Generation of Labelled Transition Systems for Alvis Models Using Haskell Model Representation

Marcin Szpyrka, Piotr Matyasik, and Michał Wypych

AGH University of Science and Technology
Department of Applied Computer Science
Al. Mickiewicza 30, 30-059 Kraków, Poland
{mszpyrka,ptm,mwypych}@agh.edu.pl

Abstract. Alvis is a formal modelling language for concurrent systems with the following advantages: a graphical modelling language used to define interconnections among agents, a high level programming language used to define the behaviour of agents and the possibility of a formal model verification. An Alvis model semantics find expression in an LTS graph (*labelled transition system*). Execution of any language statement is expressed as a transition between formally defined states of such a model. An LTS graph is generated using Haskell representation of an Alvis model and user defined Haskell functions can be used to explore the graph. The paper deals with the problem of translation of an Alvis model into its Haskell representation and discusses possibilities of model verification with the so-called Haskell filtering functions.

Keywords: Alvis, formal verification, Haskell, labelled transition system

1 Introduction

Alvis [1], [2], [3] is a formal modelling language being developed at AGH-UST in Krakow, Department of Applied Computer Science. The main aim of the project is to provide a flexible modelling language for concurrent systems with possibilities of a formal models verification. Alvis combines advantages of high level programming languages with a graphical language for modelling interconnections between subsystems (called agents) of a concurrent system. States of a model and transitions among them are represented using a labelled transition system (LTS graph for short [4]) which is used to verify the model formally by using model checking techniques [5], [6], [7], [8]. Previous research on Alvis was focused on the untimed version of the language with α^0 system layer (multiprocessor environments) [1], [2]. Using this system layer makes Alvis an alternative for other formalisms as Petri nets [9], [10], [11], process algebras [12] etc., but main advantages of Alvis approach for systems modelling are: similarity of Alvis syntax and syntax of procedural languages, graphical language for modelling interconnections between agents and the method of models states description which is similar to information provided by software debuggers.

The scheme of the modelling and verification process with Alvis is shown in Fig. 1. From the users point of view, the process starts from designing a model using prototype modelling environment called *Alvis Editor*. The designed model is stored using XML

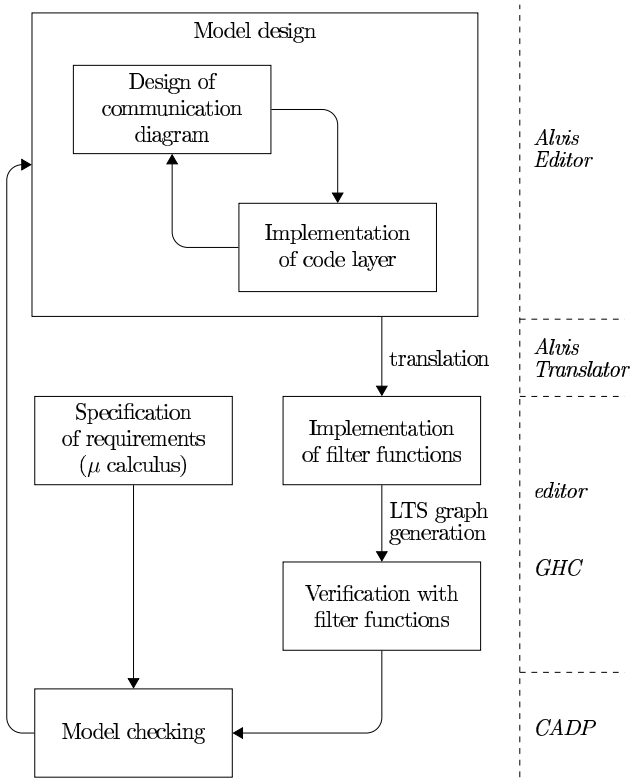


Fig. 1. Modelling and verification process with Alvis

file format. Then it is translated into Haskell [13] source code and its Haskell representation is used to generate the LTS graph. The Haskell functional language has been chosen as middle-stage representation of an Alvis model because Haskell is also used as a part of Alvis language i.e. Alvis uses Haskell to define parameters, data types and data manipulation functions. Haskell has been also used to implement the LTS graph generation algorithm. Such an LTS graph is stored as a Haskell list. A designer has access to such a source code, so user-defined Haskell functions (called *filtering functions*) that search an LTS graph for some states or parts of the graph that meet given requirements can be included into the model. The source code is compiled with GHC compiler. The results of received program execution are the LTS graph for the given model and the report of the model verification with filtering functions.

Another approach to Alvis model verification relies on using CADP toolbox [14]. CADP offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking [5], [6], [7]. An Alvis LTS graph can be converted into BCG (Binary Coded Graphs) format which is one of acceptable input formats for CADP Toolbox. Then the CADP *evaluator* is used to check whether the model satisfies requirements given as regular alternation-free μ-calculus formulae [6], [15], [7].

The paper deals with the problem of translation of an Alvis model into its Haskell representation and discusses possibilities of model verification with filtering functions. Selected ideas connected with Alvis and LTS graphs are presented in Secion 2. Section 3 deals with the middle-stage Haskell model representation. Methods of LTS graph exploration are considered in Section 4. A short summary is given in the final section.

2 Alvis Models

An Alvis *model* is defined as a triple $\mathbf{A} = (H, B, \varphi)$, where H is a *hierarchical communication diagram*, B is a syntactically correct *code layer*, and φ is a *system layer*. In this paper we consider models with α^0 system layer only. This layer is based on the assumption that each active agent has access to its own processor and in case of conflicts agents priorities are taken under consideration. If two or more agents with the same highest priority compete for the same resources, the system works non-deterministically. Moreover, before generation of the Haskell model representation models are transformed into equivalent non-hierarchical form. Thus, from now on we will consider models defined as $\mathbf{A} = (D, B, \alpha^0)$, where $D = (\mathcal{A}, \mathcal{C}, \sigma)$ is a *non-hierarchical communication diagram*, where: $\mathcal{A} = \{X_1, \ldots, X_n\}$ is the set of *agents* consisting of two disjoint sets, $\mathcal{A}_A$, $\mathcal{A}_P$ such that $\mathcal{A} = \mathcal{A}_A \cup \mathcal{A}_P$, containing *active* and *passive* agents respectively; $\mathcal{C} \subseteq \mathcal{P} \times \mathcal{P}$, where $\mathcal{P}$ is the set of all ports, is the *communication relation*, such that:

- a connection cannot be defined between ports of the same agent;
- procedure ports are either input or output ones i.e. ports defined as procedures are used to transfer signals (values) either to or from a passive agent;
- a connection between an active and a passive agent must be a procedure call;
- a connection between two passive agents must be a procedure call from a non-procedure port.

The start function σ makes it possible to delay activation of some agents.

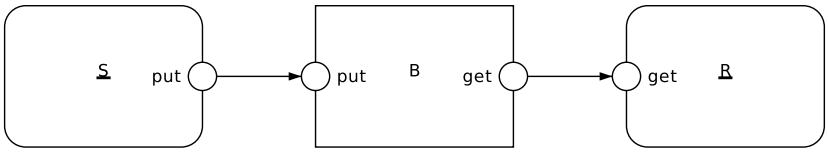
Active agents perform some activities and are similar to tasks in Ada programming language [16]. Each of them can be treated as a thread of control in a concurrent system. By contrast, *passive agents* do not perform any individual activities, and are similar to protected objects (shared variables). Passive agents provide other agents with a set of procedures (services). For more details see [2]. A description of the Alvis syntax can be also found at the Alvis project web site *http://fm.kis.agh.edu.pl*.

An example of Alvis model for a *sender-buffer-receiver* system is given in Fig. 2. Agent S (sender) puts sequentially valueless signals to the buffer (agent B) and agent R (receiver) gets such signals from the buffer. Agent B offers two procedures (services, ports) to connected agents.

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

Definition 1. *A* Labelled Transition System *is a tuple* $\mathrm{LTS} = (S, A, \rightarrow, s_0)$*, where:*

- S *is the set of* states *and* $s_0 \in S$ *is the* initial state*;*
- A *is the set of* actions*;*
- $\rightarrow \subseteq S \times A \times S$ *is the* transition relation.




```
agent Sender {
  loop { out put; }}                          -- 1, 2

agent Buffer {
  i :: Int = 0;
  proc (i == 0) put { in put; i = 1; }        -- 1, 2
  proc (i /= 0) get { out get; i = 0; }}      -- 3, 4

agent Receiver {
  loop { in get; }}                           -- 1, 2
```

Fig. 2. Alvis model for *sender-buffer-receiver* system

The usage of LTS graphs is a universal method of a state space representation and is omnipresent in formal modelling languages. Different languages like Petri nets, time automata, process algebras etc. use different methods of describing nodes and edges in LTS graphs. They also use different names for them e.g. reachability graphs in Petri nets [9], [11], but the general structure of these graphs is still the same. The common feature of these approaches is the encoding of the considered system states using mathematical ideas typical for the chosen formalism. On the other hand they differ from methods used in programming languages significantly. In contrast to this, Alvis syntax is very similar to procedural programming languages and the used method of a model state description is similar to information provided by software debuggers. A state of an Alvis model is represented as a sequence of agents states [4], [2]. To describe the current state of an agent we use the following pieces of information.

- *Agent mode* (am) represents the type of the current agent activity e.g., *Running* (X) means that an agent is performing one of its statements, while *waiting* (W) means that the agent is waiting for an event (for active agents). For passive agents, *waiting* means that the corresponding agent is inactive and waits for another agent to call one of its accessible procedures. On the other hand, *Taken* (T) means that one of the passive agent procedures has been called and the agent is executing it.
- *Program counter* (pc) points out the current statement of an agent.
- *Context information list* (ci) 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.
- *Parameters values list* contains the current values of the corresponding agent parameters.

LTS graph for model from Fig. 2 is shown in Fig. 3. Let us consider state 11:

- $am(B) = \mathsf{T}$, $pc(B) = 1$ – agent B is taken and performs its first step;

– $ci(S) = [proc(B.put, put)]$ – agent S has called procedure $B.put$ via port $S.put$;
– $am(R) = \mathsf{W}$, $pc(R) = 2$, $ci(R) = [in(get)]$ – agent R is waiting after performing step 2 (`out` statement), context information list points out that the agent is waiting for finalisation of the communication that has been initialised via port $R.get$.

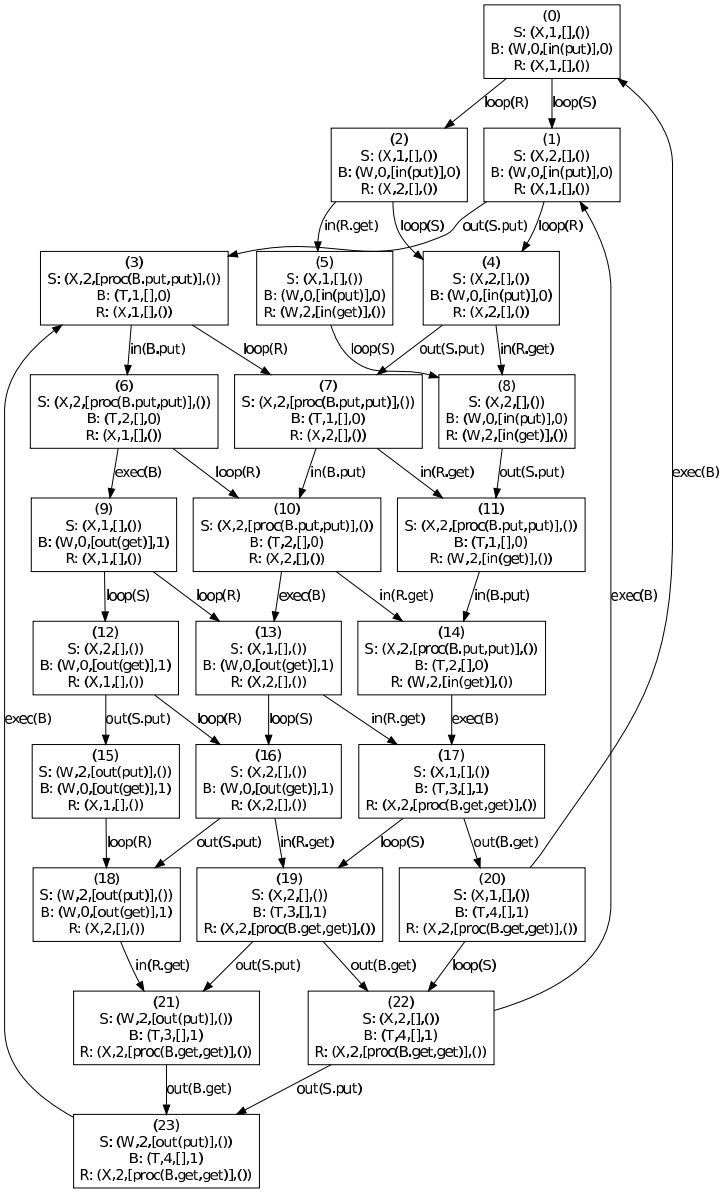


Fig. 3. LTS graph for model from Fig. 2

A state of a model can be changed as a result of executing a step. Most of the Alvis statements e.g. *exec*, *exit*, etc. are *single-step* statements. By contrast, *if*, *loop* and *select* are *multi-step* statements. We use recursion to count the number of steps for multi-step statements. For each of them the first step enters the statement interior. Then we count steps of statements put inside curly brackets. The set of all possible steps for the considered Alvis models contains the following elements:

- *exec* – performs an evaluation and assignment;
- *exit* – terminates an agent or a procedure,
- *if* – enters an if statement,
- *in* – performs communication (input side),
- *jump* – jumps to a label,
- *loop* – enters a loop,
- *null* – performs an empty statement,
- *out* – performs communication (output side),
- *select* – enters a select statement,
- *start* – starts an inactive agent,
- *io* – performs communication (both sides).

Results of all these steps execution have been formally defined in [2].

3 Haskell Model Representation

An Alvis model is translated into Haskell source code and this middle-stage representation is used for LTS graph generation and for verification purposes. In order to obtain it following steps has to be performed:

- flattening Alvis hierarchical model,
- constructing agents list,
- generating state tuple for every agent,
- generating system state tuple by combining individual agents states, ordered respectively to agent list generated earlier,
- generating the `enable` function according to Alvis language rules [1], [2],
- generating the `fire` function,
- appending LTS generation code,
- appending LTS export code,
- appending main function.

Some elements common for all Alvis models are defined inside *Alvis* module, which is included into any model source file. This module contains for example enumerated data types for possible steps, entries of context information lists etc. Individual source files generated for models have the following structure:

1. User defined data types (if any).
2. User implemented functions for parameters manipulation (if any).
3. Definition of individual agents state types and the corresponding model state type.
4. Definition of the initial state.
5. Implementation of *enable* and *fire* functions.
6. Implementation of LTS graph generation algorithm.
7. Implementation of export functions into: text, *dot* and *aldebaran* formats.

8. User implemented filtering functions.
9. The *main* function.

The initial part of the Haskell source file for the model from Fig. 2 is shown in Fig. 4. It contains: the list of the model agents, data types for these agents' states and for the model state (type `State`), data type for LTS graph node representation (type `Node`) and the initial model state. The `Node` type contains: the node index, a model state and a list of enabled steps in that state (a step label and the target node number).

```
module Main where
import Alvis

agents = ["S", "B", "R"]

type SState = (Mode, Int, [ContentsInfo], ())
type BState = (Mode, Int, [ContentsInfo], (Int))
type RState = (Mode, Int, [ContentsInfo], ())

type State = (SState, BState, RState)
type Node = (Int, State, [(String, Int)])

s0 :: State
s0 = ((X,1,[],()), (W,0,[CIn "put"],(0)), (X,1,[],()))
```

Fig. 4. Part of Haskell source file for model from Fig. 2

The Haskell representation of an Alvis model behaviour is based on the so-called *enable-fire* approach which takes inspiration from Petri nets. The *enable* function takes a model state and an agent name and provides a list of the agent steps that are enabled (can be performed) in the state. The *fire* function takes an enabled step and a state and gives a new state that is the result of the step execution. A pseudo-code representation of the LTS graphs generation algorithm is shown in Fig. 5. It requires three elements to be given based on selected system layer, model structure and agents code. The first one is an initial state which can be straightforward extracted from system description. The second and third ones are *enable* and *fire* functions mentioned before.

The algorithm runs until it processes all elements form $nodeList$. This list after computation contains the resulting LTS. The $stateList$ is a helper list for quick check if a given state was already computed. For each state (line 5) algorithm computes a list of all enabled transitions (line 7). Afterwards it fires every transition and checks whether the resulting state has already been computed (line 11). Effectiveness of this step is crucial for computation time of the whole algorithm. If the state is present on $stateList$ a new transition is added to the currently checked state (line 14). Otherwise a new state is added to $nodeList$ and $stateList$ (lines 17-18) and a transition from currently investigated state to the new one is appended (line 16). A small part of the *enable* and *fire* functions generated for the considered example is given in Fig. 6.

1: $nodeList \leftarrow [(0, s_0, [])]$
2: $curIdx \leftarrow 0$
3: $stateList \leftarrow [(0, s_0)]$ ▷ for quick access to index of a given state
4: **while** $curIdx < \#nodeList$ **do**
5: $s \leftarrow nodeList[curIdx].state$
6: $tl \leftarrow nodeList[curIdx].transitions$
7: $transList \leftarrow$ transitions enabled in state s
8: **while** $transList$ not empty **do**
9: $trans \leftarrow$get and remove first element from $transList$
10: $state \leftarrow fire(trans, s)$
11: $i \leftarrow$ index of $state$ in $stateList$
12: $last \leftarrow \#nodeList - 1$ ▷ indexing from 0
13: **if** $\exists i$ **then** ▷ $state$ already on list
14: $nodeList[curIdx] \leftarrow (curIdx, s, tl +\!\!+ [t, i])$
15: **else** ▷ $state$ is new
16: $nodeList[curIdx] \leftarrow (curIdx, s, tl +\!\!+ [t, last + 1])$
17: $nodeList$ append $(last + 1, state, [])$
18: $stateList$ append $(last + 1, state)$
19: **end if**
20: **end while**
21: $curIdx \leftarrow curIdx + 1$
22: **end while**

Fig. 5. LTS graph generation algorithm

```
enable :: State -> String -> [TTransition]
enable ((am1,pc1,ci1,()),(am2,pc2,ci2,pv2),(am3,pc3,ci3,())) "S"
  | am1 == X && pc1 == 1 = [TLoop "S" 1]
  | am1 == X && pc1 == 2 && (procfree ci1) && am2 == W
  && elem (CIn "put") ci2  = [TOutAP "S.put" "B.put" 2]
  | am1 == X && pc1 == 2 && (procfree ci1) = [TOut "S.put" 2]
  | otherwise = []

fire :: TTransition -> State -> State
fire (TOutAP "S.put" "B.put" 2)
  ((am1,pc1,ci1,pv1),(am2,pc2,ci2,pv2),(am3,pc3,ci3,pv3))
  = ((am1,pc1,ci1 ++ [CProc "B.put,put"],pv1),(T,1,[],pv2),
    (am3,pc3,ci3,pv3))
```

Fig. 6. Part of the source code for *enable* and *fire* functions for model from Fig. 2

The generation of LTS graphs is the main aim of using the Haskell model representation. However, it should be underlined that the source file contains also functions for exporting LTS graphs into different formats. The most important one is the *aldebaran* format. LTS graphs stored in the *aldebaran* format can be automatically converted into BCG (Binary Coded Graphs) format which is one of the acceptable input formats for the CADP Toolbox. The conversion method is provided by one of CADP tools.

4 Model Verification with Filtering Functions

The Haskell approach to Alvis model verification requires Haskell programming skills, because the so-called *filtering functions* must be user-defined and included into the generated source file. Some of the functions are universal and can be included into any model, so it is possible to import them from an external Haskell module. However, most of these functions are based on the considered model `State` type and must be defined for a model individually.

```
deadState :: Node -> Bool
deadState (n,s,ls) = ls == []
-- filter deadState lts

singleOutState :: Node -> Bool
singleOutState (n,s,ls) = (length ls) == 1
-- filter singleOutState lts
```

Fig. 7. Examples of universal filtering functions

Examples of universal filtering functions are given in Fig. 7. The `deadState` function searches for states without outgoing arcs (dead states), while the `singleOutState` function searches for states with single outgoing arc. Included comments illustrate the usage of these functions.

```
sRunning :: Node -> Bool
sRunning (_,((X,_,_,_),_,_),_) = True
sRunning _ = False

twoWaiting :: Node -> Bool
twoWaiting (_,((W,_,_,_),(W,_,_,_),_),_) = True
twoWaiting (_,((W,_,_,_),_,(W,_,_,_)),_) = True
twoWaiting (_,(_,(W,_,_,_),(W,_,_,_)),_) = True
twoWaiting _ = False

procfree :: [ContentsInfo] -> Bool
procfree [] = True
procfree ((CProc _):_) = False
procfree (_:xs) = procfree xs

noProc :: Node -> Bool
noProc (_, ((_,_,ci1,_),_,(_,_,ci3,_)), _)
  | procfree ci1 && procfree ci3 = True
  | otherwise = False
```

Fig. 8. Examples of special filtering functions

These functions do not use the internal structure of the LTS graph node, thus can be used in any model. However, knowledge of the `State` type details is fundamental for implementing more sophisticated filtering functions. The main disadvantage of such functions is their adaptation to the given model. Examples of special filtering functions implemented for the model from Fig. 2 are shown in Fig. 8. The `sRunning` function searches for states with agent S in the *running* mode. Presented functions use the Haskell pattern matching mechanism. The underscore sign is a wild-card and its role changes depending on the place e.g. the first one replaces the number of a node, the second one – the program counter of agent S and the fifth – the state of agent B. The `twoWaiting` function searches for states with two agents in the *waiting* mode. The last `noProc` function searches for states when no procedure is executed. The auxiliary recursive `procfree` function searches a context information list for *proc* entries.

The functions presented so far are used to search for states which fulfil given filter condition. As shown in Fig. 7, they are used together with the standard `filter` function. More elaborated functions may search an LTS graph oneself. Example of such a function is given in Fig. 9. The `node2node` function returns pairs of nodes connected with an arc with the given label. It uses two auxiliary functions: `iNode` searches for a node with the given number and `endNodeNo` searches for the number of the end node for the given arc.

```
iNode :: Int -> [Node] -> Node
iNode i ((n,s,ls):ns)
  | i == n = (n,s,ls)
  | otherwise = iNode i ns

endNodeNo :: String -> [(String, Int)] -> Int
endNodeNo _ [] = -1
endNodeNo s ((a,i):ls)
  | s == a = i
  | otherwise = endNodeNo s ls

node2node :: String -> [Node] -> [Node] -> [(Node,Node)]
node2node _ _ [] = []
node2node label ltscopy ((n,s,ls):ns) =
  if k /= -1
  then ((n,s,ls),(iNode k ltscopy))
    : (node2node label ltscopy ns)
  else node2node label ltscopy ns
  where k = endNodeNo label ls
```

Fig. 9. Filtering function searching for parts of an LTS graph

5 Summary

Alvis is being developed to provide a simple tool for formal modelling and verification of concurrent systems. Compared to the most popular formalisms like Petri nets, process

algebras etc. its syntax is simple and very similar to procedural programming languages. The knowledge of all of the formal definitions presented in [2] is obsolete for the end user. From user's point of view, the most important are an Alvis model and its LTS graph generated for the model automatically.

The paper deals with the problem of LTS graphs generation for Alvis models. It has been solved using a middle-stage Haskell model representation. This approach has been chosen out of consideration for the usage of Haskell in Alvis models. The Haskell representation of an Alvis model is based on two functions called *enable* and *fire* that provide the list of transitions enabled in the given states and results of these transitions execution. The functions are used in the presented algorithm for LTS graphs generation. It should be emphasized that this *enable-fire* approach can be used for generation states spaces for other formalism. For example, it has been successfully used for XCCS process algebra [17], [18]. In this case, instead of Alvis 4-tuples string values have been used to represent states of individual agents (algebraic equations). Nevertheless, exactly the same Haskell implementation of the LTS generation algorithm has been used to generate LTS graphs. This stresses the flexibility of this approach. It is enough to adapt the *enable* and *fire* functions to a considered formalism and presented approach can be used for verification purposes.

The second advantage of the considered approach is the possibility of model verification using Haskell implemented algorithms (functions). The generated LTS graph is stored as a Haskell list, thus not only is it possible to translate it into the *aldebaran* format and use CADP toolbox to verify its properties, but also user defined Haskell functions can be used to explore an LTS graph. This Haskell based approach is a completion of the CADP based verification. Analysis of Alvis models can be realized using the CADP *evaluator* tool. In such approach, a specification of requirements is given as a set of μ-calculus formulas [6] and the tool is used to check whether the model LTS graph satisfies them. It should be emphasized that this is an action based approach. A μ-calculus formula concerns actions labels while states of considered model are represented using their numbers only. On the other hand, the Haskell approach is rather states oriented. We can use the Haskell pattern matching mechanism to filter states that fulfil given requirements. Moreover, the Haskell approach can be used to implement user defined verification algorithms that search for some specified parts of an LTS graph and are not provided by verification toolbox. For example, this is a good path to test user-defined non-standard verification procedures fast. Moreover, Haskell expressiveness allows to fit even quite complicated algorithms in a few lines of code as compared to imperative languages.

References

1. Szpyrka, M., Matyasik, P., Mrówka, R.: Alvis – modelling language for concurrent systems. In Bouvry, P., Gonzalez-Velez, H., Kołodziej, J., eds.: Intelligent Decision Systems in Large-Scale Distributed Environments. Volume 362 of Studies in Computational Intelligence. Springer-Verlag (2011) 315–341
2. Szpyrka, M., Matyasik, P., Mrówka, R., Kotulski, L.: Formal description of Alvis language with α^0 system layer. Fundamenta Informaticae (2013) (to appear).

3. Szpyrka, M., Matyasik, P., Wypych, M.: Alvis language with time dependence. In: Proceedings of the Federated Conference on Computer Science and Information Systems, Krakow, Poland (2013) 1607–1612
4. Kotulski, L., Szpyrka, M., Sędziwy, A.: Labelled transition system generation from Alvis language. In König, A., et al., eds.: Knowledge-Based and Intelligent Information and Engineering Systems – KES 2011. Volume 6881 of LNCS. Springer-Verlag (2011) 180–189
5. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, London, UK (2008)
6. Emerson, E.A.: Model checking and the Mu-calculus. In Immerman, N., Kolaitis, P.G., eds.: Descriptive Complexity and Finite Models. Volume 31 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1997) 185–214
7. Mateescu, R., Sighireanu, M.: Efficient on-the-fly model-checking for regular alternation-free μ-calculus. Technical Report 3899, INRIA (2000)
8. Penczek, W., Półrola, A.: Advances in Verification of Time Petri nets and Timed Automata. A Temporal Logic Approach. Volume 20 of Studies in Computational Intelligence. Springer-Verlag (2006)
9. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
10. Suraj, Z., Fryc, B.: Timed approximate Petri nets. Fundamenta Informaticae **71**(1) (2006) 83–99
11. Szpyrka, M.: Analysis of RTCP-nets with reachability graphs. Fundamenta Informaticae **74**(2–3) (2006) 375–390
12. Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science, Upper Saddle River, NJ, USA (2001)
13. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. O'Reilly Media, Sebastopol, USA (2008)
14. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In Damm, W., Hermanns, H., eds.: Computer Aided Verification. Volume 4590 of LNCS., Springer-Verlag (2007) 158–163
15. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27**(3) (1983) 333–354
16. Barnes, J.: Programming in Ada 2005. Addison Wesley (2006)
17. Balicki, K., Szpyrka, M.: Tag abstraction for XCCS modelling language. In: Proceedings of the Concurrency Specification and Programming Workshop (CSP 2009). Volume 1., Krakow, Poland (September 28-30 2009) 26–37
18. Balicki, K., Szpyrka, M.: Formal definition of XCCS modelling language. Fundamenta Informaticae **93**(1-3) (2009) 1–15