DOI: 10.15439/2016F264
ACSIS, Vol. 8. ISSN 2300-5963

Alvis models of safety critical systems state-base verification with nuXmv

Jerzy Biernacki
AGH University of Science and Technology
Department of Applied Computer Science
Al. Mickiewicza 30, 30-059 Krakow, Poland
E-mail: jbiernac@agh.edu.pl

***Abstract*—For modelling of real-time safety critical systems, when traditional testing techniques cannot be applied, formal system verification is crucial. Alvis is a modelling language that combines possibilities of formal models verification with flexibility and simplicity of practical programming languages. Solutions introduced in Alvis make the development process easier and help engineers to cope with more complex systems. The paper deals with a state-based approach to the verification of Alvis models. Until the research presented in the paper were conducted, the verification process was mostly action-based. The nuXmv tool, as one of the top model checkers, was selected for the task of state-base verification of Alvis models translated into the SMV modelling language. The paper presents a translation algorithm and usability studies performed on existing safety critical systems.**

I. Introduction

Alvis [1], [2] is a formal modelling language developed at AGH-UST in Kraków, Department of Applied Computer Science (*http://alvis.kis.agh.edu.pl*). The motivation behind its creation and development is to provide a formal language which could be used by an average software engineer to model and verify complex systems. To this end, Alvis combines advantages of high level programming languages with a visual modelling language for defining communication channels between subsystems. Its most significant advantage over classical formal methods (Petri nets [3], [4], timed automata [5], [6] and process algebras [7], [8], [9]) is an engineer-friendly syntax. The heavy mathematical foundations are hidden from the user without compromising the capabilities and expressive power of the formalism. Alvis, as a formal language, has an advantage over the industry programming languages – Alvis models can be formally verified using model checking techniques [10]. Using Alvis language, an average software engineer is able to model and verify complex systems which can be then easily implemented. This is particularly important in concurrent and distributed systems where traditional methods of software testing are not applicable. The formal verifications of such systems is a ground for many current scientific projects [11]. The ongoing research on the Alvis language include also building an Alvis simulator, Alvis Virtual Machine [12] and automatic Java code generation [13].

The nuXmv tool [14], [15] is currently one of the top-notch and mainstream model checkers for temporal logics. It features a prominent and state-of-the-art verification engine. The project is still supported and developed, new versions of the tool are released regularly.

The nuXmv can check whether a given finite state model satisfies a given temporal logic formula and if not, it can provide a proper counter-example. System requirements specification, in the form of a set of LTL [16] and CTL [16], [17] temporal logics formulae, can be therefore automatically verified by the tool. In addition, it has a dedicated modelling language called SMV [15] which is relatively easy to use for modelling the system. Furthermore, according to the authors of the project it can verify systems of high complexity, i.e. containing more than 10^{20} states.

These outstanding features initially made nuXmv the best possible choice for the task of state-base Alvis model verification. The main goal of the conducted research was to verify whether nuXmv can be effectively used in the process of verification of complex systems modelled in the Alvis language. In order to prove the concept, a translation algorithm was conceived and then implemented. Extensive experiments of the solution were performed, including modelling and verification of existing real-time safety critical systems.

The paper is organised as follows. Section II contains a short introduction to the Alvis language and basic information about the process of designing and verification of Alvis models. In Section III formal definitions concerning Alvis state space representation are provided. Section IV deals with the algorithm of Alvis model translation into nuXmv. Usability studies conducted on two examples of real-time safety critical systems are presented in Section V. A short summary is given in the final section.

II. Alvis modelling language

An Alvis model is basically a collection of subsystems called *agents* that may run concurrently, communicate with each other, compete for shared resources etc. The concept of *agent* is borrowed from CCS [8], [18]. Agents are divided into *active* and *passive* ones and mimic, to some degree, tasks and protected objects in the Ada programming language [19].

Active agents may perform some activities and are treated as threads of control in a concurrent system. *Passive agents* provide a mechanism for the mutual exclusion and data synchronization.

Interconnections between agents are defined on *communication diagrams*, a visual part of the Alvis language. These diagrams present agents as nodes and communication channels as arcs in a directed graph. To model the behaviour of the agents, the *code layer* is used. Alvis source code is similar to the one of high level programming languages. Alvis statements may also incorporate elements of the Haskell functional programming language [20].

Furthermore, the complex systems may be modelled using hierarchical communication diagrams [21]. They introduce a concept of a hierarchical node which represents a subsystem defined at the lower level. Therefore, it allows to describe a system on many different levels of abstraction. A summary of Alvis graphical elements and code statements is presented in Fig. 1.

Alvis models are designed using an Alvis design toolkit including *Alvis Editor*, *Alvis Simulator* and *Alvis Compiler* tools. *Alvis Editor* is a visual modelling environment featuring design of Alvis models. *Alvis Simulator* enables step-by-step simulations of the models. *Alvis Compiler* [22] translates designed models into Haskell program. The Haskell middle-stage representation is used to generate LTS graphs (*labelled transition system* [23]) of the Alvis models. LTS graphs will be explained in more detail in the next section. They can be used to formally verify models using model checking techniques. LTS graphs are checked in terms of satisfaction of model properties described as temporal logic formulae. The original verification process included only action-based verification with μ-calculus [24] in CADP toolbox [25]. The approach presented in this paper employs nuXmv tool to allow the usage of LTL and CTL temporal logics. The modelling and verification process of Alvis models is presented in Fig. 2.

More details on this topic may be found in the manual at the website of the Alvis project.

III. Alvis state space representation

Before an Alvis LTS to nuXmv translation algorithm can be introduced, some of the key concepts regarding Alvis state space must be defined.

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

$$S(X) = (am(X), pc(X), ci(X), pv(X)),$$

where $am(X)$ is an agent mode, $pc(X)$ is a program counter, $ci(X)$ is a context information list, and $pv(X)$ is parameters values.

Each agent state can be described unambiguously with information contained by this four-tuple. Where necessary, to every one of am, pc, ci and pv symbols, there can be a state index added, e.g. pv_{S_i}, to indicate which state it refers to.

The *agent mode* can take one of the five following values: *Finished* (F), *Init* (I), *Running* (R) and *Taken* (T). *Finished* means that an agent has finished its work. *Init* is the default mode for agents that are inactive in the initial state. *Running* means that an agent is performing one of its statements. *Taken* means that one of the passive agent's procedures has been called and the agent is executing it. *Waiting*, for passive agents, means that the corresponding agent is inactive and waits for another agent to call one of its accessible procedures. For active agents, this mode means that the corresponding agent is waiting either for a communication with another active agent, or for a currently inaccessible procedure of a passive one.

The *program counter* points at the current statement of an agent i.e. the next statement to be executed or the statement that has been already executed by an agent but needs a feedback from another agent to be completed (e.g. a communication between agents).

The *context information list* contains additional information about the current state of an agent e.g. if an agent is in the waiting mode, ci contains information about events the agent is waiting for.

The *parameters values list* contains the current values of the agent's parameters.

Definition 2. A state of a model $\boldsymbol{A} = (D, B, \alpha^0)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ and $\mathcal{A} = \{X_1, ..., X_n\}$ is a tuple $S = (S(X_1), ..., S(X_n))$.

The concept of an Alvis model state is explained in Fig. 3.

Execution of any step is expressed as a transition between formally defined states of an Alvis model. States of a model and transitions among them are represented using a labelled transition system (LTS graph).

Definition 3. A Labelled Transition System is a tuple:

$$LTS = (S, A, \rightarrow, s_0),$$

where S is a set of states, A is a set of actions, $\rightarrow \subseteq S \times A \times S$ is the transition relation and s_0 is an initial state.

For an Alvis model, an LTS is a four-tuple:

$$LTS = (\mathcal{R}(S_0), \mathcal{T}, \rightarrow, S_0),$$

where $\mathcal{R}(S_0)$ is a set of states reachable from the initial state, $\mathcal{T}$ is a set of all possible steps for a given model, $\rightarrow = \{(S, t, S') \colon S - t \rightarrow S' \wedge S, S' \in \mathcal{R}(S_0)\}$, where $t \in \mathcal{T}$, and S_0 is an initial state. In untimed Alvis models arcs are labelled with names of individual steps performed by agents. In the timed models arcs are labelled with the sets of parallel steps.

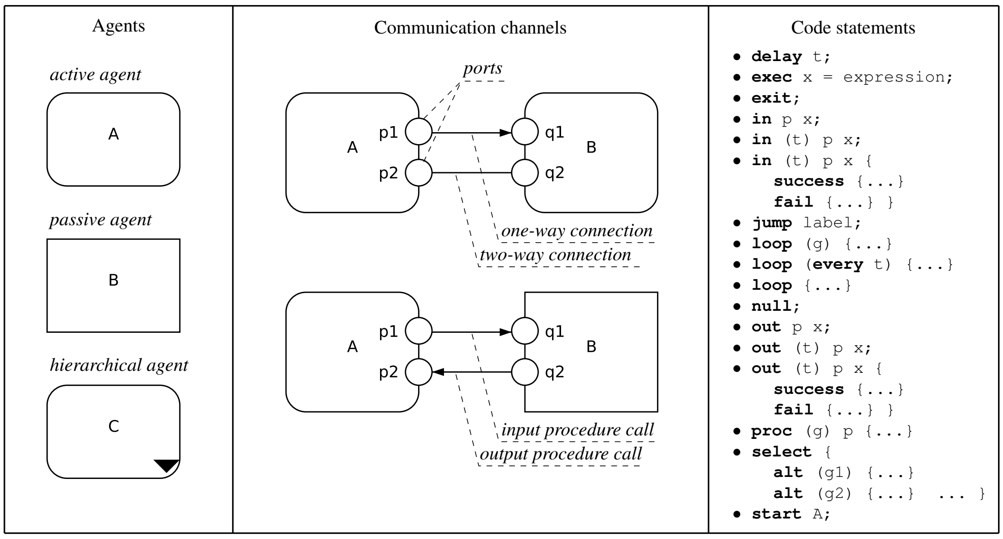
In order to describe the translation algorithm in the next section, a few additional terms need to be introduced:

- $B(X)$ – Agent X dynamics definition (code);
- $card(B(X))$ – number of steps in $B(X)$;
- $\mathcal{N}(t)$ – a name of the t transition.

NuXmv models are basically *finite state transition systems* [15] which can be defined as *Kripke structures* [26].

Definition 4. A *finite state transition system* is a tuple $TS = (S, I, \rightarrow, L)$, where:

- S is a finite set of *states*,
- $I \subseteq S$ is the set of *initial states*,
- $\rightarrow \subseteq S \times S$ is the *transition relation*, specifying the possible transitions from state to state,
- L is the *labelling function* that labels states with *atomic propositions* that hold for the given state.



```
• delay t;
• exec x = expression;
• exit;
• in p x;
• in (t) p x;
• in (t) p x {
    success {...}
    fail {...} }
• jump label;
• loop (g) {...}
• loop (every t) {...}
• loop {...}
• null;
• out p x;
• out (t) p x;
• out (t) p x {
    success {...}
    fail {...} }
• proc (g) p {...}
• select {
    alt (g1) {...}
    alt (g2) {...}  ... }
• start A;
```

Figure 1: Elements of Alvis language.

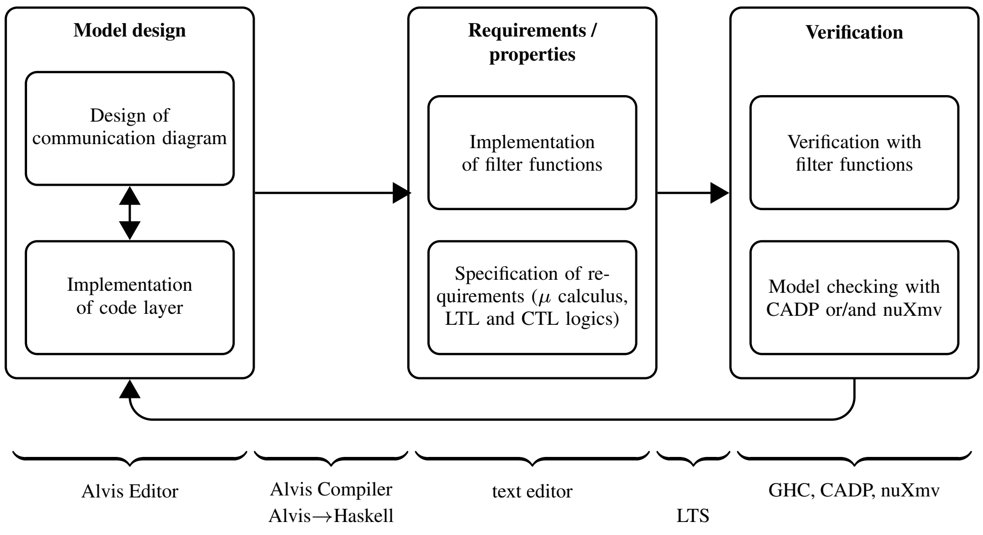

Model design
Design of communication diagram
Implementation of code layer
Requirements / properties
Implementation of filter functions
Specification of requirements (μ calculus, LTL and CTL logics)
Verification
Verification with filter functions
Model checking with CADP or/and nuXmv
Alvis Editor
Alvis Compiler
Alvis→Haskell
text editor
LTS
GHC, CADP, nuXmv

Figure 2: Alvis modelling and verification process.

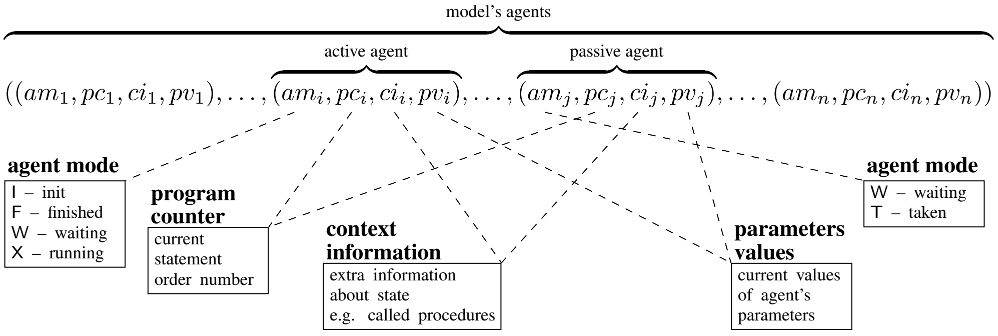

model's agents
active agent
passive agent
$((am_1, pc_1, ci_1, pv_1), \ldots, (am_i, pc_i, ci_i, pv_i), \ldots, (am_j, pc_j, ci_j, pv_j), \ldots, (am_n, pc_n, ci_n, pv_n))$
agent mode
I – init
F – finished
W – waiting
X – running
program counter
current statement order number
context information
extra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

Figure 3: Representation of an Alvis model state.

IV. Alvis LTS to nuXmv translation algorithm

Finite state transition systems in nuXmv tool are modelled with a dedicated modelling language called SMV [15]. In the presented approach, a nuXmv model after translation consists of three main parts: Variables definitions (`VAR` and `IVAR`), `ASSIGN` section and specification of transitions' availability

(TRANS). The first one of them, the IVAR section, contains definition of an input variable action. It is used to contain transitions' labels.

The VAR section is used to contain definitions of standard variables. These include set of states and atomic propositions variables. The ASSIGN section is composed of three main parts. The first one initializes the state variable, the second is responsible for defining transitions between the states and the final one assigns values to the atomic propositions for specific states. The set of atomic propositions is given implicitly using variables and their domains. The last main part, the TRANS section, specifies for every state which actions are available in it. For instance, line `TRANS s = s1 -> (action = a1) | (action = a2)` determines that when the system is in state s1, the only available actions are a1 and a2.

A translation algorithm of an Alvis model LTS into the nuXmv code is presented in Fig. 1. In the adopted notation, ▷ symbol indicates generated nuXmv code, # represents string concatenation, $Label()$ produces a variable name and $Type()$ returns a variable's data type.

The first part of the algorithm is responsible for generation of a declaration of the input variable action. In the proposed approach, this variable is used to define names of the transitions between the model states as edges' labels in the nuXmv transition system. It enables using transition names in LTL formulae during the verification process. During the model simulation this additional information is a major enhancement that allows to analyse not only the state changes, but also the steps that led to them. The *action* set is initialised with a single element representing empty action named *NOP*. Every transition label is added to the set then.

In the next step, the domain of the state variable s is defined. Its value denotes the current state of an Alvis model. For every reachable state ($S \in \mathcal{R}(S_0)$), the state name is added to the set of nuXmv states.

Lines 13–41 contain steps required to define variables representing elements of the model's state. This section starts with L_{am}, L_{pc}, L_{ci} and L_{pv} sets initialisation. They represent the sets of defined variable labels for agent mode, program counter, context information list, and parameters values correspondingly. Every agent has exactly one am and pc variable, while possibly multiple ci and pv variables.

For every agent of the model, variable labels are generated by concatenating agent's name with a two letter abbreviation indicating the element of the agent state's tuple it refers to. Agent mode variable is an enumeration and can have assigned one of the x, w, f, i and t values. Program counter variable is a bounded integer, ranging from 0 to $card(B(X_i))$, where X_i is the given agent. Context information variable labels, in addition to the basic naming convention, contain also information about the possible values of the original ci. They are booleans and the TRUE value implies that the given ci entry is present in the context list in the given state. Parameters values labels are generated by concatenating agent name, pv keyword, pv order number. If pv variable's type in the code layer is integer or boolean, the nuXmv variable is of the same type. In other case pv value is appended to the label and the type of the nuXmv variable is boolean. TRUE means that the given agents parameter is of the value specified as the last part of nuXmv variable label in the given state.

The next part of the algorithm starts with adding of the beginning of the ASSIGN section and initialization of the s variable with the name of the initial state. Then the transition relation switch statement is opened (line 44). $successors_{ik}$ is the set of successors of the si state, reachable through transition t_k. In the nested loops that follow, successors lists for every reachable state are generated.

The next section of the algorithm contains five similar blocks of pseudocode (lines 57–63, 64–70, 71–80, 81–95, 96–104). Each generates labelling functions for the agent state variables defined in the VAR section. Labelling functions are basically switch statements in which the proper value is assigned to the variable depending on the current state of the system. pv variables labelling functions are divided into two separate loops, depending on the type of the variable in the code layer.

The last part of the algorithm (lines 105–114) defines availability of the transitions. It is determined by the value of the si variable. TRANS line is generated for every reachable state. It contains a list of available transitions. NOP action is not listed there because it is only available when no successor exists. This situation indicates a terminal state of the system.

Algorithm 1 Alvis LTS to nuXmv translation algorithm.

```
 1: ▷ MODULE main
 2: ▷ IVAR
 3: $action \leftarrow \{NOP\}$
 4: for all $t_i \in T$ do
 5:     $action \leftarrow action \cup \{Label(\mathcal{N}(t_i))\}$
 6: end for
 7: ▷ action : {NOP,N(t0), N(t1), ...};
 8: ▷ VAR
 9: for all $S_i \in \mathcal{R}(S_0)$ do
10:     $s \leftarrow s \cup \{Label(s_i)\}$
11: end for
12: ▷ s : {s0, s1, ...};
13: $L_{am} \leftarrow \emptyset$
14: $L_{pc} \leftarrow \emptyset$
15: $L_{ci} \leftarrow \emptyset$
16: $L_{pv} \leftarrow \emptyset$
17: for all $X_i \in \mathcal{A}$ do
18:     $l_{am} \leftarrow Label(X_i\#\mathtt{am})$
19:     ▷ X_i#am : {x, w, f, i, t};
20:     $L_{am} \leftarrow L_{am} \cup \{l_{am}\}$
21:     $l_{pc} \leftarrow Label(X_i\#\mathtt{pc})$
22:     $k \leftarrow card(B(X_i))$
23:     ▷ X_i#pc : 0..k;
24:     $L_{pc} \leftarrow L_{pc} \cup \{l_{pc}\}$
25:     for all $S_j \in \mathcal{R}(S_0)$ do
26:         $l_{ci} \leftarrow Label(X_i\#\mathtt{ci}\#ci_{S_j}(X_i))$
27:         ▷ l_ci : boolean;
```

```
        $L_{ci} \leftarrow L_{ci} \cup \{l_{ci}\}$
    end for
    for all $pv_j \in pv(X_i)$ do
        $type \leftarrow Type(pv_j)$
        if $type = Integer \vee type = Boolean$ then
            $l_{pv} \leftarrow Label(X_i\#\mathtt{pv}\#j)$
            $\triangleright$ l_pv : type;
        else
            $l_{pv} \leftarrow Label(X_i\#\mathtt{pv}\#j\#pv(X_i))$
            $\triangleright$ l_pv : boolean;
        end if
        $L_{pv} \leftarrow L_{pv} \cup \{l_{pv}\}$
    end for
end for
$\triangleright$ ASSIGN
$\triangleright$ init(s) = s0;
$\triangleright$ next(s) := case
for all $si \in s$ do
    for all $t_k \in \mathcal{T}$ do
        $successors_{ik} \leftarrow \emptyset$
        for all $sj \in s$ do
            if $S_i \xrightarrow{t_k} S_j$ then
                $successors_{ik} \leftarrow successors_{ik} \cup \{sj\}$
            end if
        end for
        $\triangleright$ s = si & action = t_k: {$successors_{ik}$};
    end for
end for
$\triangleright$ esac;
for all $l_{am} \in L_{am}; (l_{am} = Label(X_i\#\mathtt{am}))$ do
    $\triangleright$ l_am := case
    for all $sj \in s$ do
        $\triangleright$ s = sj : am_sj(X_i);
    end for
    $\triangleright$ esac;
end for
for all $l_{pc} \in L_{pc}; (l_{pc} = Label(X_i\#\mathtt{pc}))$ do
    $\triangleright$ l_pc := case
    for all $sj \in s$ do
        $\triangleright$ s = sj : pc_sj(X_i);
    end for
    $\triangleright$ esac;
end for
for all $l_{ci} \in L_{ci}; (l_{ci} = Label(X_i\#\mathtt{ci}\#ci_{S_j}(X_i))$ do
    $\triangleright$ l_ci := case
    for all $sk \in s$ do
        if $ci_{s_k}(X_i) = ci_{S_j}(X_i))$ then
            $\triangleright$ s = sk : TRUE;
        end if
    end for
    $\triangleright$ TRUE: FALSE;
    $\triangleright$ esac;
end for
for all $l_{pv} \in L_{pv} : Type(l_{pv}) = Integer \vee Type(l_{pv}) = Boolean; (l_{pv} = Label(X_i\#\mathtt{pv}\#j))$ do
    $\triangleright$ l_pv := case
    for all $sk \in s$ do
        $value \leftarrow pv_{s_k}(X_i)$
        if $value \neq 0 \wedge value \neq FALSE$ then
            $\triangleright$ s = sk : value;
        end if
    end for
    if $Type(l_{pv}) = Integer$ then
        $\triangleright$ TRUE: 0;
    else
        $\triangleright$ TRUE: FALSE;
    end if
    $\triangleright$ esac;
end for
for all $l_{pv} \in L_{pv} : Type(l_{pv}) \neq Integer \wedge Type(l_{pv}) \neq Boolean; (l_{pv} = Label(X_i\#\mathtt{pv}\#j\#pv(X_i)))$ do
    $\triangleright$ l_pv := case
    for all $sk \in s$ do
        $value \leftarrow pv_{s_k}(X_i)$
        $\triangleright$ s = sk : value;
    end for
    $\triangleright$ TRUE: FALSE;
    $\triangleright$ esac;
end for
for all $si \in s$ do
    $T_i \leftarrow \emptyset$
    for all $t_k \in \mathcal{T}$ do
        if $\exists_{S_j} S_i \xrightarrow{t_k} S_j$ then
            $T_i \leftarrow T_i \cup \{t_k\}$
        end if
        $\triangleright$ TRANS s = si ->
        (action = Ti_0|action = Ti_1 | ... );
    end for
end for
```

The main purpose of including the above algorithm in this paper is to convey the basic concept behind the translation. Therefore, as one may notice, the above algorithm is not optimal. Nonetheless, after many optimizations and enhancements, this algorithm was implemented as an additional module to the PetriNet2ModelChecker tool. This module enables automatic conversion of an LTS graph of an Alvis model stored in .dot file into nuXmv code, and therefore allows to verify any Alvis model using LTL and CTL logics in one of the top model checkers available.

V. Usability studies

The presented approach was tested against models of existing safety-critical systems. Among them, the tests were conducted on railway switch system and fire alarm contol panel. The former is the solution manufactured by Grupa ZUE S.A. [27] and employed in public transport in Krakow, Szczecin and Wroclaw [28]. The latter is a project of the SITP organization [29]. More information on this system can be found in [30]. The approach presented in this paper will be illustrated on the first one of them. A schematic of the system is shown in Fig. 4.

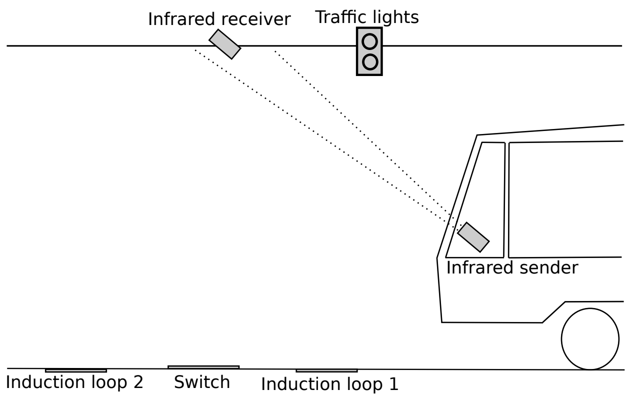


Figure 4: Railway switch system.

In every tram, there is an NP03 infra-red transmitter used to send IR signals to the switch control system. It is located on a tram driver console panel. An OP03 receiver is responsible for collecting infra-red signals and sending them to system driver. It is usually installed on overhead lines or on special poles placed before the switch. Traffic lights are providing a motorman with two pieces of information, i.e. the current direction of the tracks and a status of a switch blades lock.

A motorman's responsibility is to ensure that switch blades are set in the right direction and locked before he can drive a tram through a switch. He can change blade's direction using an NP03 transmitter. On receiving a signal, an IR receiver sends a pulse to a switch motor controller, which in turn sends a signal to blades controller to change direction of the rails. The change is possible only in the operating range of the receiver. Tram driver has limited time for choosing the expected direction, depending on the speed of the tram. If the motorman does not change the direction while the tram is in a reach zone of the IR receiver, he would have to stop the tram and manually change the direction using a special lever.

In a switch zone, there are also two induction loops installed, one before and one after switch blades. They are responsible for detecting a tram entering and leaving the crossing zone. When a tram is detected, an electrical switch lock mechanism is locking blades in the current position, in order for tram to pass safely through the switch zone. They are unlocked immediately after the tram leaves the crossing.

An Alvis model of this system was constructed. Its communication diagram is presented in Fig. 5.

Using the implementation of the algorithm presented in Section IV the system model was automatically translated into a nuXmv source file. The nuXmv representation maintains every piece of the original information about system behaviour stored in the LTS. The next step involves verification of system properties. In the presented approach they are described as LTL and CTL formulae. Three examples of such formulae are given in Listing 1.

Listing 1: Examples of LTL and CTL formulae for the railway switch system model.

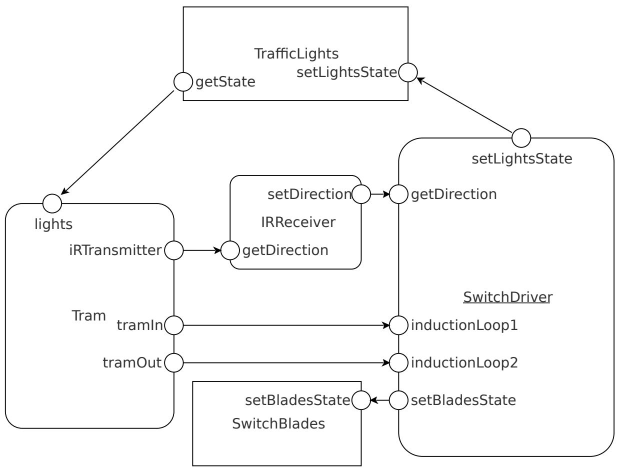


Figure 5: Railway switch system communication diagram.

```
--1) TramPassed = false U BladesLocked = true
LTLSPEC (SwitchDriver#pv3 < 2) U (SwitchBlades#pv1 <
  ↪  0)

--2) EF TramPassed = true
CTLSPEC EF (SwitchDriver#pv3 = 2)

--3) F TramPassed = true
LTLSPEC F (SwitchDriver#pv3 = 2)
```

`SwitchDriver#pv3` denotes a `tram` variable from the *SwitchDriver* agent. It can have three values: 0 when the tram is before the first induction loop, 1 when tram passed the first induction loop and 2 when the tram passed the crossing (the second induction loop). `SwitchBlades#pv1` is a `bladesState` variable. When its value is positive, the switch blades are not locked and when the value is negative, the blades are locked.

The first formula verifies whether the switch is being locked before the tram passes the crossing. This formula is crucial for the safety of railway switch mechanism. The nuXmv confirmed that it is satisfied in the model. The second formula checks whether there is a path in which a tram passes through the crossing. It is also true. The last one is similar to the previous, except it checks whether a tram always finally passes through the crossing. This one is not satisfied. The true potential of nuXmv is in providing the counterexample. If a formula is not true, nuXmv provides a path that proves it. In this case the tram does not pass through the crossing if the driver did not manage to send the signal to change the blades direction when the tram was in the zone of the infra-red receiver. Driver has to stop the tram, step out of it and manually change the direction. It is a desired behaviour of the system.

Three versions of the same model were prepared with varying complexity, each describing the system on a different level of abstraction. For each one of them, the same set of properties was verified. They were categorized into three groups: reachability, safeness and liveness properties. For each group an average verification time was measured and presented in Table I. Tests were performed on a PC with AMD Phenom II X6 processor and 16 GB of RAM. The verification times are growing fast with complexity but even for the most complex railway switch system version, they are quite acceptable.

For the comparison, translation to nuXmv was performed also on the fire alarm control panel system. Its communication diagram is presented in Fig. 6. As this system is significantly more complex, the amount of states in the LTS is adequately larger. Although the translation itself was fast, nuXmv couldn't handle loading of such a complex system on the testing machine. The amount of memory needed to load the model exceeded available resources (RAM and swap space) and the nuXmv process was killed by the operating system. Therefore, the average verification times are not provided.

The results of the tests confirm that the presented approach is performing well for models of medium complexity. More complex ones require a lot of resources, especially RAM. Provided enough RAM or swap space, the approach can be applied to most models.

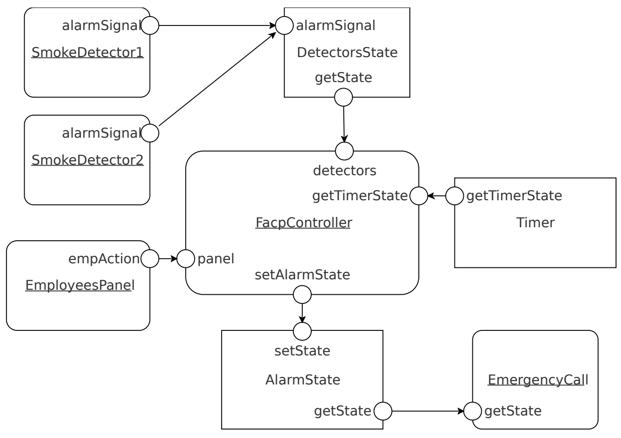


Figure 6: Fire alarm control panel communication diagram.

VI. Summary

The paper introduces state-base approach to verification of Alvis models. The presented solution enables to automatically verify properties of the modelled system using the mainstream model checking tool nuXmv. It employs formulated and implemented algorithm of Alvis LTS translation into nuXmv source code.

The usability studies of the concept were conducted on models of actual real-time safety critical systems, railway switch system and fire alarm control panel. Illustrative properties of these systems have been specified as LTL and CTL formulae and verified with nuXmv to demonstrate the capabilities and limitations of the approach. The results of the tests have been presented and summarised.

The proposed verification method proved to be handling most middle-sized models with ease, even on a regular PC. Although nuXmv supposed capabilities exceed the current needs, the solution is limited by the amount of RAM available. Complex systems require a great deal of memory to load. Compared to the amount of memory required to verify systems using action-based approach in CADP, nuXmv seems ineffective. On the other hand, state-based solution allows much more thorough verification because of the information stored about the states. The combination of both state- and action-based verification is currently the most optimal option.

Future work on state-based verification of Alvis models will focus on other possibilities. The most promising is a concept of a dedicated query language operating on the middle-stage Haskell representation.

References

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

[2] M. Szpyrka, P. Matyasik, R. Mrówka, and L. Kotulski, "Formal description of Alvis language with α^0 system layer," *Fundamenta Informaticae*, vol. 129, no. 1-2, pp. 161–176, 2014. doi: 10.3233/FI-2014-967

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

[4] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

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

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

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

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

[9] T. Bolognesi and E. Brinksma, "Introduction to the iso specification language lotos," *Comput. Netw. ISDN Syst.*, vol. 14, no. 1, pp. 25–59, Mar. 1987. doi: 10.1016/0169-7552(87)90085-7. [Online]. Available: http://dx.doi.org/10.1016/0169-7552(87)90085-7

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

[11] I. Grobelna, R. Wisniewski, M. Grobelny, and M. Wisniewska, "Design and verification of real-life processes with application of petri nets," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. PP, no. 99, pp. 1–14, 2016. doi: 10.1109/TSMC.2016.2531673

[12] P. Matyasik, "Alvis virtual machine," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, Sept 2014. doi: 10.15439/2014F267 pp. 1639–1645.

[13] P. Matyasik, M. Szpyrka, and M. Wypych, "Generation of Java code from Alvis model," in *International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2015)*, ser. AIP Conference Proceedings, vol. 1702. Athens, Greece: AIP Publishing, March 20-23 2015. doi: 10.1063/1.4938890 pp. 100 013–1–100 013–4.

[14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 334–342.

[15] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[16] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[17] E. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier Science, 1990, vol. B, pp. 995–1072.

[18] M. Szpyrka and P. Matyasik, "Formal modelling and verification of concurrent systems with XCCS," in *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008)*, Krakow, Poland, July 1-5 2008, pp. 454–458.

[19] A. Burns and A. Wellings, *Concurrent and real-time programming in Ada 2005*. Cambridge University Press, 2007.

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

Table I: Performance tests for Alvis models verification in nuXmv.

Model	**Number of states**	**RAM usage [GB]**	**Avarage verification time [s]**		
			Reachability properties	**Safeness properties**	**Liveness properties**
Basic railway switch system	1221	0.17	0.91	1.71	1.32
Detailed railway switch system	3005	0.83	4.01	4.08	4.81
Complex railway switch system	6833	3.6	25.13	46.31	41.89
Fire alarm control panel	43624	>20	-	-	-

[21] M. Szpyrka, P. Matyasik, J. Biernacki, A. Biernacka, M. Wypych, and L. Kotulski, "Hierarchical communication diagrams," *Computing and Informatics*, vol. 35, pp. 55–83, 2016.

[22] M. Wypych, M. Szpyrka, and P. Matyasik, "Extension of Alvis compiler front-end," in *International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2015)*, ser. AIP Conference Proceedings, vol. 1702. Athens, Greece: AIP Publishing, March 20-23 2015. doi: 10.1063/1.4938892 pp. 100015–1–100015–4.

[23] M. Szpyrka, P. Matyasik, and M. Wypych, "Generation of labelled transition systems for alvis models using haskell model representation," in *Proceedings of the 22nd International Workshop on Concurrency, Specification and Programming (CS&P 2013)*, vol. 1032. Warsaw, Poland: CEUR Workshop Proceedings, 2013, pp. 409–420.

[24] E. A. Emerson, "Model checking and the Mu-calculus," in *Descriptive Complexity and Finite Models*, ser. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, N. Immerman and P. G. Kolaitis, Eds. American Mathematical Society, 1997, vol. 31, pp. 185–214.

[25] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A toolbox for the construction and analysis of distributed processes," in *Computer Aided Verification (CAV'2007)*, ser. LNCS, vol. 4590. Berlin, Germany: Springer, 2007, pp. 158–163.

[26] S. Kripke, "A semantical analysis of modal logic I: normal modal propositional calculi," *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, vol. 9, pp. 67–96, 1963, announced in *Journal of Symbolic Logic*, **24**, 1959, p. 323.

[27] ZUE S.A. (2015) Infrared-systems. Http://www.grupazue.pl/en/oferta/produkcja-urzadzen/infrared-systems.

[28] M. Gorowski. (2014) Rail transport. Http://www.transportszynowy.pl/zwrotnicetramsterowanie.php.

[29] J. Ciszewski, K. Kunecki, W. Markowski, J. Sawicki, and M. Sobecki, *SITP Guideline WP-02:2010. Fire alarm systems. The design*, 2010.

[30] J. Biernacki, A. Biernacka, and M. Szpyrka, "Action-based verification of RTCP-nets with CADP," in *International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2015)*, ser. AIP Conference Proceedings, vol. 1702. Athens, Greece: AIP Publishing, March 20-23 2015. doi: 10.1063/1.4938888 pp. 100011–1–100011–4.