UDC 681.234

L. KOTULSKI, M. SZPYRKA, A. SĘDZIWY, K. GROBLER-DĘBSKA

AGH University of Science and Technology, Krakow, Poland

ON GENERATION OF COMPOSITE LABELLED TRANSITION SYSTEMS FOR ALVIS PASSIVE AGENTS

The paper presents the method of generating Labelled Transition Systems for passive agents in Alvis models. Alvis is the language designed for the modelling and formal verification of embedded systems. The key concept of Alvis is an agent which is any distinguished part of a considered system with defined identity persisting in the time. Alvis combines a graphical modelling of interconnections among agents with a high level programming language used for describing a behaviour of agents. The basic property of the Alvis Toolkit is the ability of a direct generation of a formal system description from an Alvis source code. The LTS graphs represent control flow related to Alvis code execution and they are the base for the formal model verification.

Keywords: *Alvis modeling language, embedded systems, microinstruction, formal verification*

Introduction

Alvis [1, 2] is the novel modelling language designed for real-time systems, especially for embedded ones. The main goal of the Alvis project was to strike a happy medium between formal and practical, user-frendly modelling languages. From programmers point of view, it is necessary to design two layers of an Alvis model. The code layer uses Alvis statements supported by the Haskell functional programming language to define a behaviour of individual agents. The graphical layer (communication diagrams) is used to define communication channels between agents. The layer takes the form of a hierarchical graph, that allows designers to combine sets of agents into modules that are also represented as agents (called hierarchical ones). Alvis modelling environment called Alvis Toolkit creates in parallel a model of a considered embedded system and the corresponding LTS graph (Labeled Transition System) that is its formal representation. The LTS graph can be formally verified with the help of the CADP toolbox [4].

The paper is organized as follows. Section 1 provides the short presentation of the Alvis modelling language. The formal definition of an agent state is introduced in Section 2. The generation of an LTS graph for a single passive agent is described in Section 3 and the definition of LTS for the whole system in presented in Section 4. The algorithm of merging single LTSs into LTS representing the whole system is presented in Section 5. Finally, an example of LTS merging is presented in Section 6.

1. Alvis overview

Alvis is a successor of the XCCS modelling language [4, 5], which was an extension of the CCS process algebra [7, 8]. However, instead of algebraic equations, Alvis uses a high level programming language based on the Haskell syntax.

An Alvis model consists of three layers, but the last one (system layer) is predefined. The system layer is used for the simulation and analysis (generation of an LTS graph) purposes. An Alvis model is a system of agents that usually run concurrently, communicate one with another, compete for shared resources etc. The agents, in Alvis, are used for the design of communication diagrams (see Fig. 3). Active agents perform some activities and are similar to tasks in the Ada programming language [9], [10]. Each of them can be treated as a thread of control in a concurrent system. Passive agents (Agent Container) are used to store data shared among agents and to avoid the simultaneous use of such data by two or more agents.

An agent can communicate with other agents through ports drawn as circles placed at borders of rounded boxes or rectangles. A communication channel is defined for two agents and connects two ports. Communication channels are drawn as lines . One-way communication channel (connection (X1.p,X2.q)) contain an arrowhead that points out the input port for the particular connection.

The code layer is used to define data types used in a considered model, functions for data manipulation and behaviour of individual agents. The layer uses the Haskell functional language (e.g. the Haskell type system) and original Alvis statements. The set of AlvisCL statements is given in Table 1. To simplify the syntax, following symbols have been used. A stands for an agent name, p stands for a port name, x stands for a parameter, g, g1, g2,... stand for guards (Boolean conditions), e stands for an expression and ms stands for milliseconds. Each non-hierarchical agent placed in the communication diagram must be defined in the code layer and vice-versa.

© L. Kotulski, M. Szpyrka, A. Sędziwy, K. Grobler-Dębska

Table 1

Table of selected some Alvis statements

Statement	Description
`if (g1) {...} elseif (g2) {...} ... else {...}`	Conditional statement.
`in p` `in p x`	Collects a signal via the port p. Collects a value via the port p and assigns it to the parameter x.
`loop (g) {...}` `loop {...}`	Repeats execution of the contents while the guard if satisfied. Infinite loop
`out p` `out p x`	Sends a signal via the port p. Sends a value of the parameter x via the port p; a literal value can be used instead of a parameter.
`select { alt (g1) {...} alt (g2) {...} ... }`	Selects one of the alternative choices
`proc (g) p {…}`	Defines the procedure for the port p of a passive agent.

2. Models

An embedded system designed with the help of an object abstraction (expressed by hierarchical agents) can be finally represented by a set of non-hierarchical agents cooperating in the way described by the maximal flat representation of the communication diagram. The polynomial algorithm of designing such a flat representation was described in [2].

In this paper, primarily we are interested in a characteristic of states of passive agents. The characteristic of states of passive agents was described in [2]. A current state of an agent is represented by a tuple with four pieces of information: agent mode *(am),* program counter *(pc),* context information list *(ci),* parameters values tuple *(pv).* An passive agent is always in one of two modes: *waiting* or *taken*.

The *waiting* means that the agent to call one of its accessible procedures. In such a situation the program counter is equal to zero and the context information list contains names of accessible procedures. In any state, the parameters values list contains the current values of the agent parameters.

The *taken* mode means that one of the passive agent procedures has been called and the agent is executing it. In such a case, *ci* contains the name of the called procedure (i.e. the name of the port used for current communication). The *pc* points out the index of the next statement to be executed or the current statement if the corresponding active agent is waiting.

The formal definition of an agent state is as follows.

Definition 1. *A state of an agent X is a tuple S(X) = (am(X), pc(X), ci(X), pv(X)), where am(X), pc(X), ci(X) and pv(X) denote mode, program counter, context information list and parameters values of the agent X respectively.*

3. Labelled Transition System for Single Passive Agent

Active agent statements which form the Alvis code may be either single- or multi-step ones. An agent state may be changed in a result of executing a program step.

Passive Agents provide a set of procedures that can be called by other agents and their executions are mutually exclusive. Each procedure has its own port assigned and a communication with a passive agent via that port is treated as the corresponding procedure call. Depending on the communication direction, such a procedure may be used either to send or to collect some data from the passive agent.

```
Agent_Container {
x1,x2,s :: Int;
      proc p1 {
         in p1 x1;}                      --1
      proc p2 {
         in p2 x2;}                      --2
      proc ret { s = (x1 + x2)/2;        --3
                    out ret s;}          --4
}
```

Listing. 1. The Alvis code for the sample passive agent

The Listing 1 shows the Alvis code of the passive agent Container. Container agent exposes three procedures, namely p1, p2, ret, matched to corresponding input/output ports. The statement numbering convention used in the listing above is described in [3].

To simplify the formal description of transition the formal description of transitions, for agent X we use the following notation convention:

– when exist transition t $S \xrightarrow{t} S'$ the state is S = (amX, pcX, ciX, pvX) is substituted by S' = (am'X, pc'X, ci'X, pv'X)

– for a port p, p* denotes a port associated with p in a communication diagram (note that p = p**),

– we provide $nextpc_X$ function that determines a number of a next step (a next program counter for an agent) and $instr_X$ *(i)* function that determines a type of instruction associated with number of step, *i*.

– for a currently considered agent we neglect the postfix pointing an agent (i.e. we put a instead a_X).

The initial state of passive agent is *waiting* and he waits until some other agent Y has put a message and is pending on port p* then a message is taken form p, *am'=taken* and *pc'=nextpc* (executes procedure p); note that the state of agent Y will be also changed from run-

ning to waiting and pc_Y points out an index of the next agent statement after out statement. If passive agent is *taken*, then some other agent Y, which waits in a select guard containing out p*(i.e. ci_Y=[q(...,out p*:npc,...]) must waiting until a passive agent X will be back waiting.

The execution of the step *i* (called transition from state *S* to *S'*) by the *taken* agent means a statement execution of current procedure and changes a current state. More details about the transition idea in Alvis models can be found in [2].

Table 2

Table of relationship between
the mode and program counter for passive agents

am(X)	pc(X)
waiting	0
taken	current statement of current procedure.

4. Labelled Transition Graphs

Assume that $\overline{A} = \left(D, B, \alpha^0\right)$ is an Alvis model. For the pair of states S, S' we say that S' is directly reachable from S iff there exists transition $t \in T$ such that $S \xrightarrow{t} S'$.

All states directly reachable from S is denoted as R(S). We say that S' is reachable from S iff there exists a sequence of states $S^{(1)}, \ldots, S^{(k+1)}$ and $t^{(1)}, \ldots, t^{(k)} \in T$ a sequence of transitions such that

$$S = S^{(1)} \xrightarrow{t^{(1)}} S^{(2)} \xrightarrow{t^{(2)}} \ldots \xrightarrow{t^{(k)}} S^{(k+1)} = S'$$

The set of all states that are reachable from the initial state S_0 is denoted by $R^*(S_0)$.

States of an Alvis model and transitions between them are represented by a labelled transition system (LTS graph for short). An LTS graph is a directed graph LTS=(V, E, L), such that $V = R(S_0)$, $L = T$, and

$$E = \left\{(S, t, S') : S \xrightarrow{t} S', \text{where} S, S' \in R\left(S_0\right) \text{and} t \in L\right\}$$

In other words, an LTS graph represents all states reachable from S_0 and transitions between them in the form of a directed graph.

Primarily we generate LTS graph for a single agent, starting from AlvisCL representation of its behaviour. Let us consider the agent Container presented in Listing 1 with LTS graph shown in Fig. 1.

The initial state of the passive agent is *waiting* with all accessible ports. Note that transition from a taken state leads to another taken state unless it starts in a last statement of a given procedure.

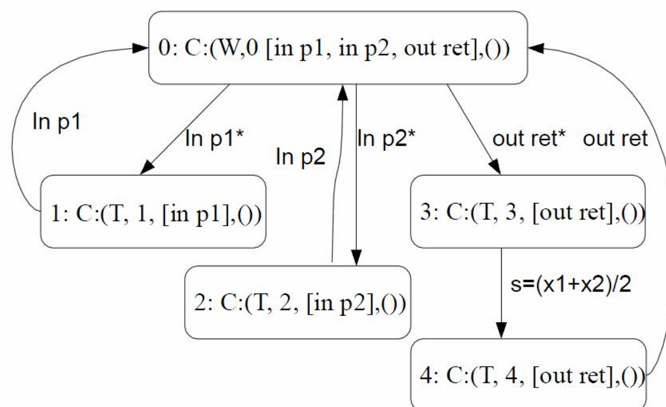

Fig. 1. LTS diagram for listing 1

5. LTS Generation

In this section we assume that we have a system consisting of N Alvis passive agents $X_1, X_2, \ldots, X_N$ and M Alvis active agents $Y_1, Y_2, \ldots, Y_M$. Connections between them are represented by a communication diagrams. Using the method described in Section 3 we can generate LTS for single passive agents. Methods how to generate LTS for single active agents and merge such individual LTS's for active agents into composite on representing a system was described in [3]. Now we will show how to merge such individual LTS's both active and passive agents into composite on representing a whole system.

Let us compare the method of merging active agents only (as presented in [3]) and the case when a passive agent exists in the set of merged agents.

The tables 2 and 3 describe modified algorithms from the paper [3] for the general system with active and passive agents.

Let $S_{k,0}$ represents the initial state of the agent X_k then $R^*(S_{k,0})$ represents states of LTS representing k-th agent. By A_k we denote cardinality of $R^*(S_{k,0})$ set for active agents. By P_k we denote cardinality of $R^*(S_{k,0})$ set for passive agents. By L we denote maximal number of states directly reachable from any state belonging to any single LTS.

Now we can formulate two theorems.

Table 3

Table of the Algorithm 1.
CheckTransition

	input : x – an individual agent's LTS state, s – a composite LTS state **output:** S – set of all states accessible from s
1	**begin**
2	**if** X ← the active agent described by x;
3	**if** *no* **in/out** *in the current statement* **then**
4	**foreach** *running state x' directly reachable from x* **do**
	s' ← s < x ;
5	*S ← S ∪ {s'};*
6	**else if** *current statement contains* **in/out** *and some active*
7	*agent Y waits for X* **then**
	y ← current state of Y ;
8	*x' ← running state directly reachable from x;*
9	*y' ← running state directly reachable from y;*
10	*s' ← s < x', y'*
11	*S ← {s'};*
12	**else if** *current statement contains* **in/out** *and some pas*
13	*sive*
	agent Y waits for X **then**
14	*y ← current state of Y ;*
15	*x' ← waiting state directly reachable from x;*
16	*y' ← taken state directly reachable from y;*
17	*s' ← s < x', y'*
18	*S ← {s'};*
19	**else if** *current statement contains* **in/out** *and no agent*
	waits
20	*for X* then
21	*x'← waiting state directly reachable from x;*
22	*s' ← s < x*
23	*S ← {s'};*
24	**else if** X ← the passive agent described by x;
25	**if** *no* **in/out** *in the current statement* **then**
	foreach *taken state x' directly reachable*
26	*from x* **do**
27	*s' ← s < x ;*
28	*S ← S ∪ {s'};*
	else if *current statement contains* **in/out** *and some agent*
29	*active Y waits for X* **then**
30	*y ← current state of Y ;*
31	*x' ← waiting state directly reachable from x;*
32	*y' ← running state directly reachable from y;*
33	*s' ← s < x', y'*
34	*S ← {s'};*
	else if *current statement contains* **in/out** *and some pas-*
35	*sive*
36	*agent Y waits for X* **then**
37	*y ← current state of Y ;*
38	*x' ← taken state directly reachable from x;*
39	*y' ← taken state directly reachable from y;*
	s' ← s < x', y'
	S ← {s'};
	return S
	end

Lemma 1. *The number of states of a composite LTS graph generated from LTS graphs $X_1, \ldots X_N$ passive agents and $Y_1, \ldots Y_M$ active agents is not greater than*

$$S \le \prod_{i=1}^{M} A_i \cdot \prod_{i=1}^{N} P_i$$

Proof. The proof is based on the observation that a number of possible states for a composite LTS is not greater than a number of cells in N+M-dimensional hypercube

$$HC = R^*(S_{1,0}) \times R^*(S_{2,0}) \ldots \times R^*(S_{N+M,0}) .$$

By the Theorem 1 and the Theorem 2 for active agents which has been proved in paper [3], we get the following lemma.

Lemma 2. *The complexity of a composite LTS graph generation from individual LTS $X_1, \ldots X_N$ passive agents and $Y_1, \ldots Y_N$ active agents* is limited by

$$O\left(\left(N+M\right)\cdot L\cdot\left(\prod_{i=1}^{M} A_i \cdot \prod_{i=1}^{N} P_i\right)\right).$$

Proof. The general idea is using the hypercube HC and putting edges inside it reflecting all possible subsequent transitions, starting from the initial state $S_0 = (S_{1,0}, S_{2,0}, \ldots, S_{N+M,0})$. The unreachable states are removed. The finding of all possible transitions is made in the Algorithm 1 and it is based on the observation that a transition in the hypercube HC may be performed if some active agent, say A, is in the running state or some passive agent, say P, is in the taken state and they trigger a transition. Let consider it in a more detail.

The agent A being in running state may transit to following states:

1. *running*, when neither in nor out operation exists in a currently executed code line (Algorithm 1, line 3).

2. *running*, when either in or out operation is to be executed in a current code line and some active agent remains in a waiting state on suitable port, ready to contact (either write or read) A. In that case a waiting agent will also change its state to running (Algorithm 1, line 7).

3. *waiting*, when either in or out operation is in a current code line but no agent waits or passive agent waits for A (Algorithm 1, line 13 and 19).

The agent P being in taken state may transit to following states:

1. t*aken*, when neither in nor out operation exists in a currently executed code line (Algorithm 1, line 24).

2. *waiting*, when either in or out operation is to be executed in a current code line and some active agent remains in a waiting state on suitable port, ready to contact (either write or read) P. In that case a waiting agent will change its state to running (Algorithm 1, line 28).

3. *taken*, when either in or out operation is to be executed in a current code line and some passive agent remains in a waiting state on suitable port, ready to contact (either write or read) P. In that case a waiting agent will change its state to taken (Algorithm 1, line 7).

In Algorithm 2 we use the queue of composite states Q which initially is empty. To simplify pseudocodes the following notation was used in algorithms: s < x, y denotes that for a given composite state s we replace individual states of given agents with states x, y. States of other agents remain unchanged.

Table 4

Table of the Algorithm 2.

Merge $\left(x_1^{(0)}, x_2^{(0)}, \ldots x_K^{(0)}\right)$

input : $s^0 = \left(x_1^{(0)}, x_2^{(0)}, \ldots x_K^{(0)}\right)$ – a sequence of individual initial states of agents $X_1, X_2, \ldots X_K$ (some active and same passive) **output:** G = (V, E) – a composite LTS graph for $X_1, X_2, \ldots X_K$	
1	**begin**
2	$Q \leftarrow s^0$;
3	Mark all composite states as unvisited;
4	**while** *Q is nonempty* **do**
5	*s ← Q.dequeue();*
6	Mark s as visited;
7	Add s to *V* if not present;
8	***S*** *← ∅;*
9	**foreach** *running or taken state x in* ***s*** **do**
10	***S*** *←* ***S*** *∪ CheckTransition(x, s);*
11	**foreach** ***s'*** *∈* ***S*** **do**
12	Enqueue ***s'*** in *Q* if unvisited;
13	**if** ***s*** *∉ V* then
14	*V ← V ∪* {***s'***};
15	*E ← E ∪* {(***s***, ***s'***)};
16	*G = (V, E);*
17	**return** *G*;
18	**end**

To evaluate the computational complexity of the Algorithm 2 we should remark that since each composite state can be enqueued at least once (as unvisited) the while loop (Algorithm 2, line 4) can be executed not more than $\prod_{i=1}^{M} A_i \cdot \prod_{i=1}^{N} P_i$ times. The loop foreach (Algorithm 2, line 9) is executed N+M times and in each case the size of S is increased by not more than L, hence |S| ≤ (N+M). Thus body of the next loop foreach (Algorithm 2, lines from 12 to 15) can be executed not more than $(N+M) \cdot L \cdot \left(\prod_{i=1}^{M} A_i \cdot \prod_{i=1}^{N} P_i\right)$.

6. Example

To illustrate LTS graph generation we consider the model shown in Fig. 2 that represents communication between passive and one active agent.

The agent Container returns an arithmetic mean of the values, which are entered by independent processes and agent Active is the receiver. The LTS graph for this model is shown in Fig 3.

Initially we are in the state 0 defined as:

C:(waiting,0,[in p1, in p2, out ret],[]), A:(running,1,[],[]). In this state only agent A can run so we move to state 1 defined as C:(waiting,0,[in p1, in p2, out ret],[]), A:(running,2,[],[]). In state 1 again, only agent A can run so we move to state 2 defined as C:(taken, 3,[out ret],[]), A:(waiting,2,[in get],[]); the execution of *in get* shifts agent C to state 3 in his individual LTS and agent A is already waiting on the port get. In the state 2 the agent C is taken; the statement execution of procedure shift agent C to state 4 in his individual LTS and agent A is still waiting on the pot get. The execution of *out ret* shifts both agents to the initial state.

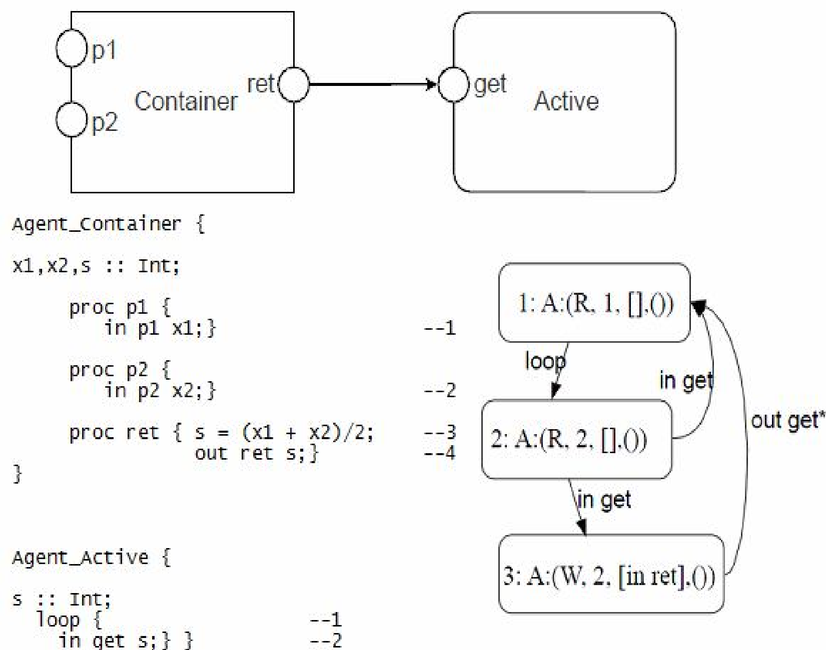

Fig. 2. Example

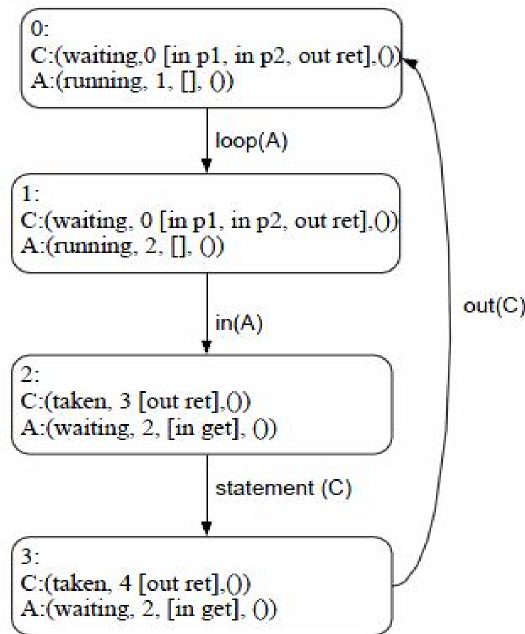

Fig. 3. Example – LTS graph

Conclusion

In the paper the algorithm of generation of the LTS for all agents present in a given system coded in the Alvis toolkit is presented. This generation is made in two phases: firstly we generate LTS graphs for single agents. In the second step we merge those graphs into a formal presentation of a whole system. This gives the possibility of the formal verification of the defined system properties.

The estimations of the space and computational complexities of this approach are also presented.

The disadvantage of generation of the LTS by this method is duplication of waiting states of active agent, which waits to ability to communicate to the taken passive agent. This state is performed in the composite LTS as many times as number of steps in called procedure, which has been called by another active agent. This issue is a continuation of further studies on Alvis toolkit.

References

1. Szpyrka, M. Alvis – modelling language for concurrent systems. Intelligent Decision Systems in Large-Scale Distributed Environments. SC [Text] / M. Szpyrka, P. Maryasik, R. Mrowka. – Heidelberg: Springer, 2012.

2. Formal introduction to Alvis modelling language [Text] / M. Szpyrka, P. Matyasik, R. Mrowka, L. Kotulski, K. Balicki // International Journal of Applied Mathematics and Computer Science (to appear, 2011).

3. Kotulski, L. Labelled Transition Systems Generation from Alvis Language [Text] / L. Kutulski, M. Szpyrka, A. Sedziwy // KES2011, LNAI. – 2011. – vol. 688. – P. 180 – 189.

4. CADP 2006: A toolbox for the construction and analysis of distributed processes [Text] / H. Gravel, F. Lang, R. Mateescu, W. Serwe // CAV 2007. LNCS. – 2007. – Vol. 4590. – P. 158 – 163.

5. Balicki, K. Formal definition of XCCS modelling language. Fundamenta Informaticae [Text] / K. Balicki, M. Szpyrka. – 2009. – № 93 (1 - 3). – P. 1 – 15.

6. Matyasik, P. Design and analysis of embedded systems with XCCS process algebra. PhD thesis, AGH University of Science and Technology [Text] / P. Matyasik // Faculty of Electrical Engineering, Automatics, Computer Science and Electronics. – Kraków, Poland, 2009.

7. Milner, R. Communication and Concurrency [Text]/R. Milner. – Prentice-Hall, Englewood Cliffs, 1989.

8. Reactive Systems: Modelling, Specification and Verification [Text] / L. Aceto, A. Ing´ofsd´ottir, K. Larsen, J. Srba // Cambridge University Press – Cambridge, 2007.

9. Barnes, J. Programming in Ada 2005. [Text] / J. Barnes // Addison-Wesley. – Reading, 2006.

10. Burns, A. Concurrent and real-time programming in Ada 2005 [Text] / A. Burns, A. Wellings // Cambridge University Press. – Cambridge, 2007.

Поступила в редакцию 2.03.2012

Рецензент: д-р техн. наук, проф. Б.М. Конорев, Национальный аэрокосмический университет им. Н.Е. Жуковского «ХАИ» , Харьков, Украина

СОЗДАНИЕ СЛОЖНОЙ РАЗМЕЧЕННЫХ ПЕРЕХОДНЫХ СИСТЕМ ДЛЯ ПАССИВНЫХ АГЕНТОВ ALVIS

Л. Котульски, М. Шпирка, А. Седживы, К. Гроблер-Дебска

Статья представляет метод создания размеченных переходных систем для пассивных агентов в моделях Alvis. Alvis является языком, спроектированным для моделирования и формальной верификации встроенных систем. Ключевым понятием в языке Alvis является агент – любая отличительная часть рассматриваемой системы, имеющая определенное обозначение, устойчивое во времени. Alvis объединяет графическое моделирование взаимосвязей между агентами с высокоуровневым языком программирования, использующимся для описания поведения агентов. Основным свойством инструментария Alvis является способность непосредственно создавать формальное системное описание из исходного кода Alvis. Графы размеченных переходных систем представляют последовательность операций, связанных с выполнением кода на языке Alvis, и они являются основой для формальной верификации моделей.

Ключевые слова: язык моделирования Alvis, встроенные системы, микроинструкции, формальная верификация.

СТВОРЕННЯ СКЛАДНИХ РОЗМІЧЕНИХ ПЕРЕХІДНИХ СИСТЕМ ДЛЯ ПАСИВНИХ АГЕНТІВ ALVIS

Л. Котульски, М. Шпирка, А. Седжіви, К. Гроблер-Дебська

У статті викладено метод створення розмічених перехідних систем для пасивних агентів у моделях Alvis. Alvis - це мова спроектована для моделювання та формальної веріфікації вбудованих систем. Ключовим поняттям у мові Alvis є агент – будь-яка відмітна частина системи, що розглядається, яка має певне позначення стійке в часі. Alvis об'єднує графічне моделювання взаємозв'язків між агентами з високорівневою мовою програмування, яка використовується для опису поведінки агентів. Основною властивістю інструментарію Alvis є здатність безпосередньо створювати формальний системний опис з вихідного коду Alvis. Графи розмічених перехідних систем становлять собою послідовність операцій, пов'язаних з виконанням коду на мові Alvis, і вони становлять основу для формальної верифікації моделей.

Ключові слова: мова моделювання Alvis, вбудовані системи, мікроінструкції, формальна веріфікація

Kotulski Leszek – Prof., holds a position of associate professor in AGH UST in Krakow, Poland, Department of Applied Computer Science. He has a MSc, PhD and DSc (habilitation) in Computer Science.

Szpyrka Marcin – Prof., holds a position of associate professor in AGH UST in Krakow, Poland, Department of Applied Computer Science. He has a MSc in Mathematics and PhD and DSc (habilitation) in Computer Science.

Sędziwy Adam – PhD, is the assistant professor at the Department of Applied Computer Science, AGH UST, Cracow, Poland.

Grobler-Dębska Katarzyna – MSc, the assistant in Department of Automatics, AGH UST in Krakow. She graduated in Applied Mathematics at the Jagiellonian University.