Proceedings of the 2014 Federated Conference on
Computer Science and Information Systems pp. 1639–1645

DOI: 10.15439/2014F267
ACSIS, Vol. 2

Alvis Virtual Machine

Piotr Matyasik
AGH University of Science and Technology
Department of Applied Computer Science
Al. Mickiewicza 30, 30-059 Krakow, Poland
Email: {ptm}@agh.edu.pl

***Abstract*—Alvis is a formal modelling language. It combines graphical modelling of communication schema and a high level programming language to describe behaviour of individual system entities. An Alvis model can be verified formally by using methods based on a system state space. The paper presents the design and the command list of the Alvis Virtual Machine. The aim of the project is to provide an execution environment for Alvis language. Moreover, one of the goals is to allow different hardware units to run Alvis models. Thus, a virtual machine was chosen as a solution.**

I. Introduction

ALVIS [1], [2], [3] is a formal modelling language designed to provide a user friendly method for developing concurrent systems, especially embedded ones. *Agents* are basic entities of Alvis models. Usually they run concurrently and communicate with one another. From a user point of view a model consist of two layers. The *code layer* provides a high level programming language used to describe agents behaviour. It's syntax is similar to C, Java or Pascal and it provides high level constructions as loop or conditional statements. The *graphical layer* (called a communication diagram) is a visual hierarchical language used to define communication channels between agents [1]. The language is being developed at AGH-UST in Krakow, Department of Applied Computer Science. An on-line manual and software supporting modelling with Alvis can be found at the project web site *http://fm.kis.agh.edu.pl*.

States of a model and transitions among them are represented using a labelled transition system (LTS graph [4]). An LTS graph is used to verify the corresponding model formally with model checking techniques [5]. The Alvis Compiler allows users to write LTS graphs in different formats.

Aldebaran format is used to export LTS graph to the CADP Toolbox [6]. Thus, system behaviour requirements can be provided by using μ calculus [7], [8] or XTL [9] and the CADP Toolbox can be used to check whether the model satisfies them. All things considered, the result of developing concurrent systems with Alvis is an easy to understand model with properties verified formally. Moreover it creates an open environment that allows using other tools based on system state graph semantics. The only thing that is required is an appropriate export function in Haskell.

This paper addresses the problem of executable Alvis models by using virtual machine. The paper presents binary code organisation and the virtual machine design and operation. The presented solution allows execution of a formal Alvis models.

The paper is organised as follows. Section II provides an overview of developing formal models with Alvis and associated tools. The AVM design assumptions and operation are presented in section III. The binary code organisation is described in section IV. Section V deals with details about all the AVM instructions. The paper is summarised in the final section.

II. Alvis Environment

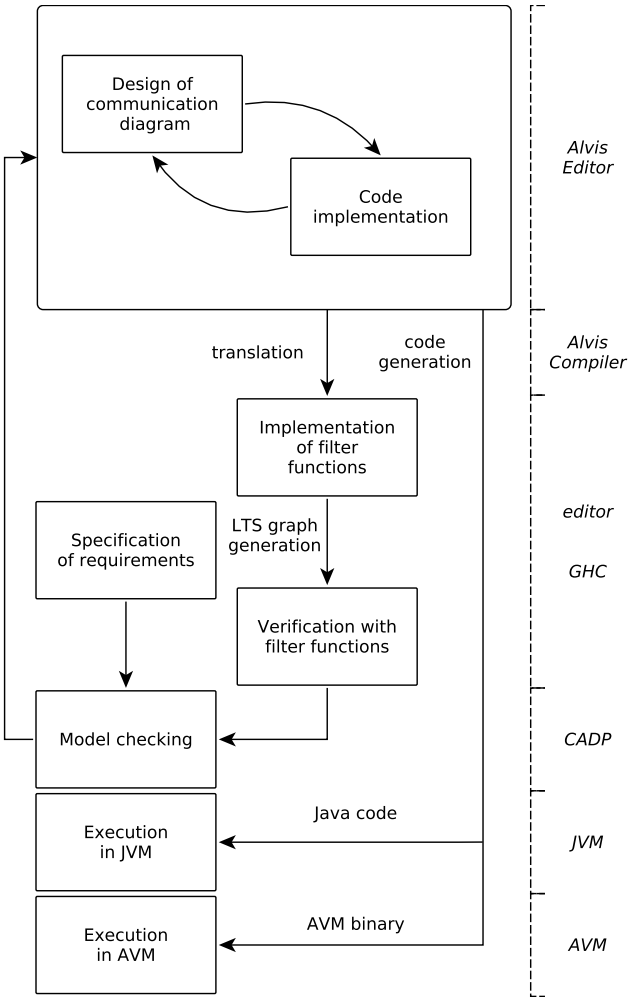

Figure 1. The modelling and the verification process with Alvis

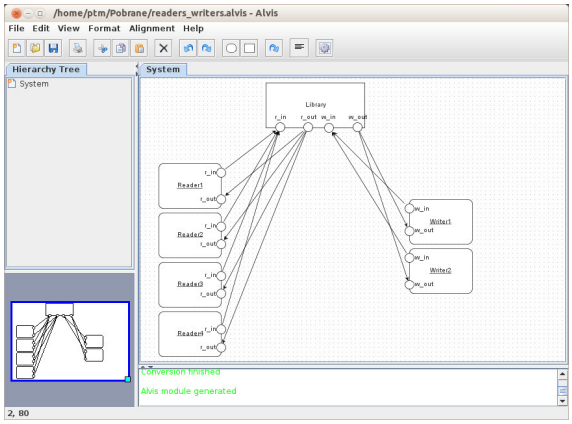

Figure 2. Alvis Editor

The scheme of the modelling, verification and execution process with Alvis is shown in Fig. 1. From a user's point of view, the process starts from designing a model using a prototype modelling environment called *Alvis Editor* (shown on Fig. 2). A system can be designed using a hierarchical representation of Alvis model. With *Alvis Editor* user can collapse and expand any valid part of a system back and forth. It is a very useful feature from the human point of view. However, before applying transformation methods, it has to be flattened. This Alvis system representation proved to be the best one for machine processing.

It is worth mentioning, that from the very beginning of Alvis all model transformations meant to be fully automatic. It was so to remove any "ideas" that user may introduce to system during human-powered translation.

A flat model is translated into Haskell [10], [4] source code and its Haskell representation is used to generate the LTS graph. A designer is able to define additional Haskell functions (called *filtering functions* [4]) that search an LTS graph for some states or parts of the graph that meet given requirements. The source code is compiled with the GHC compiler [11]. The results of the received program execution are the LTS graph for the given model and the report of the model verification with filtering functions. Further verification is performed with the CADP Toolbox [6] and the μ calculus [7], [8] or XTL [9]. Furthermore other tools that support LTS system specification might be used with little effort from the user.

For testing and educational purposes LTS graph can be exported to the DOT format and visualized. It is a useful feature during learning to model with Alvis. Unfortunately, it can be used only to fairly small systems (in terms of generated state space).

Alvis model can also be translated to an executable form. Currently Java target and a dedicated virtual machine, presented here, are supported. As it is shown on Fig. 1 it is not required for a model to be formally validated before execution. Any syntactically correct one can be translated to executable form, however it is wise to perform this step.

For more details on the Alvis syntax see [1] and the on-line manual at the project web site. The formal semantics for Alvis can be found in [2].

III. AVM DESIGN AND OPERATION

Alvis Virtual Machine was designed to run Alvis models without modifying its structure. Common problem in using formal methods in real cases is translation from a model to code. Even properly designed system can be ruined during implementation phase. Thus an *executable specification* concept was introduced and becomes more and more popular in different applications and forms f.e. [12], [13]. The whole Alvis project is a part of that conception. It brings a modelling environment and, by generating LTS of the possible system states, it provides ability to verify formally given system with tools that operate on a such representation [6].

As it will be presented in section V, AVM instruction set almost completely reflects Alvis language statements. The main goal here is to provide identical execution paths as generated in LTS graph [4]. AVM can be considered as a high level virtual machine. It has a very complex commands and is more like BEAM Erlang VM than JavaVM [14], [15]. Also the architecture of the Alvis Virtual Machine can be classified as register VM. Most of the operations are performed "in place" with variable location treated as operational registers.

Like in most virtual machines, the common operation that precedes execution, is code loading. In AVM this procedure prepares binary code for execution.

The preliminary step is checking cryptographic keys. It is based on public-private key pair. A code is signed with private key. The machine has a public key of the software supplier.

If the cryptographic signature is included in code it will be checked before execution. If it passes, the next steps for preparing code for execution will be performed, otherwise the loading will be abandoned.

There are two strategies being considered for the loader. First one, is for devices with large RAM pool. In that case, the whole binary code is loaded to RAM. It is very simple yet effective. There is no need to copy initial values for variables. All offsets are relative to the beginning of a code or the beginning of a given block.

The second code loading strategy is for devices with limited RAM capacity and FLASH memory like embedded SoC or microcontrolers. In that case a code resides in program memory alongside with AVM itself and before execution an additional step has to be performed.

Data field of the variable is moved to the RAM and an additional record is created to translate original address (in ROM/EEPROM) to the actual location. Unfortunately, this slows the execution which is the price for decreasing RAM occupancy. During execution a function which fetches variable, its pointer is modified by additional address translation step. The direct and indirect variable fetching is presented on Fig. 3.

To support online code upgrade AVM may use double code buffer. If the device is powerful enough the new code may be loaded during executing the old one.

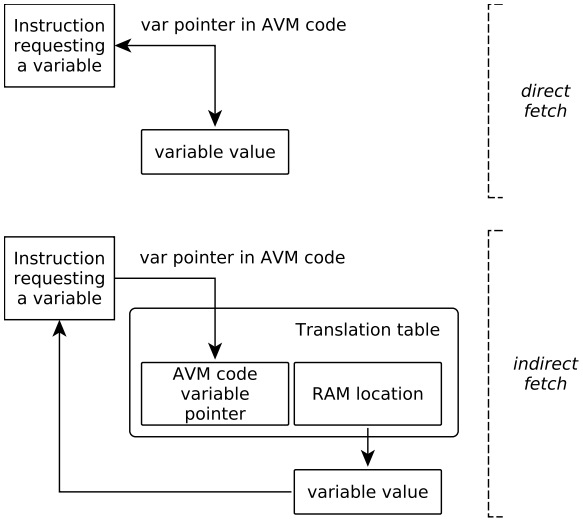

Figure 3. Direct vs indirect variable access

The machine performs all the necessary steps of preparing code for execution while executing the current one. It will be run as a background task. When new code will be ready, machine will restart and begin executing a new one.

Current AVM implementation is a bytecode interpreter. It fetches instruction from specified locations and runs them. Every AVM instruction has a function associated within virtual machine, responsible for its execution.

If a block is executed by VM, a *next instruction after block* (see block description in Section IV) is placed onto agent's *return stack*. When it reaches *block end*, this address is popped and execution continues from that location. In term of efficient virtual machine design and implementation it is suboptimal, but the main goal here is to reflect original model behaviour.

Dynamic memory management is minimized in AVM to simplify code execution and increase reliability. The only dynamic data structures required during execution are:

- stacks for agents for tracing block processing,
- data fields if code is run form ROM,
- address translation table if code is run from ROM,
- temporary storage for guard evaluation.

IV. Code format

In this section a detailed binary code organization for the Alvis Virtual Machine is presented. It is organized in top-down fashion, starting form overall description and ending with details about subsequent elements. The size information is presented as bytes.

Two byte word was chosen as a primary data size for AVM. Thus limits the maximal code size to 2^{16} bytes, including all extensions. As AVM is a high level virtual machine and its instruction are complex, it is enough for fitting quite large systems in its address space.

Table I presents general organization of the AVM code. First is the *header* block, then *functions* block. After it, agent's data are placed sequentially. The size of every block depends on the model.

Table I
General AVM binary code organization

Header	Functions	Agents	Crypto

The AVM *header* block is presented in Table II. It starts with a magic number, which is "AVM " in ASCII code. Next element is a version number. Virtual machine cannot load the code if magic word is incorrect or version number is higher than it can understand.

The function block offset represents displacement from beginning of the AVM binary code to the first function (see Sec. III). The last two elements describes agents in model. First one is an agent counter and the second is a table with offsets to a specific agent structures.

Table II
AVM Header

Name	Description	Size
MAGIC	Magic number 'AVM '	4
VERSION	Code version	4
FUN	Function block offsets table	2
ACNT	Size of the agents table	2
AGENTS	Agents block offsets table	2
SECURITY	Cryptographic extension block offset	2

The agent block is shown in Table III. It consists of: agent's name truncated to 12 characters, mainly for debug purposes, agent's code offset from beginning of AVM code, location of port and variable definitions. The overall single agent code organization is presented in Table V. The *STATE* field contains actual code pointer, execution block and agent state information (see [2]).

Table III
Active agent header

Name	Description	Size in bytes
NAME	Agent name	12
STATE	Agent state	8
CODE	Agent code block	6
PCNT	Ports count	2
PORTS	Ports table pointer	2
VCNT	Variables count	2
VARS	Variables table pointer	2

Table IV
Variable block

Name	Description	Size in bytes
NAME	Variable name	12
TYPE	Variable type	2
LOCATION	Value pointer	2

Table IV presents variable structure. The variable table consists of such elements. The whole variable block combines variable table and variable values. NAME and TYPE are left here mainly for debug reasons and their removal is considered to reduce code size.

Table V
AGENT BINARY CODE ORGANIZATION

Agent header	Ports	Variables	Code

The active port structure is presented in Table VI. This element is generated for every port's data type pair. It holds port's name type identifier for transferred data and pointer for the other side port structure.

Table VI
ACTIVE PORT BLOCK

Name	Description	Size in bytes
NAME	Port name	12
TYPE	Type of data	2
CPORT	Connected port	2

Passive agent definition is in Table VII. It is almost identical to active agent. The only difference is it holds passive ports definitions instead of active ones.

Table VII
PASSIVE AGENT BLOCK

Name	Description	Size in bytes
NAME	Agent name	12
STATE	Agent state	2
PCNT	Ports count	2
PPORTS	Ports table pointer	2
VCNT	Variables count	2
VARS	Variables table pointer	2
CODE	Code pointer	2

Table VIII
PASSIVE PORT BLOCK

Name	Description	Size in bytes
NAME	Port name	12
TYPE	Type of data	2
CODE	Port's code block	6

Table IX presents security extension block. It consists of: key type, key data length and key location as an offset from code beginning.

Key type specifies how the key data should be interpreted during code loading. Virtual machine which is not able to process a specific key type, should refuse to execute the code. Key type may by set to NULL value. In that case, no code verification is performed and key length and value pointer should also be set to NULL.

Table IX
SECURITY EXTENSION BLOCK

Name	Description	Size in bytes
TYPE	Key type	2
SIZE	Key length	2
LOCATION	Value pointer	2

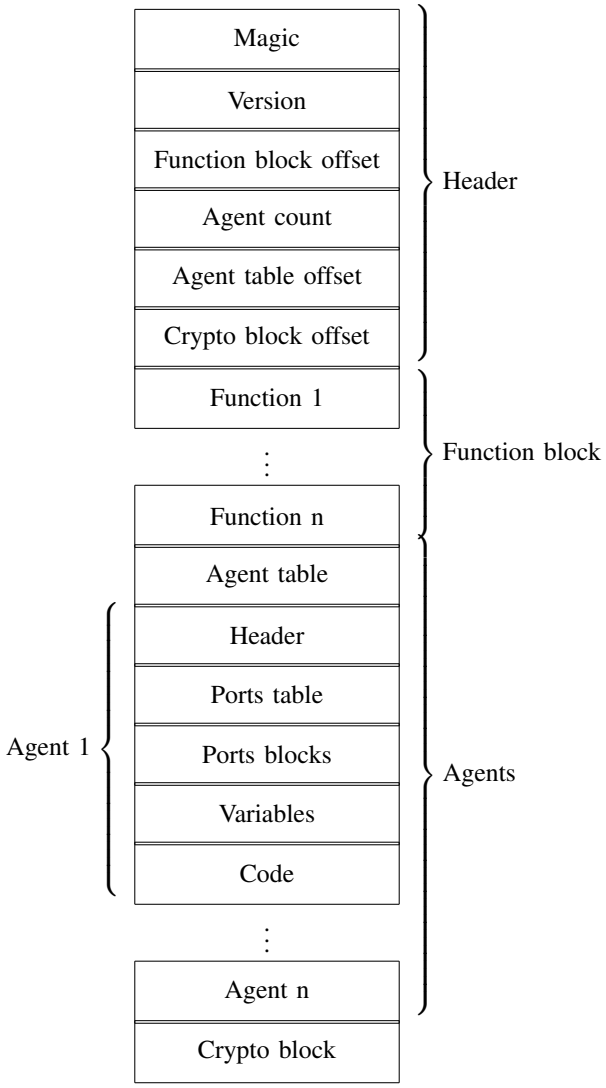

Figure 4. Example binary code map

Figure 4 shows an overview of the complete AVM binary code. The header is presented in detail. Then function block is placed. After it is an agent table. Then all the agents are located. The final block is occupied by cryptographic extension. This block location is not crucial for the AVM operation. It represents actual block placement but it can be reorganized. It is because most of the offsets are relative to the beginning of the AVM code (magic word location). This makes code generation more complicated because, in fact,

some linking operations are applied during that phase. An advantage of that approach is faster code loading and easy code execution. No linking is required during code loading. Also, multiple memory access is not required to access a specific structure nor additional memory to cache frequently used object locations.

V. INSTRUCTIONS

Below all the instructions supported by the Alvis Virtual machine are presented. All of them are shown as a small tables where there is instruction and type parameters (if any) in the first column, and short description in the second column. Binary format of the instruction is equal to the first column contents.

Instructions were divided into two blocks. The first one consists of instructions present in Alvis itself, whether the second one is composed of instructions added to allow Alvis models to be executed on small embedded systems with limited resources, in case it is impossible to fit native Haskell code.

Haskell is not a problem during generation of LTS graph because its already there. Executing even simplest functions as an arithmetic operations in Haskell is perfectly reasonable. But in embedded environment Haskell brings a huge overload to smaller systems. In such case, using it to do a few additions do not seem to be a good idea.

Moreover, during examination of already created test models during Alvis development, this subset allows for running quite large set of code without involving Haskell binaries.

The following types of arguments are used in AVM binary code:

- agP agent pointer - offset to specified agent structure;
- fP function pointer - offset to the start of specific function;
- vP variable pointer - offset to specific variable;
- pP port pointer - offset to specified port structure;
- ppP passive port pointer - offset to specified port structure;
- cP code pointer - offset to specified instruction;
- bs block specification - it consists of three code pointers (block start, block end, first instruction to execute after block);
- char 8 bit signed character;
- uint32 32 bit unsigned integer;
- int32 32 bit signed integer;
- double 64 bit double precision floating point number;
- list list of any of the simple data types (char, uint32, int32, double);

All offsets are relative to beginning of the AVM binary code. It speeds up code execution by allowing to fetch specific data without multiple memory access. Moreover, it simplifies a ”virtual memory” (see Fig. 3) implementation for devices where code lays in read only area. In this case, some portions of the data have to be moved to random access memory which is required for execution. This design feature complicates a bit code generation, but it is a consequence of execution requirements for small devices.

NULL	do nothing

The *NULL* instruction does nothing. However, it increments agent's program counter and consumes some time during execution.

START	start agent
agP	agent pointer

The *START* instruction begins execution of agent pointed by the first argument. Its state is changed from *initialized* to *ready*.

EXIT	stop agent
agP	agent pointer

The *EXIT* instruction stops execution of agent pointed by the first argument. Its state is changed to *finished*.

EXEC	execute function
fp	function pointer

The *EXEC* instruction executes specified function. Function arguments are hard-coded inside so there is no need for passing them.

NEXEC	native exec
nfP	native function pointer
uint32	argument count
vP	result pointer
vP	first argument pointer
...	...
vP	n-th argument pointer

The *NEXEC* instruction is a wrapper for executing natively implemented functions. It requires pointers for all arguments and for result. Also types of AVM arguments have to match natively implemented function. Function code has to be compiled and linked with AVM.

IF	start agent
fP	guard pointer
bs	true block specification

The *IF* instruction is the simplest implementation of the *if* Alvis statement. It is the case when there is no *elseif* or *else* clause. If a guard function evaluates to *true*, it executes a code block, otherwise the next instruction specified by *bs* is selected.

IFE	if-else instruction
fP	guard pointer
bs	true block specification
bs	false block specification

The *IFE* instruction covers the case when there is an *else* statement in Alvis code. If its value is interpreted as true, its first block is executed, otherwise second block is executed.

IFEIFE	select instruction
uint32	branch count
fP	first guard pointer
...	...
fP	n-th guard pointer
bs	first branch block
...	...
bs	n-th branch block
bs	else branch block

The *IFEIFE* is the most complicated version of the *if* statement in Alvis. It is the case when there is *if-elseif-else* form of the statement. It consists of table of guards and a table of connected code blocks to execute. The last code block is an else branch one and it is executed when all guard functions evaluate to *false*. There is also the *IFEIF* virtual machine instruction. The only difference from *IFEIFE* is it has no else branch.

The *if* statement was split into several cases because during example code analysis, the most commonly used statement was *if* or *if-else* one. It was made to optimize VM instruction execution and to make implementation clearer.

LOOP	conditional loop
fP	guard pointer
bs	block specification

The *LOOP* instruction executes specified block as long as associated guard function evaluates to true. Otherwise, the next instruction denoted by *bs* is used.

LOOPE	timed loop
uint32	dalay value
bs	block specification

The *LOOPE* is a special loop instruction. It loops indefinitely, but each iteration should start every *delay value* milliseconds. To achieve the process a time stamp is taken before code block execution. After it, another time stamp is taken and the remaining time is calculated. If there is some time left, agent suspends its execution.

JUMP	unconditional jump
cP	code pointer

The *JUMP* instruction performs an unconditional jump to specified code location.

IN	input from port
vP	input variable pointer
pP	local port pointer
pP	remote point pointer

The *IN* instruction performs communication with other agent via port specified. It requires a variable structure pointer to store a new value, local port structure and remote port structure.

OUT	output to port
vP	output variable pointer
pP	local port pointer
pP	remote point pointer

The *OUT* instruction performs communication with other agent via port specified. It requires a variable structure pointer to send value, local port structure and remote port structure.

INP	input from passive port
vP	input variable pointer
pP	local port pointer
ppP	remote point pointer

The *INP* instruction performs passive agent call via specified port. The caller is an active agent. It requires a variable structure pointer to save new value, local port structure and remote passive port structure.

OUTP	output to passive port
vP	output variable pointer
pP	local port pointer
ppP	remote point pointer

The *OUTP* instruction performs passive agent call via port specified. The caller is an active agent. It requires a variable structure pointer to send value, local port structure and remote passive port structure.

INPP	input from passive port
vP	input variable pointer
ppP	local port pointer
ppP	remote point pointer

OUTpP	output to passive port
vP	output variable pointer
ppP	local port pointer
ppP	remote point pointer

The *INPP OUTPP* instructions perform passive agent call via port specified. The caller is a passive agent. They require a variable structure pointer to send or save to, local port structure and remote passive port structure.

SELECT	select instruction
uint32	branch count
fP	first guard pointer
...	...
fP	n-th guard pointer
bs	first branch pointer
...	...
bs	n-th branch pointer

The *SELECT* instruction reflects Alvis's *select* statement. It consists of a table of guard functions and a table of code blocks. Guards are evaluated sequentially. If guard evaluates to *true*, a corresponding code block is executed.

READY	check if port is ready for communication
pP	port pointer

PREADY	check if passive port is ready for communication
ppP	passive port pointer

The *READY* and *PREADY* instructions check if a specified port is ready for communication. It is required for guard functions mainly in select statement.

Table X
INTERNAL COMMANDS FOR FUNCTION EXECUTION

ADD	add operands
SUB	subtract operands
MUL	multiply operands
DIV	divide operands
HEAD	get head of a list
TAIL	get tail of a list
INS	insert element in list at the beginning
ADD	append element to list
AND	logical and
OR	logical or
NOT	logical not

Table X presents summarized list of AVM commands implemented for internal functions. Almost all of them take three arguments except for the last one which takes two. The binary layout of instructions is presented below.

{INSTR}	instruction code, see Table X
vP	result
vP	first operand
vP	second operand

All the operations are defined for appropriate datatypes. Arithmetical instructions are automatically applied to all simple types. Conversions are executed as in ISO-C standard [16].

Presented AVM instruction list should not be considered as closed. AVM is in active development phase and commands are added, removed and reorganized.

VI. Summary

The Alvis Virtual Machine was presented in the paper. The main goal of the project is to provide executable form of an Alvis models. There is an Alvis to Java conversion already done, which was a test drive for Alvis Compiler code generation facility. AVM is a second attempt for automatic Alvis code execution.

AVM was designed for executing a formally checked code in high availability environment. Thus a code signing and a double code buffer were introduced in it.

The first feature is crucial for upgrading AVM code in installations where the software provider has no full control over device and running unauthorized code is a security risk.

The second feature is required in situations when device has to work continuously while providing ability to hot code swapping.

AVM is currently under development and presented features may be a subject to a change.

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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21271-0_15

[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.

[3] M. Szpyrka, P. Matyasik, and M. Wypych, "Alvis language with time dependence," in *Proceedings of the Federated Conference on Computer Science and Information Systems*, Krakow, Poland, 2013, pp. 1607–1612.

[4] ——, "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.

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

[6] 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*, ser. LNCS, vol. 4590. Springer-Verlag, 2007, pp. 158–163. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73368-3_18

[7] 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.

[8] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free μ-calculus," INRIA, Tech. Rep. 3899, 2000. [Online]. Available: http://dx.doi.org/10.1016/S0167-6423(02)00094-1

[9] R. Mateescu and H. Garavel, "Xtl: A meta-language and tool for temporal logic model-checking," 1998.

[10] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. Sebastopol, CA, USA: O'Reilly Media, 2008. [Online]. Available: http://dx.doi.org/10.1145/1668113.1668115

[11] "Glasgow Haskell Compiler documentation," http://www.haskell.org/haskellwiki/GHC.

[12] G. Cancro, W. Innanen, R. Turner, C. Monaco, and M. Trela, "Uploadable executable specification concept for spacecraft autonomy systems," in *Aerospace Conference, 2007 IEEE*, March 2007. doi: 10.1109/AERO.2007.352802. ISSN 1095-323X pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/AERO.2007.352802

[13] A. Khwaja and J. Urban, "Realspec: An executable specification language for modeling control systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on*, March 2009. doi: 10.1109/ISORC.2009.36. ISSN 1555-0885 pp. 219–227. [Online]. Available: http://dx.doi.org/10.1109/ISORC.2009.36

[14] "Erlang documentation," http://www.erlang.org.

[15] "Oracle Java documentation," http://java.oracle.com.

[16] ISO, "Iso c standard 1999," Tech. Rep., 1999, iSO/IEC 9899:1999 draft. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf