discrete control systems are Grafcet [13], [14] and SFC being its further development [15], [16]. An SFC diagram consists of steps connected by transitions. Actions are associated with steps, and logical conditions are associated with transitions. The actions can be expressed in the Pascal-like ST language, which provides possibilities of data proceeding similar to Haskell used in Alvis. Priorities and time dependencies can also be used. However, the steps are typically active for short periods of time, not permanently, as some agents in Alvis. The possibilities of communication between parallel steps are limited. They can communicate via global data, and the parallel sequences of steps can be synchronised by means of transitions, but there are no direct mechanisms for sending and receiving data via ports. Like in the case of CPN, such Alvis possibilities as non-blocking instructions cannot be represented directly.

The scheduling methods used in the mentioned models are rather simple. Simulation of a colored Petri net in CPN tools is performed in such a way that at each step at most one transition is executed, and if more than one is firable, one of them is selected randomly. SFC models are supposed to be synchronous; their execution on a single processor or controller is performed in such a way that at each scan cycle the actions associated with each active SFC step are executed. The order of execution of parallel active steps within a single scan cycle is undefined [17].

Timed automata [9] are one more model allowing to represent time dependencies, delays, and communication. A timed automaton can be understood as a finite state machine enhanced with real-valued clocks. Timed automata can be successively used for modelling real time systems [18]. A network consisting of such automata can specify, in certain cases, the systems equivalent to Alvis models. However, timed automata, including some of their extensions (such as the hybrid automata used for modelling cyber-physical systems [19]), have no possibility of data processing using the algorithms described in programming languages, and their model of communication is simple, compared to communication in Alvis. Besides, there is no direct way to specify the duration of execution of an action, unlike in Alvis. Because of this, the Alvis models can be much more detailed than the timed automata models. Timed automata are easier for analysis, but their modelling power is weaker compared to Alvis. It is supposed that every automaton in a timed automata network can have its own clock, there is no global clock (in an Alvis model such a clock exists), and the scheduling methods supposing execution on a single processor are not considered for such networks.

There exists a family of formalisms using parallel and hierarchical compositions of automata, based on statecharts [20], [21]. Communication between the state machines in statecharts is performed via events. Different execution models, also using preemption, are considered for statecharts.

Hierarchical Concurrent Finite State Machine (HCFSM) [22], [23], [24] is a further development of statecharts. It includes, except of the possibility of parallel and hierarchical composition of automata, also various additional features, such as timeouts (allowing to model the non-blocking instructions as in Alvis). A scheduler that manages the execution of an HCFSM system in a synchronous way is described in [22].

HCFSMs in their basic form do not allow complex data proceeding. Such a possibility provides an enhancement with a sequential program model, known as a Program State-Machine (PSM) [23]. This formalism is similar to Alvis in the sense that the modules acting in parallel can execute complex algorithms. An example of a language that supports PSMs is SpecCharts [25], later developed as SpecC [26]. SpecCharts are a combination of hierarchical concurrent state machines and VHDL used for specification of their behaviour. Communication between automata is based on the VHDL concept of signals. Each VHDL process is executed by a separate controller or processor. A SpecCharts model can be executed on a single processor, using various scheduling methods.

One of the classical approaches to control the behaviour of discrete event systems is known as the Supervisory Control Theory of Discrete Event Systems (SCTDES) [27], [28]. In this approach, which initially was developed for untimed systems and later extended to timed and real-time DES, a system is modelled as one or more automata (timed if necessary), and a supervisor is synthesised, also being an automaton, which can disable certain transitions to avoid entering the system into the undesired states. Such a supervised system can be represented in Alvis, after applying an appropriate algorithm of the supervisor synthesis.

There exist methods of scheduling for the parallel systems running on a single processor based on the SCTDES formal framework. In such methods, the scheduling is enforced by the supervisor. Different variants of the approach are intended to obtain a scheduling that meets the timing constraints under certain assumptions [29], even when some faults are possible [30], and to avoid the processor idling in the presence of ready-to-execute tasks [31]. Introducing a supervising agent into an Alvis system modelled with α^1_{FPPS} system layer can provide the corresponding scheduling if the highest priority is assigned to the supervisor.

III. MODELLING WITH ALVIS

An Alvis model is a set of components called agents (the name was taken from the CCS process algebra [32]). We distinguish active and passive agents, which resemble the tasks and protected objects of the Ada language [33], respectively. The agents usually run concurrently, communicate one with another, compete for shared resources, etc. From the user's perspective, a model is built as two layers. The code layer defines the behaviour of individual agents, while the communication diagram [5] defines the communication connections between agents. In addition, the communication diagram is a graphical representation of the model that allows for an easy understanding of the model structure from the control and data flow perspective. The behaviour of agents is defined using Alvis language statements, which are supported by the Haskell functional language [34]. Haskell is used to define the data types and functions that manipulate the data. The user-defined functions in Haskell can be attached to a model. A call of such a function, for example, to sort a list of numbers, is considered as a single transition from the model behaviour perspective. Finally, Alvis Compiler translates an Alvis model into its Haskell representation, which is used for simulation and verification purposes. The generated file can be directly compiled using the GHC Compiler. Nevertheless, the source code can be modified by the user in an arbitrary text editor.

The Alvis language is supported by a prototype modelling and verification environment that includes an editor and a compiler. The editor is used to design a model (communication diagram and code layer) and the compiler translates the model into an executable file. The compilation result depends on the compiler options selected. By default, the resulting executable file is used to generate an LTS (Labelled Transition System) for model verification. It is also possible to obtain a file for simulation purposes, as well as to implement the user's own verification methods. An LTS can be exported to the DOT, Aldebaran and CSV file formats. These formats are used for model checking verification using external tools such as nuXmv [35] and CADP [36]. Statistical analysis using the R and Python languages is also possible.

IV. MODELS WITH α^1 SYSTEM LAYER

The Alvis Editor supports hierarchical modelling. Using of hierarchies is convenient when designing the complex models. Before compilation, a model must be transformed into the equivalent non-hierarchical version. Because the purpose of the paper is to provide the semantics of Alvis models with single-processor layer, it is enough to consider non-hierarchical models only [4], [37].

Definition 1: An Alvis model is a triple $A = (D, B, \varphi)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ is a non-hierarchical communication diagram, B is a syntactically correct code layer, and φ is a 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.

We need to introduce some notation to give the definition of a non-hierarchical communication diagram:

- $\mathcal{P}(X)$ – the set of all ports of agent X,
- $\mathcal{P}_{in}(X)$ – the set of input ports of agent X,
- $\mathcal{P}_{out}(X)$ – the set of output ports of agent X,
- $\mathcal{P}_{proc}(X)$ – the set of procedure ports of agent X,
- $\mathcal{P}(W) = \bigcup_{X \in W} \mathcal{P}(X)$, where W is a set of agents, similarly $\mathcal{P}_{in}(W)$, $\mathcal{P}_{out}(W)$, $\mathcal{P}_{proc}(W)$,
- $\mathcal{P}$ – the set of all model ports, similarly $\mathcal{P}_{in}$, $\mathcal{P}_{out}$, $\mathcal{P}_{proc}$.

An input (output) port is a port with at least one one-way connection leading to (from) it or with at least one two-way connection. In the case of passive agents, the ports can represent the procedures (services) provided by the agent.

Definition 2: A non-hierarchical communication diagram is a triple $D = (\mathcal{A}, \mathcal{C}, \sigma)$, where: $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$ is the set of agents, $\mathcal{A}_A$ and $\mathcal{A}_P$ are the disjoint sets containing active and passive agents respectively; $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$ is the communication relation, such that:

$$\forall_{X \in \mathcal{A}} (\mathcal{P}(X) \times \mathcal{P}(X)) \cap \mathcal{C} = \emptyset, \tag{1}$$

$$\mathcal{P}_{proc} \cap \mathcal{P}_{in} \cap \mathcal{P}_{out} = \emptyset, \tag{2}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_A) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow q \in \mathcal{P}_{proc}, \tag{3}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_A)) \cap \mathcal{C} \Rightarrow p \in \mathcal{P}_{proc}, \tag{4}$$

$$(p, q) \in (\mathcal{P}(\mathcal{A}_P) \times \mathcal{P}(\mathcal{A}_P)) \cap \mathcal{C} \Rightarrow$$
$$(p \in \mathcal{P}_{proc} \wedge q \notin \mathcal{P}_{proc}) \vee (q \in \mathcal{P}_{proc} \wedge p \notin \mathcal{P}_{proc}), \tag{5}$$

and $\sigma : \mathcal{A}_A \to \{False, True\}$ is the start function that denotes the initially activated agents.

Each element belonging to $\mathcal{C}$ is called a communication channel (connection). A communication channel always connects two ports, with the following restrictions: (1) – A connection cannot be defined between ports of the same agent. (2) – Procedure ports are either input or output ones. (3), (4) – A connection between an active and a passive agent must be a procedure call. From conditions (2)-(4), it follows that any connection with a passive agent must be an one-way connection. (5) – A connection between two passive agents must be a procedure call from a non-procedure port. If $(p, q) \in \mathcal{C}$ then p is an output port and q is an input port of the (p, q) connection.

A syntactically correct code layer means not only the correct syntax of both Alvis and Haskell statements, but also that, for example, only the input ports may be used as arguments of *in* statements, and only the output ports may be used as arguments of *out* statements. Syntax validation of the code layer is carried out by the compilers.

Modelling assuming a single-processor environment requires a process scheduling algorithm. There exists a range of such algorithms used in operating systems and real-time systems [38]. The simplest of them is First Come First Served (FCFS) algorithm, which uses a single queue. This method can lead to long waiting times, and it is nonpreemtive. Virtually all modern systems use various forms of preemptive multitasking. Among them, one of the simplest is Shortest Job First (SJF), which requires the evaluation of the time amount required for the processes to be completed. There are difficulties with using it when the evaluation cannot be performed exactly enough or when the processes are cyclic, which is typical for real-life systems. Besides, the algorithms which do not allow to assign priorities to the processes are not sufficient for complex systems.

The algorithms for priority scheduling usually have to deal with situations where there are still several processes with the same priorities. Then, a queue can be created for every priority level, and within the same priority the FCFS approach is used. The Fixed Priority Preemptive Scheduling is the basic variant of such algorithms. As far as a scheduling policy in a general modelling method cannot be too specific and complicated, but - especially for real-time systems - has to take into account priorities and preemption, the Fixed Priority Preemptive Scheduling (FPPS) algorithm is selected for Alvis models for a single-processor platform.

To add the scheduling algorithm to Alvis, we provide a new system layer denoted by α^1_{FPPS}. The basis of this system layer is the assumption that only one processor is available, and all active agents compete for access to this processor. This means that at most one agent can be in the *running* mode (executing statements), but many agents may be ready to execute statements, waiting for access to the processor (*Ready* mode). The Fixed Priority Preemptive Scheduling algorithm is used to rank the agents. The algorithm is embedded in the α^1_{FPPS} system layer.

A priority can be assigned to each agent in an Alvis model. The highest priority is represented by 0. The agents in the *ready* mode are placed into a two-dimensional queue structure Q. The structure can be treated as a sequence of the FIFO queues (*levels* $Q_0, Q_1, \ldots, Q_n$), each of which stores agents with the same priority (see Fig. 1). The first agent at the given level is placed at position 0. The agent at position 0 at the highest level in the two-dimensional queue is the first to take over the processor. Of course, any queue Q_i may be empty in the current state. The highest level means here the non-empty queue Q_i with the highest priority.

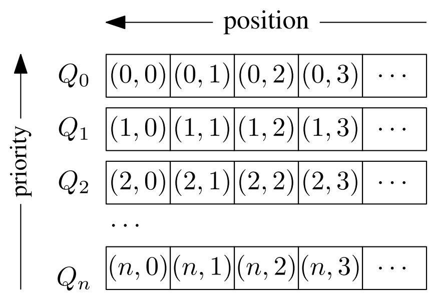

position
priority
Q_0
$(0,0)$ $(0,1)$ $(0,2)$ $(0,3)$ $\cdots$
Q_1
$(1,0)$ $(1,1)$ $(1,2)$ $(1,3)$ $\cdots$
Q_2
$(2,0)$ $(2,1)$ $(2,2)$ $(2,3)$ $\cdots$
$\cdots$
Q_n
$(n,0)$ $(n,1)$ $(n,2)$ $(n,3)$ $\cdots$

FIGURE 1. Priority queue.

To promote an agent from the *ready* to *running* mode, the scheduler performs the following steps:

- The highest level Q_r is selected.
- Assume that X_r is at position 0 in Q_r and X_c is the agent that controls the processor currently. If the priority of X_r

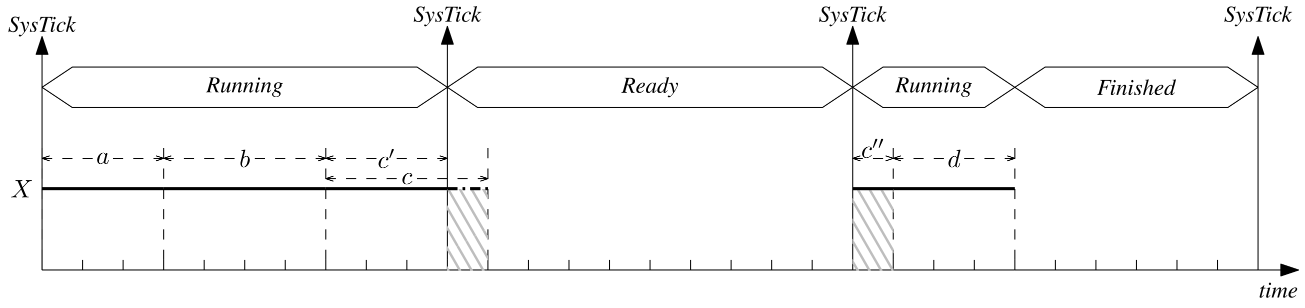


FIGURE 2. System interruption *SysTick* breaks a statement execution.

is greater than or equal to the priority of X_c then X_r is promoted to the Running mode and removed from Q_r. Otherwise, the queue remains unchanged and agent X_c remains in the *running* mode.

- If X_r is promoted to the *running* mode remaining agents in Q_r are shifted by one position.
- If the new mode of X_c is *ready* (X_r is ready to continue the current instruction or to execute the next one), X_c is placed at the end of the corresponding level.

The scheduling algorithm is called every t time-units, where the value of t is fixed for the given model. The Alvis time model is based on the idea of a global clock used to measure the duration of executed statements (transitions). To each statement in the model its duration is assigned. The time-units used in a given model are strictly connected with the model interpretation.

A fixed period for calling the scheduling function means that it may be called when the current statement is not complete. In this case, the execution of the statement is interrupted and postponed until the agent accesses the processor again. The scheduling function may also be called additionally when the agent using the processor cannot execute another statement (it has finished its work or must wait, for example, for another agent to finish a communication).

Let us consider the example presented in Fig. 2. Agent X executes a sequence of statements a, b, c, d and finishes its work. The *SysTick* interruption occurs every 10 time-units. During the first period in the *running* mode, the agent completed statements a and b but failed to complete c. One time-unit is missing to complete the statement. The execution of the c statement is divided into two phases: c' in the first period in the *running* mode and c'' in the next.

V. STATES, TRANSITIONS AND LTS GRAPHS

To describe the state of an Alvis model with the α^1_{FPPS} system layer we need some information: states of the agents, contents of the Q queue, and the number of time-units until the next call of the scheduling function.

Definition 3: A state of an agent X is a tuple

$$S(X) = (am(X), pc(X), ci(X), pv(X)), \tag{6}$$

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 mode specifies the current activity of the agent. In addition to the already mentioned *running* (X) and *ready* (R) modes, the following modes are allowed: *init* (I – the agent is idle before any activity), *finished* (F – the agent finished its activity), *taken* (T – the passive agent is running one of its procedures) and *waiting* (W). The last mode in the case of active agents denotes that the agent performed an action and is waiting for an event, e.g., releasing a currently inaccessible procedure after calling it. For passive agents, it means that the agent is idle (it is possible to call its accessible procedure).

The program counter stores the ordinal number of the current statement. The context information list contains some extra information about the current state (see Table 1 for more details). The parameters values tuple contains the current values of the local agent's variables.

Definition 4: Let $A = (D, B, \alpha^1_{FPPS})$ be a non-hierarchical Alvis model, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ and $\mathcal{A} = \{X_1, \ldots, X_n\}$. Let Q be a a two-dimensional priority queue as described in Section IV and t be the number of time-units to the next call of the scheduling function. A state of a model A is a tuple

$$S = (S(X_1), \ldots, S(X_n), Q, t), \tag{7}$$

where $S(X)$ is the state of agent X.

To define the initial state of a model, we must define in what order the active agents X_i and X_j are placed in the queue, if they have the same priority. Some model verification tools, for example, CADP, require that the initial state be specified unambiguously. For these practical reasons, it is assumed that the compiler places agents of the same priority in the order in which they are defined in the code layer. The user can modify the generated Haskell file and define a different initial state.

The initial state is defined as follows:

- $am(X)$ = R, for any active agent X such that $\sigma(X) = True$; $am(X)$ = I, for any active agent X such that $\sigma(X) = False$; and $am(X)$ = W, for any passive agent X;
- $pc(X) = 1$ for any active agent X in the R mode and $pc(X) = 0$ for other agents.
- $ci(X) = [\ ]$ for any active agent X; and $ci(X)$ contains names of all accessible procedures of X together with the direction of parameters transfer, e.g. $in(a)$, $out(b)$, etc. for any passive agent X.
- For any agent X, $pv(X)$ contains X parameters with their initial values.

TABLE 1. Context information list entries.

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

- All agents that are in ready mode are placed in the Q queue according to their priority value and the order in which their behaviour is defined.
- Agent X_r determined according to the method presented in Section IV is removed from Q and its mode is set to X.

From a theoretical perspective, each Alvis statement is represented by one or more transitions. The exception is the *proc* statement, which is only used to enclose a piece of code that defines a single service and to define a condition for the availability of that service. The list of all Alvis transitions for the models is given in Table 2.

TABLE 2. Transitions.

Transition	*Description*
TDelay	a *delay* statement execution
STDelayEnd	termination of agent's suspension
TExec	an *exec* statement execution
TExit	an *exit* statement execution
TIn	initialisation of communication when no peer is available (execution suspension) or reading data from a procedural port – *in* statement
TInAP	calling a procedure by an active agent – *in* statement
STInAP	system version of *TInAP* – for wake up purposes
TInPP	calling a procedure by a passive agent – *in* statement
STInPP	system version of *TInPP* – for wake up purposes
TInF	finalisation of a communication with an active agent – *in* statement
STInEnd	abandonment of communication – non-blocking *in* statement
TJump	a *jump* statement execution
TLoop	entering a *loop* statement
TLoopEvery	entering a *loop every* statement
STLoopEnd	termination of current *loop every* run
TNull	*null* statement execution
TOut	initialisation of communication when no peer is available (execution suspension) or writing data to a procedural port – *out* statement
TOutAP	calling a procedure by an active agent – *out* statement
STOutAP	system version of *TOutAP* – for wake up purposes
TOutPP	calling a procedure by a passive agent – *out* statement
STOutPP	system version of *TOutPP* – for wake up purposes
TOutF	finalisation of a communication with an active agent – *out* statement
STOutEnd	abandonment of communication – non-blocking *out* statement
TSelect	entering a *select* statement
TStart	a *start* statement execution
STTime	passage of time
STSysTick	running the scheduling algorithm

Simple statements such as *exec*, *exit*, *jump*, *loop*, *null*, *select* and *start* are represented by single transitions that denote the execution of such a statement. In some cases, it is necessary to have a pair of transitions; for example, *TDelay* represents the execution of a *delay* statement and as a result the suspension of the agent. To terminate this suspension after the suitable period of time, the transition *STDelayEnd* must be executed. The latter transition belongs to the set of so-called *system transitions*. The names of these transitions start with a capital S. All these transitions represent some activities of the model run-time environment. A special system transition, named *STTime*, represents a passage of time. It is used when there are no transitions available at 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. Other system transitions are used for waking up agents that, for some reasons, are in the *waiting* mode.

Finally, due to the different ways in which communication between agents is interpreted (communication between active agents, communication between an active and a passive agent, communication between passive agents, blocking communication, non-blocking communication, initiating communication, ending communication), *in* and *out* statement are represented by several transitions. More details on the meaning of the different transitions can be found in [4] and in the Alvis language manual [39].

For each transition the *enable* and *firing* rules are defined (see [4], [39] for details). The *enable rule* specifies when the given transition can be executed, while the *firing rule* specifies what can be the result of the transition execution. The activity of a transition is always considered for a given agent and statement. An active agent can execute its statements independently, but a passive agent always works in the context of an active agent.

Assume $A = (D, B, \alpha^1_{FPPS})$ is an Alvis model with the current state S and consider a non-system transition t for agent X. Let $context(X)$ denote the active agent whose context is used by the passive agent X. The general conditions for the activity of t are as follows:

1) If X is an active agent, then X is in the *running* mode, otherwise X is in the *taken* mode, and $context(X)$ is in the *running* mode.
2) The transition t refers to the current agent's statement, i.e. for the *TDelay* transition the current agent statement must be a *delay* statement, for the *TExec* transition the current agent statement must be an *exec* statement etc.

```
1:  $s_0 \leftarrow init\_state$                                  ▷ read the initial state from the model
2:  $t_0 \leftarrow init\_time$                                   ▷ read the SysTick period from the model
3:  $q_0 \leftarrow createPriorityQueue(s_0)$                     ▷ generate the initial priority queue
4:                 ▷ A node contains: number, state, queue, time to the next SysTick, list of outgoing arcs
5:  $unprocNodes \leftarrow [(0, s_0, q_0, t_0, [])]$                           ▷ unprocessed nodes
6:  $trans \leftarrow alpha1EnableTransitions(s_0, q_0, t_0)$               ▷ list of enabled transitions
7:  $auxList \leftarrow [(0, s_0, q_0, t_0, [])]$                            ▷ already generated nodes
8:  $lts \leftarrow ltsgen(unprocNodes, trans, auxList)$                    ▷ recursive LTS generation
9:
10: function $ltsgen(unprocNodes, trans, auxList)$
11:     if $unprocNodes \neq [\,] \wedge trans \neq [\,]$ then
12:         $(number, state, queue, time, arcs) \leftarrow head(unprocNodes)$
13:         $tr \leftarrow head(trans)$
14:         $trs \leftarrow tail(trans)$
15:         $shift \leftarrow timeShift(state, tr, time)$
16:         $results \leftarrow timeFire(shift, tr, state, queue, time)$
17:         $(newUnprocNodes, newAuxList) \leftarrow update((unprocNodes, auxList), tr, shift, results)$
18:         return $ltsgen(newUnprocNodes, trs, newAuxList)$
19:     else if $length(unprocNodes) > 1 \wedge trans = [\,]$ then
20:         $node \leftarrow head(unprocNodes)$
21:         $nodes \leftarrow tail(unprocNodes)$
22:         $(number, state, queue, time, arcs) \leftarrow head(nodes)$
23:         $trans \leftarrow alpha1EnableTransitions(state, queue, time)$
24:         return $[node] + ltsgen(nodes, trans, auxList)$
25:     else                                              ▷ $length(unprocNodes) = 1 \wedge trs = [\,]$
26:         return $unprocNodes$
27:     end if
28: end function
```

FIGURE 3. LTS graph generation algorithm.

Some extra conditions must be fulfilled for communication transitions (*TIn, TInAP, TInPP, TInF, TOut, TOutAP, TOutPP, TOutF*). For example, for calling procedure p of passive agent Y by an active agent X (transition *TInAP*), agent Y must be in the *waiting* mode, and output procedure p must be accessible (see [4], [39] for details).

The modelling environment for the Alvis language does not offer the built-in formal analysis methods, except where the user provides the analysis methods implemented in Haskell. Verification is carried out using the external model checkers. An Alvis model is compiled into its Haskell representation (Intermediate Haskell Representation – IHR). This version is used to generate the state space of the model and store it into a Labelled Transition System.

Definition 5: A Labelled Transition System (LTS) is a four-tuple LTS $= (S, E, L, S_0)$, where: S is the set of states, L is the set of action labels, $E \subseteq S \times L \times S$ is the set of arcs, and S_0 is the initial state.

For a model $A = (D, B, \alpha^1_{FPPS})$, the set S contains the states as defined in Definition 3. Each label contains two pieces of information: the name of the transition and the number of time units that have elapsed between the states connected by the corresponding arc.

A pseudocode representation of the LTS graph generation algorithm is shown in Fig. 3. We start with collecting information about the initial state (line 1) and the *SysTick* period (line 2) from the considered model. Then the initial value for the priority queue is generated (line 3). The algorithm uses a recursive function *ltsgen* for the LTS generation (line 10). The function takes three arguments:

1) a list of unprocessed nodes,
2) the list of transitions enabled in the first state of the unprocessed nodes,
3) an auxiliary list of already generated nodes.

As the initial value for both lists of nodes, a list with the initial node is taken (lines 5 and 7). A node is a Haskell wrapper for a state. It contains some extra information, such as the node number or agent names, to make an LTS more readable. See Fig. 7 for more details.

We use the *alpha1EnableTransitions* function (line 6) to generate the list of transitions enabled in the initial state.

We consider three cases within the function *ltsgen*. The first case is when there is at least one enabled transition for the first state in the *unprocNodes* list. We take the first of the enabled transitions (line 13) and determine by how many time units the realisation of the model can proceed for this transition (function *timeShift*, line 15). The function determines the maximal time shift for the transition. This value is selected so as not to lose any information about the changes of states of the analysed system. Then we use the determined value to calculate the result of the *tr* transition firing (line 16). The *update* function (line 17) udpates the list of unprocessed nodes and the list of auxiliary nodes. For each state from the list of new states, the function checks whether the auxiliary list already contains a node with such a state. If so, only a new arc is added to the first node of unprocessed nodes. Otherwise, a new node is added to both lists (and the corresponding arc to the first node of unprocessed nodes). Finally, the *ltsgen* function is called again with updated node lists and a reduced list of enabled transitions.

The second case is when there are no enabled transitions for the first state in the *unprocNodes* list. In such a case, the first node is omitted (it is included into the final LTS), and the *ltsgen* function is called again with the updated list of unprocessed nodes, and a new list of enabled transitions generated for the first of these nodes. Finally, if the list of unprocessed nodes contains only one node and the list of enabled transitions is empty, we finish the generation of the LTS.

The process of building an LTS graph poses many problems due to the state-space explosion. The factors that mostly affect the number of states are the number of active agents in the modelled system and the number of local variables that can potentially have many internal states. It is easy to define an agent with an infinite number of states. In the most general case, the time complexity is $\mathcal{O}(n \log n)$, where n is the number of states, and the memory complexity is $\mathcal{O}(n)$.

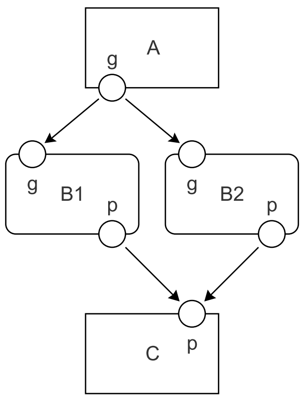
VI. CASE STUDIES

A. EXAMPLE 1

Let us consider the example of an Alvis model shown in Fig. 4. The model is composed of four agents – two active agents $B1$ and $B2$, and two passive agents A and C. Each of the active agents collects an initial value for computation from the passive agent A, runs its function, and places the result in the passive agent C. Because all communication channels are defined between the active agents and the passive agents, they are treated as the procedure calls of the passive agent. The model under consideration also shows how easily we can add our own functions implemented in Haskell to a model in Alvis. It is worth mentioning that an execution of such a function is treated as a single transition in the corresponding LTS, regardless of body of the function. The comments in Fig. 4 contain the instruction numbers and their assumed execution times.

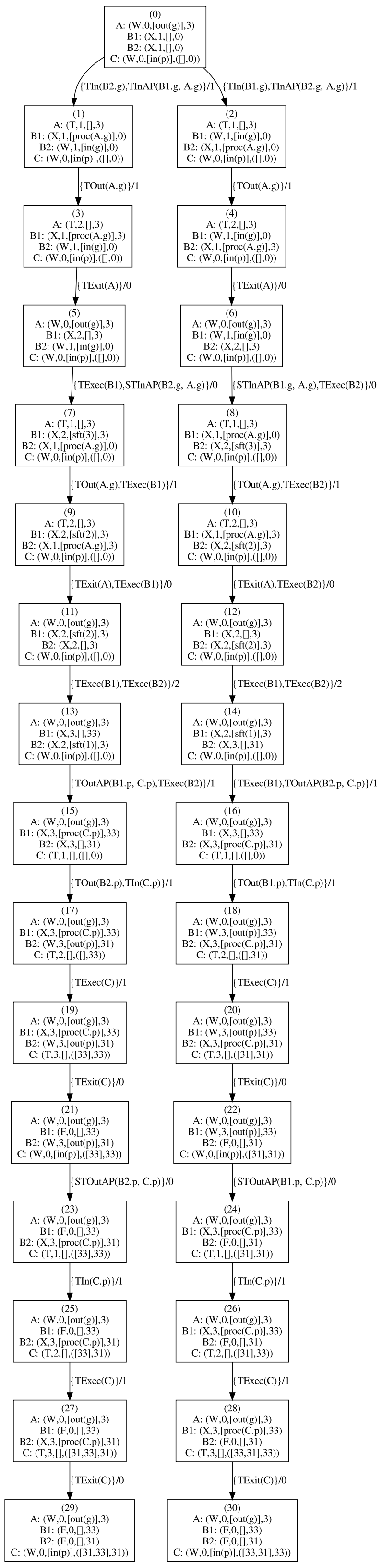
The same model can be compiled with different sets of options. By default, the α^0 system layer is used. The LTS generated for the model shown in Fig. 4 is presented in Fig. 5. The LTS contains two paths because all agents have been assigned the same default priority, so the selection of the agent who first calls the procedure $A.g$ is non-deterministic. Labels of arcs provide two pieces of information: a set of transitions that are executed in parallel and lead from the arc source state to the destination state and the time that elapses between these two consecutive states. It is easy to calculate that in both cases it takes 10 time-units to do all calculations.

Including the `-a1` option when calling the compiler changes the system layer to α^1_{FPPS}. The LTS generated for



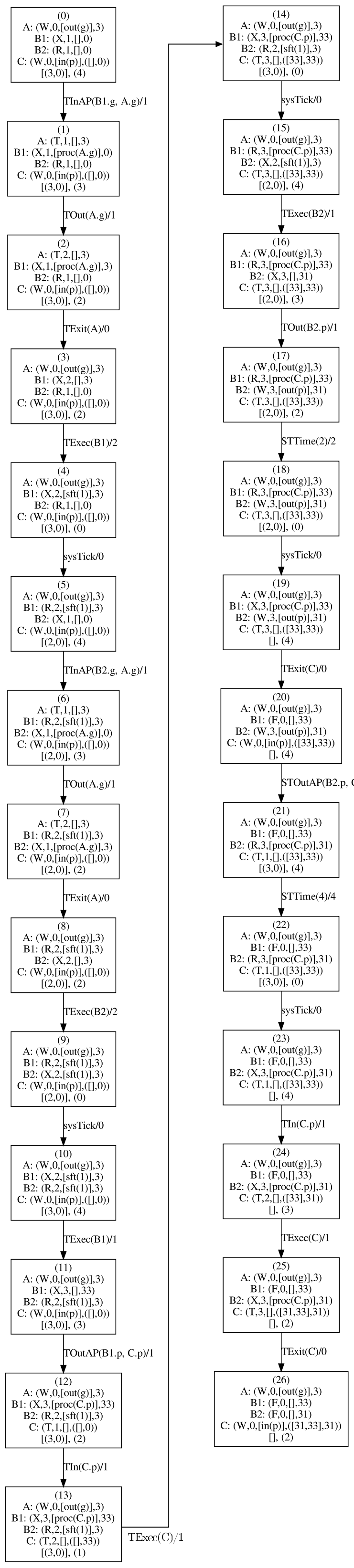
```
agent A {                    -- stat. no,   duration
  x :: Int = 3;
  proc g {
    out g x;                 -- 1           1
    exit;                    -- 2           0
  }
}

agent B1 {
  y :: Int = 0;
  in g y;                    -- 1           1
  y = f1 y;                  -- 2           3
  out p y;                   -- 3           1
}

agent B2 {
  y :: Int = 0;
  in g y;                    -- 1           1
  y = f2 y;                  -- 2           3
  out p y;                   -- 3           1
}

agent C {
  x :: [Int] = [];
  y :: Int = 0;
  proc p {
    in p y;                  -- 1           1
    x = y:x;                 -- 2           1
    exit;                    -- 3           0
  }
}

-- Example functions for data processing
f1 x = x^3 + 2*x
f2 x = x^3 - x + 7
```

FIGURE 4. Parallel data processing.



(0) A: (W,0,[out(g)],3) B1: (X,1,[],0) B2: (X,1,[],0) C: (W,0,[in(p)],([],0))
{TIn(B2.g),TInAP(B1.g, A.g)}/1
{TIn(B1.g),TInAP(B2.g, A.g)}/1
(1) A: (T,1,[],3) B1: (X,1,[proc(A.g)],0) B2: (W,1,[in(g)],0) C: (W,0,[in(p)],([],0))
(2) A: (T,1,[],3) B1: (W,1,[in(g)],0) B2: (X,1,[proc(A.g)],0) C: (W,0,[in(p)],([],0))
{TOut(A.g)}/1
(3) A: (T,2,[],3) B1: (X,1,[proc(A.g)],3) B2: (W,1,[in(g)],0) C: (W,0,[in(p)],([],0))
(4) A: (T,2,[],3) B1: (W,1,[in(g)],0) B2: (X,1,[proc(A.g)],3) C: (W,0,[in(p)],([],0))
{TExit(A)}/0
(5) A: (W,0,[out(g)],3) B1: (X,2,[],3) B2: (W,1,[in(g)],0) C: (W,0,[in(p)],([],0))
(6) A: (W,0,[out(g)],3) B1: (W,1,[in(g)],0) B2: (X,2,[],3) C: (W,0,[in(p)],([],0))
{TExec(B1),STInAP(B2.g, A.g)}/0
{STInAP(B1.g, A.g),TExec(B2)}/0
(7) A: (T,1,[],3) B1: (X,2,[sft(3)],3) B2: (X,1,[proc(A.g)],0) C: (W,0,[in(p)],([],0))
(8) A: (T,1,[],3) B1: (X,1,[proc(A.g)],0) B2: (X,2,[sft(3)],3) C: (W,0,[in(p)],([],0))
{TOut(A.g),TExec(B1)}/1
{TOut(A.g),TExec(B2)}/1
(9) A: (T,2,[],3) B1: (X,2,[sft(2)],3) B2: (X,1,[proc(A.g)],3) C: (W,0,[in(p)],([],0))
(10) A: (T,2,[],3) B1: (X,1,[proc(A.g)],3) B2: (X,2,[sft(2)],3) C: (W,0,[in(p)],([],0))
{TExit(A),TExec(B1)}/0
{TExit(A),TExec(B2)}/0
(11) A: (W,0,[out(g)],3) B1: (X,2,[sft(2)],3) B2: (X,2,[],3) C: (W,0,[in(p)],([],0))
(12) A: (W,0,[out(g)],3) B1: (X,2,[],3) B2: (X,2,[sft(2)],3) C: (W,0,[in(p)],([],0))
{TExec(B1),TExec(B2)}/2
(13) A: (W,0,[out(g)],3) B1: (X,3,[],33) B2: (X,2,[sft(1)],3) C: (W,0,[in(p)],([],0))
(14) A: (W,0,[out(g)],3) B1: (X,2,[sft(1)],3) B2: (X,3,[],31) C: (W,0,[in(p)],([],0))
{TOutAP(B1.p, C.p),TExec(B2)}/1
{TExec(B1),TOutAP(B2.p, C.p)}/1
(15) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (X,3,[],31) C: (T,1,[],([],0))
(16) A: (W,0,[out(g)],3) B1: (X,3,[],33) B2: (X,3,[proc(C.p)],31) C: (T,1,[],([],0))
{TOut(B2.p),TIn(C.p)}/1
{TOut(B1.p),TIn(C.p)}/1
(17) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (W,3,[out(p)],31) C: (T,2,[],([],33))
(18) A: (W,0,[out(g)],3) B1: (W,3,[out(p)],33) B2: (X,3,[proc(C.p)],31) C: (T,2,[],([],31))
{TExec(C)}/1
(19) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (W,3,[out(p)],31) C: (T,3,[],([33],33))
(20) A: (W,0,[out(g)],3) B1: (W,3,[out(p)],33) B2: (X,3,[proc(C.p)],31) C: (T,3,[],([31],31))
{TExit(C)}/0
(21) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (W,3,[out(p)],31) C: (W,0,[in(p)],([33],33))
(22) A: (W,0,[out(g)],3) B1: (W,3,[out(p)],33) B2: (F,0,[],31) C: (W,0,[in(p)],([31],31))
{STOutAP(B2.p, C.p)}/0
{STOutAP(B1.p, C.p)}/0
(23) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (X,3,[proc(C.p)],31) C: (T,1,[],([33],33))
(24) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (F,0,[],31) C: (T,1,[],([31],31))
{TIn(C.p)}/1
(25) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (X,3,[proc(C.p)],31) C: (T,2,[],([33],31))
(26) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (F,0,[],31) C: (T,2,[],([31],33))
{TExec(C)}/1
(27) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (X,3,[proc(C.p)],31) C: (T,3,[],([31,33],31))
(28) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (F,0,[],31) C: (T,3,[],([33,31],33))
{TExit(C)}/0
(29) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (F,0,[],31) C: (W,0,[in(p)],([31,33],31))
(30) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (F,0,[],31) C: (W,0,[in(p)],([33,31],33))

FIGURE 5. LTS - parallel version.



(0) A: (W,0,[out(g)],3) B1: (X,1,[],0) B2: (R,1,[],0) C: (W,0,[in(p)],([],0)) [(3,0)], (4)
TInAP(B1.g, A.g)/1
(1) A: (T,1,[],3) B1: (X,1,[proc(A.g)],0) B2: (R,1,[],0) C: (W,0,[in(p)],([],0)) [(3,0)], (3)
TOut(A.g)/1
(2) A: (T,2,[],3) B1: (X,1,[proc(A.g)],3) B2: (R,1,[],0) C: (W,0,[in(p)],([],0)) [(3,0)], (2)
TExit(A)/0
(3) A: (W,0,[out(g)],3) B1: (X,2,[],3) B2: (R,1,[],0) C: (W,0,[in(p)],([],0)) [(3,0)], (2)
TExec(B1)/2
(4) A: (W,0,[out(g)],3) B1: (X,2,[sft(1)],3) B2: (R,1,[],0) C: (W,0,[in(p)],([],0)) [(3,0)], (0)
sysTick/0
(5) A: (W,0,[out(g)],3) B1: (R,2,[sft(1)],3) B2: (X,1,[],0) C: (W,0,[in(p)],([],0)) [(2,0)], (4)
TInAP(B2.g, A.g)/1
(6) A: (T,1,[],3) B1: (R,2,[sft(1)],3) B2: (X,1,[proc(A.g)],0) C: (W,0,[in(p)],([],0)) [(2,0)], (3)
TOut(A.g)/1
(7) A: (T,2,[],3) B1: (R,2,[sft(1)],3) B2: (X,1,[proc(A.g)],3) C: (W,0,[in(p)],([],0)) [(2,0)], (2)
TExit(A)/0
(8) A: (W,0,[out(g)],3) B1: (R,2,[sft(1)],3) B2: (X,2,[],3) C: (W,0,[in(p)],([],0)) [(2,0)], (2)
TExec(B2)/2
(9) A: (W,0,[out(g)],3) B1: (R,2,[sft(1)],3) B2: (X,2,[sft(1)],3) C: (W,0,[in(p)],([],0)) [(2,0)], (0)
sysTick/0
(10) A: (W,0,[out(g)],3) B1: (X,2,[sft(1)],3) B2: (R,2,[sft(1)],3) C: (W,0,[in(p)],([],0)) [(3,0)], (4)
TExec(B1)/1
(11) A: (W,0,[out(g)],3) B1: (X,3,[],33) B2: (R,2,[sft(1)],3) C: (W,0,[in(p)],([],0)) [(3,0)], (3)
TOutAP(B1.p, C.p)/1
(12) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (R,2,[sft(1)],3) C: (T,1,[],([],0)) [(3,0)], (2)
TIn(C.p)/1
(13) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (R,2,[sft(1)],3) C: (T,2,[],([],33)) [(3,0)], (1)
TExec(C)/1
(14) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (R,2,[sft(1)],3) C: (T,3,[],([33],33)) [(3,0)], (0)
sysTick/0
(15) A: (W,0,[out(g)],3) B1: (R,3,[proc(C.p)],33) B2: (X,2,[sft(1)],3) C: (T,3,[],([33],33)) [(2,0)], (4)
TExec(B2)/1
(16) A: (W,0,[out(g)],3) B1: (R,3,[proc(C.p)],33) B2: (X,3,[],31) C: (T,3,[],([33],33)) [(2,0)], (3)
TOut(B2.p)/1
(17) A: (W,0,[out(g)],3) B1: (R,3,[proc(C.p)],33) B2: (W,3,[out(p)],31) C: (T,3,[],([33],33)) [(2,0)], (2)
STTime(2)/2
(18) A: (W,0,[out(g)],3) B1: (R,3,[proc(C.p)],33) B2: (W,3,[out(p)],31) C: (T,3,[],([33],33)) [(2,0)], (0)
sysTick/0
(19) A: (W,0,[out(g)],3) B1: (X,3,[proc(C.p)],33) B2: (W,3,[out(p)],31) C: (T,3,[],([33],33)) [], (4)
TExit(C)/0
(20) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (W,3,[out(p)],31) C: (W,0,[in(p)],([33],33)) [], (4)
STOutAP(B2.p, C.p)/0
(21) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (R,3,[proc(C.p)],31) C: (T,1,[],([33],33)) [(3,0)], (4)
STTime(4)/4
(22) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (R,3,[proc(C.p)],31) C: (T,1,[],([33],33)) [(3,0)], (0)
sysTick/0
(23) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (X,3,[proc(C.p)],31) C: (T,1,[],([33],33)) [], (4)
TIn(C.p)/1
(24) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (X,3,[proc(C.p)],31) C: (T,2,[],([33],31)) [], (3)
TExec(C)/1
(25) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (X,3,[proc(C.p)],31) C: (T,3,[],([31,33],31)) [], (2)
TExit(C)/0
(26) A: (W,0,[out(g)],3) B1: (F,0,[],33) B2: (F,0,[],31) C: (W,0,[in(p)],([31,33],31)) [], (2)

FIGURE 6. LTS - single processor version.

the α^1_{FPPS} system layer is shown in Fig. 6. This time there is only one path in the LTS and we need 22 time-units to perform calculations. Let us focus on the initial state for a moment. Both passive agents are in the *Waiting* mode and their procedures are accessible. Agent $B1$ is in the *Running* mode, while $B2$ is in the *Ready* mode. The Alvis Compiler encodes the priority queue with a list of pairs of integers. The first element indicates the order number of the agent in the agent list, and the second element indicates the agent priority. Finally, the last integer indicates the number of time-units to the next *SysTick* event. This initial state results from the order in which the agents are placed in the model. If we change the order of agents $B1$ and $B2$, agent $B2$ will be in the *Running* mode in the initial state. Of course, using different agent priorities will result in the agent with the higher priority being started from the initial state regardless of the order in which the agents are placed in the model.

As mentioned above, a node is a wrapper for a state. By modifying a function in Haskell, the user can easily change the way nodes are printed. The default version is shown in Fig. 7.

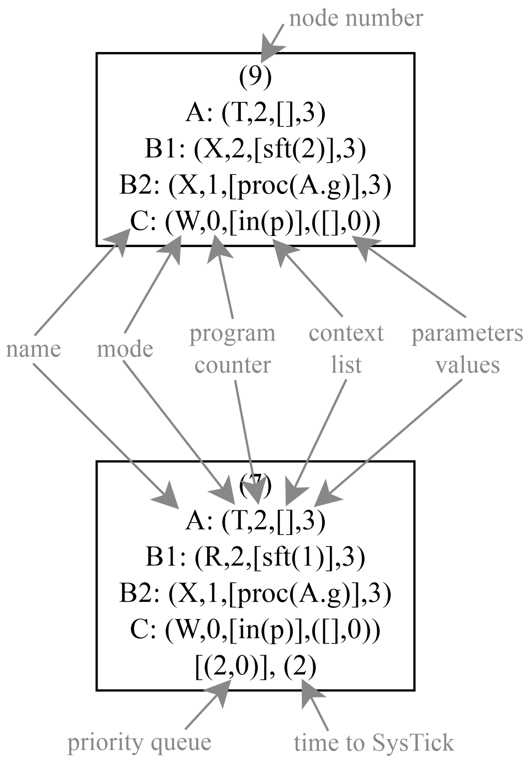


FIGURE 7. Elements of node description.

B. EXAMPLE 2

The LTS graphs presented in the previous subsection are small, so it is easy to calculate how many time-units the system needs to complete the required computations. The situation changes when the state space grows so large that we need the automatic methods to analyse an LTS. Let us consider modified versions of the model shown in Fig. 4. Suppose there are more active agents that perform the computation. The behaviour of all agents is very similar. Each agent collects an initial value from agent A (1 time-unit), performs its function (3 time-units), and saves the result to agent C (1 time-unit).

Let us focus on a single time property – the maximal time necessary to complete all computations. We analysed five models with 3, 4, 5, 6, and 7 active agents. Each model was compiled using the default α^0 system layer and the presented α^1_{FPPS} system layer. The results of the analysis are given in Table 3.

TABLE 3. Maximal time time necessary to complete all computations.

	α^0		α^1_{FPPS}	
No of agents	**Time**	**No of nodes**	**Time**	**No of nodes**
3	12	118	34	54
4	14	529	46	129
5	16	2716	58	432
6	18	16489	70	2007
7	20	108718	82	11802

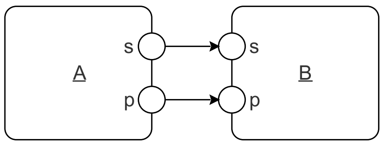
We can further modify the model under consideration. Suppose that each active agent runs in an infinite loop, and the computation (collect, call function, save) is a single run of such a loop. In the Alvis language, we can use the *loop every* statement to define a loop that is run periodically every given number of time-units. Thus, the above analyses allow us to assess whether the loop resumption time is defined correctly. In other words, we may check if the system manages to perform all calculations before the loop is scheduled to resume.

C. EXAMPLE 3

Let us focus on the model shown in Fig. 8. There are two active agents that perform their calculations in infinite loops. The *null* statements are used to model the calculations performed by agent A. Communication between the agents is used to synchronise their behaviour. It is assumed that the execution time of each statement takes one time-unit. It should be noted that agent A uses non-blocking communication on port s. When a non-blocking communication is used, the agent that initiates the communication may abandon it when the second side is not ready to finalise it. In this case, agent A waits at most two time-units for finalisation.

The Labelled Transition Systems generated for the α^0 and α^1_{FPPS} system layers are shown in Fig. 9 – left and right figures, respectively. In the first case, we have a system in which five states are cyclically repeated. The system is deadlock-free, the initial state is reversible, and each of these five states is a home state (it is reachable from any reachable state).

The LTS generated for α^1_{FPPS} system layer consists of one path containing 13 states. As a result of the lack of interoperability on agent B's side, synchronisation on port s is abandoned, leading to a deadlock represented by state 12 – agent



```
agent A {                  -- stat. no,   duration
  loop {                   -- 1           1
    null;                  -- 2           1
    out (2) s;             -- 3           1
    null;                  -- 4           1
    out p;                 -- 5           1
  }
}

agent B {
  loop {                   -- 1           1
    in s;                  -- 2           1
    in p;                  -- 3           1
  }
}
```

FIGURE 8. Data processing with synchronization points.

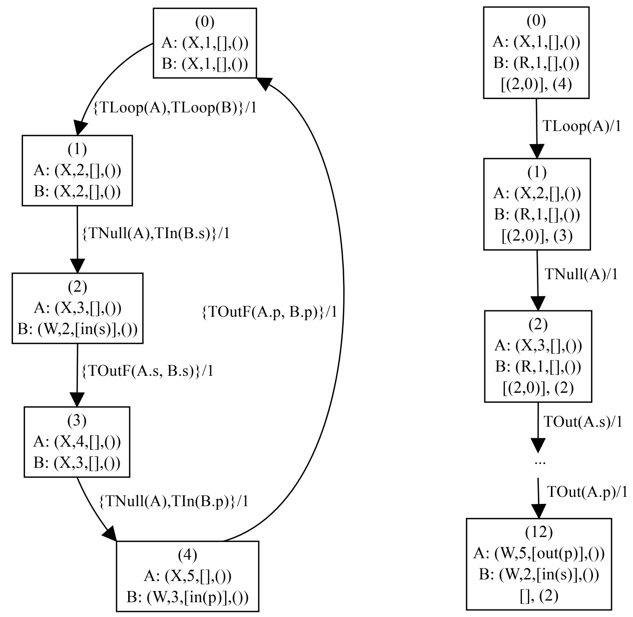

(0) A: (X,1,[],()) B: (X,1,[],())
{TLoop(A),TLoop(B)}/1
(1) A: (X,2,[],()) B: (X,2,[],())
{TNull(A),TIn(B.s)}/1
(2) A: (X,3,[],()) B: (W,2,[in(s)],())
{TOutF(A.s, B.s)}/1
(3) A: (X,4,[],()) B: (X,3,[],())
{TNull(A),TIn(B.p)}/1
(4) A: (X,5,[],()) B: (W,3,[in(p)],())
{TOutF(A.p, B.p)}/1
(0) A: (X,1,[],()) B: (R,1,[],()) [(2,0)], (4)
TLoop(A)/1
(1) A: (X,2,[],()) B: (R,1,[],()) [(2,0)], (3)
TNull(A)/1
(2) A: (X,3,[],()) B: (R,1,[],()) [(2,0)], (2)
TOut(A.s)/1
...
TOut(A.p)/1
(12) A: (W,5,[out(p)],()) B: (W,2,[in(s)],()) [], (2)

FIGURE 9. LTS - parallel version (left), single processor version (right).

A is waiting for the finalisation of the communication on port *p*, while *B* is waiting for the finalisation of the communication on port *s*. In this case, the model does not have any of the properties mentioned above.

VII. SUMMARY AND FUTURE WORK

The Alvis language has been developed and efficiently used for modelling and formal verification of a wide range of parallel discrete systems. However, the convention used in the implementation and in all research papers on this topic up to now, has certain limitations which may cause problems in the formal analysis. The system layer α^0 supposes an unlimited number of available processors to carry out the system activities and hence full physical parallelism of the processes which are logically parallel in a modelled system. It is not always adequate, as far as in real life systems the number of processors is often smaller than the number of agents. The single-processor systems are still widespread. One of the problems related to the formal analysis here is that a system with the assumption of unlimited parallelism may meet some properties, but the system having a single processor and executing parallel activities in an interleaving mode may fail to meet them. It is possible that a formal analysis of such a system using α^0 layer may be unable to detect that. Another problem is, the unlimited parallelism easily leads to a state explosion.

The paper presents an α^1 system layer providing the possibility to verify systems and to generate state spaces for single processor platforms. A formal description of the layer is presented, together with a realistic algorithm of preemptive scheduling. The experiments and case studies demonstrate that the presented approach allows to verify time dependencies for single processor systems and, when there are different priorities of the simultaneously acting agents, to avoid or to significantly reduce the state explosion.

As a future work, developing of α^n system layer is considered, in which a multiprocessor platform with a given limited number of processors is supposed. This requires, among others, developing and implementation of a proper scheduling algorithm and an algorithm of LTS generation. Such a layer is going to be, in a sense, the most universal one, applicable to modelling and formal verification of a wide range of real life systems.

In addition, if the application of the α^1 layer will demonstrate that it is necessary, other scheduling algorithms for α^1 can be considered, different from the fixed priority preemptive scheduling described in this paper.

REFERENCES

[1] H. Kopetz, *Real-Time Systems. Design Principles for Distributed Embedded Applications*. New York, NY, USA: Springer, 2011.

[2] I. Bicchierai, G. Bucci, L. Carnevali, and E. Vicario, "Combining UML-MARTE and preemptive time Petri nets: An industrial case study," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 1806–1818, Nov. 2013.

[3] M. Szpyrka, P. Matyasik, and R. Mrówka, "Alvis—Modelling language for concurrent systems," in *Intelligent Decision Systems in Large-Scale Distributed Environments* (Studies in Computational Intelligence), P. Bouvry, H. Gonzalez-Velez, and J. Kołodziej, Eds. Berlin, Germany: Springer-Verlag, vol. 362, 2011, ch. 15, pp. 315–341.

[4] M. Szpyrka, M. Wypych, J. Biernacki, and L. Podolski, "Discrete-time systems modeling and verification with Alvis language and tools," *IEEE Access*, vol. 6, pp. 78766–78779, 2018.

[5] M. Szpyrka, P. Matyasik, J. Biernacki, A. Biernacka, M. Wypych, and L. Kotulski, "Hierarchical communication diagrams," *Comput. Informat.*, vol. 35, no. 1, pp. 55–83, 2016.

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

[7] M. Szpyrka, J. Biernacki, and A. Biernacka, "Tools and methods for RTCP-nets modeling and verification," *Arch. Control Sci.*, vol. 26, no. 3, pp. 339–365, Sep. 2016.

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

[9] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets*, vol. 3098. Berlin, Germany: Springer, 2004.

[10] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured Petri nets and CPN tools for modelling and validation of concurrent systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, nos. 3–4, pp. 213–254, 2007.

[11] V. Gehlot and C. Nigro, "An introduction to systems modeling and simulation with colored Petri nets," in *Proc. Winter Simulation Conf.*, Dec. 2010, pp. 104–118.

[12] M. Westergaard, "CPN tools 4: Multi-formalism and extensibility," in *Application and Theory of Petri Nets and Concurrency*, J.-M. Colom and J. Desel, Eds. Berlin, Germany: Springer, 2013, pp. 400–409.

[13] R. David, "Petri nets and Grafcet for specification of logic controllers," *IFAC Proc. Volumes*, vol. 26, no. 2, pp. 683–688, 1993.

[14] Y. Qamsane, A. Tajer, and A. Philippot, "A synthesis approach to distributed supervisory control design for manufacturing systems with Grafcet implementation," *Int. J. Prod. Res.*, vol. 55, no. 15, pp. 4283–4303, 2017, doi: 10.1080/00207543.2016.1235804.

[15] M. Adamski and M. Chodań, *Modelling the Discrete Control Devices with the SFC Nets*. Zielona Góra, Poland: Wydawnictwo Politechniki Zielonogórskiej, 2000.

[16] (2006). *SFC for SIMATIC S7*. [Online]. Available: https://cache.industry.siemens.com/dl/files/748/24451748/att_96723/v1/s7sfcs7b_e.pdf

[17] *Series 90 Sequential Function Chart Programming Language*, GE Fanuc Automat., Charlottesville, VA, USA, 1994.

[18] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, J. Ouaknine, and J. Worrell, *Model Checking Real-Time Systems*. Cham, Switzerland: Springer, 2018, pp. 1001–1046, doi: 10.1007/978-3-319-10575-8_29.

[19] L. S. Kamireddy, "Real-time systems modeling and analysis," 2018, *arXiv:1811.10083*.

[20] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[21] H. Giese and S. Burmester, "Real-time statechart semantics," Universität Paderborn, Paderborn, Germany, Tech. Rep., tr-ri-03-239, 2003.

[22] B. Lee and E. A. Lee, "Hierarchical concurrent finite state machines in Ptolemy," in *Proc. Int. Conf. Appl. Concurrency Syst. Design*, 1998, pp. 34–40.

[23] F. Vahid and T. D. Givadrdis, *Embedded System Design: A Unified Hardware/Software Introduction*. Hoboken, NJ, USA: Wiley, 2002.

[24] C. Ptolemaeus. (2014). *System Design, Modeling, and Simulation using Ptolemy II*. [Online]. Available: http://ptolemy.org/books/Systems

[25] F. Vahid, S. Narayan, and D. D. Gajski, "SpecCharts: A VHDL front-end for embedded systems," *IEEE Trans. Comput.-Aided Design Integr.*, vol. 14, no. 6, pp. 694–706, Jun. 1995.

[26] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SPECC: Specification Language and Methodology*. Boston, MA, USA: Springer, 2000.

[27] W. Wonham, K. Cai, and K. Rudie, "Supervisory control of discrete-event systems: A brief history—1980–2015," in *Proc. IFAC World Congr.*, Toulouse, France, 2017, pp. 1791–1797.

[28] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Boston, MA, USA: Springer, 2021.

[29] P. C. Y. Chen and W. M. Wonham, "Real-time supervisory control of a processor for non-preemptive execution of periodic tasks," *Real-Time Syst.*, vol. 23, no. 3, pp. 183–208, 2002.

[30] R. Devaraj, A. Sarkar, and S. Biswas, "Fault-tolerant scheduling of non-preemptive periodic tasks using SCT of timed DES on uniprocessor systems," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 9315–9320, Jul. 2017.

[31] R. Devaraj, A. Sarkar, and S. Biswas, "Optimal work-conserving scheduler synthesis for real-time sporadic tasks using supervisory control of timed discrete-event systems," *J. Scheduling*, vol. 24, no. 1, pp. 69–82, Feb. 2021.

[32] R. Milner, *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, 1989.

[33] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada 2005*. Cambridge, U.K.: Cambridge Univ. Press, 2007.

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

[35] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The NUXMV symbolic model checker," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 8559. Cham, Switzerland: Springer, 2014, pp. 334–342.

[36] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: A toolbox for the construction and analysis of distributed processes," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 2, pp. 89–107, Apr. 2013.

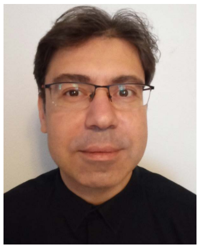
[37] M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski, "Formal description of Alvis language with α^0 system layer," *Fundamenta Informaticae*, vol. 129, nos. 1–2, pp. 161–176, 2014.

[38] A. Silberschatz, P. Gavlin, and G. Gagne, *Operating System Concepts*. Hoboken, NJ, USA: Wiley, 2009.

[39] M. Szpyrka, P. Matyasik, M. Wypych, J. Biernacki, and L. Podolski. (2017). *Alvis Modelling Language*. [Online]. Available: http://alvis.kis.agh.edu.pl

MARCIN SZPYRKA (Senior Member, IEEE) is a Full Professor with the Department of Applied Computer Science, AGH University of Science and Technology, Kraków, Poland. He is a Leader of the Alvis Project. He has authored over 140 publications in the domains of formal methods, software engineering, knowledge engineering, and data science. His research interests include the theory of concurrency, systems security, and functional programming.

JAROSŁAW BANIEWICZ received the Ph.D. degree from the Department of Applied Computer Science, AGH University of Science and Technology, Kraków, Poland. His research interests include the theory of concurrency, algorithms (including parallel algorithms), and the process of creating and managing the IT projects in Agile methodologies.



ANDREI KARATKEVICH is a Professor with the Department of Applied Computer Science, AGH University of Science and Technology, Kraków, Poland. He has authored over 100 publications in the domains of petri nets, control systems, and formal verification. His research interests include the theory of concurrency, algorithms (including parallel algorithms), and formal methods.