An Approach to Test Program Generation for Memory Coherence Verification of ”Elbrus” Microprocessors

Vladimir Agafonov[1,2], Pavel Frolov[1,2,3], Alexey Meshkov[2,3]
[1]*Moscow Institute of Physics and Technology (National Research University)*
[2]*MCST*
[3]*INEUM*
Moscow, Russia
E-mail: {Vladimir.A.Agafonov, Pavel.V.Frolov, Aleksey.N.Meshkov}@mcst.ru

***Abstract*—One of the key aspects of the correctness of the memory subsystem of a microprocessor is its functioning in accordance with the memory coherence protocol. This article presents an approach to test program generation for memory coherence verification of ”Elbrus” microprocessors. Requirements for memory coherence tests are considered. The memory map structure allowing to describe the memory areas used in tests and the types of accesses to these areas in a flexible way is presented. The method of test program generation based on the memory map structure is described. The method of automatic memory map generation is proposed. Generated tests have been used for verification of RTL models and FPGA-based prototypes.**

***Index Terms*—system verification, memory coherence verification, pseudorandom test generation, Elbrus**

I. Introduction

Memory subsystems of modern microprocessors provide support for various address spaces with address translation and include various levels of cache memory, means of ensuring data coherence, numerous buffers and switches [1]. To ensure consistency of caches states, the computing nodes of the system are combined into a single system and exchange messages in accordance with a cache coherence protocol [2]. All this determines high combinatorial complexity of verification [3], which sharply restricts the use of formal methods to individual bottlenecks or devices.

To increase the probability of reproducing various dynamic situations that occur when many devices are functioning simultaneously in the system, increasing the number of tests is necessary. In order to automate the development of test scenarios and their implementations in the form of ready-made test programs, automatic test generation is actively used. In this case, the source code of the test program is generated randomly, considering the specified parameters [3]. Parameterization allows fitting tests to reproduce certain situations with certain sets of random parameters [4]. The selection of test generation control parameters is performed at the development stage of each tool separately and is determined by the test generation algorithm.

Currently, MCST JSC is designing multi-core microprocessors with general-purpose cores of ”Elbrus” architecture version 6. The development of ”Elbrus” architecture has led to significant additions to the instruction set architecture and changes in the operation of devices included in the memory subsystem. In particular, the transition to a new coherence protocol affected the functionality of existing memory accesses. This led to inapplicability of the existing test generation algorithm to verify the memory subsystem of the developing microprocessors. For these reasons, development of a new algorithm for generating memory coherence tests and its implementation in the form of a pseudo-random Assembly test generator was required. This article discusses a new approach that is the basis for solving this problem.

II. ”Elbrus” architecture overview

”Elbrus” architecture introduces a set of memory types determining the system behavior of memory accesses. The system behavior of memory access is characterized by a combination of specified properties. Each memory type describes a unique combination of these properties.

Memory access instructions are represented by instructions of store and load types. The size of addressing memory fragment and the source/destination register format are determined by the memory access instruction format. ”Elbrus” architecture uses an operation code extension – memory address specificator (MAS) for each memory access instruction. MAS defines the additional specific properties of the memory access, the method for storing data in different cache levels and affects the memory type. Thus, the set of memory accesses types is determined by the following formula:

$$T \subseteq I \times F \times M \quad (1)$$

where:
T – set of memory accesses types,
I – set of memory access instruction types,
F – set of memory access instruction formats,
M – set of MAS.

According to the generally accepted classification "Elbrus" is a VLIW (Very Long Instruction Word) architecture [5]. Each VLIW contains a set of instructions. Instructions placed in the same VLIW are executed in parallel.

In "Elbrus" architecture VLIW consists of 6 instruction channels. The sets of instructions supported by each channel are different. Memory access instructions are supported by only 4 instruction channels. However, store instructions are supported by only 2 of these channels. The use of certain memory access types is moreover supported by only certain channels. Due to architectural restrictions, the placement of several instructions in the same VLIW is limited.

Modern "Elbrus" microprocessors are Systems-on-a-Chip (SoC) with multiple unified general-purpose cores. The unified general-purpose core includes private level 1 instruction cache (L1I), level 1 data cache (L1D) and level 2 cache (L2). SoC of various configurations are being developed. The number of unified general-purpose cores, the presence of shared level 3 cache (L3), the number of interprocessor communication channels (IPCC), the number of memory controllers and other options are defined by the SoC configuration. Multiple microprocessors can be combined in a multiprocessor system with coherent shared memory based on ccNUMA principle by IPCC.

Among the innovations of "Elbrus" instruction set version 6 can be noted: the introduction of additional memory access properties, the transition to new memory types, the introduction of additional MAS with extended caching hints and the elimination of Input-Output memory space coherence support. In addition, hardware innovations include the transition to a unified general-purpose cores, cache memory policies modification and optimization for unaligned memory accesses. The coherence protocol has also been significantly changed.

III. Test requirements

The following requirements for generated tests were formed.

Since "Elbrus" microprocessors contain multiple cores and can be combined in a variety of multiprocessor configurations, memory coherence tests should be designed for verification of multiprocessor systems. Before the execution of a test sequence, the system under test should be initialized. In order to check different operating modes of memory subsystem devices and cover more dynamic situations, the implementation of pseudo-random initialization of memory subsystem devices settings within acceptable limits is necessary. The system initialization procedure should end with a cores synchronization procedure to ensure that all of the cores of the system under test are ready for execution of a test sequence.

The existing test development environment provides a unified parameterized system initialization program and implementations of commonly used test procedures, such as cores synchronization and exit code output to the using test bench. Using the test development environment is a great way to simplify test development.

After the exetution of the system initialization procedure each core executes a test sequence of instructions. In the case of memory subsystem verification tests, the test sequence should contain memory access instructions of various memory access types and perform the operation of cores with shared memory. In order to check the functionality of individual cache memories, the organization of memory access sequences at addresses that lead to cache lines eviction is necessary. To test the memory coherence mechanisms, a stream of parallel requests to the same cache memory lines from different cores should be formed. In addition, the generation of VLIWs with various combinations of memory access instructions needs to be supported in order to check the VLIWs execution.

The test sequence of instructions for each core should end with a self-check procedure that checks the correctness of register values and data in the tested memory areas. This approach allows using tests to verify RTL models, FPGA-based prototypes [6], and to test manufactured chips. The intention is to use self-checking code generator for the self-check procedures generation. This tool is based on the functional model of the system under test and allows to generate the code for comparing registers with the reference values obtained as a result of the test execution by the functional model [7] [8].

Debugging of generated tests is supposed to be performed using a functional model of the system under test and a trace comparator. The results of the test execution on the functional model and RTL model are the executed instructions traces. Firstly, to achieve a successful test execution on the functional model is necessary. This is followed by debugging the test on RTL model. To speed up the search for differences between the functional model and RTL model executed instruction traces the trace comparator should be used. Due to the unavailability of the execution instruction trace, debugging tests on the FPGA-based prototype is difficult. For this reason the test exit code should localize the failed self-check code statement. The reasons for the test fail can be both the test errors and the system under test errors.

IV. Shared memory interaction

The operation of multiple cores with shared memory allows using maintain memory coherence mechanisms and detecting errors in their implementations. There are two ways of sharing memory: *true sharing* and *false sharing*. Both of them are presented on Fig. 1.

True sharing is the operation of multiple cores with overlapping memory fragments. The implementation of true sharing in tests is limited due to the need to ensure deterministic test execution. To ensure determinism of the true sharing usage, synchronization of code execution by the cores before and after each modification of the overlapping memory fragments is necessary. Since the cores synchronization procedure requires multiple memory accesses from each of the synchronized cores and blocks the test sequence execution, using of true sharing can lead to high overhead costs.

False sharing is the operation of multiple cores with non-overlapping memory fragments. One of the cores is designated for each memory fragment – the core-owner. Only the core-owner can operate with its own memory fragments. Since

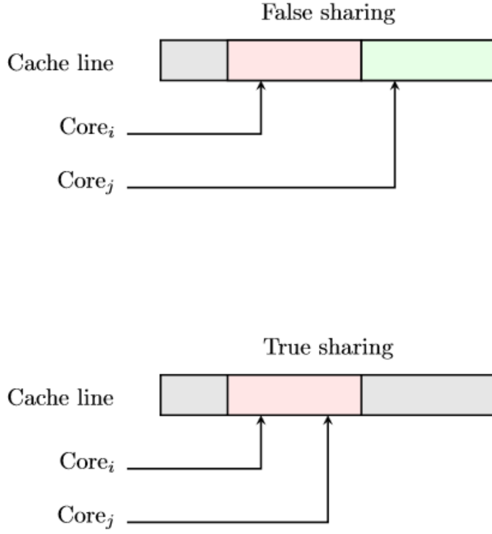

Fig. 1. Methods of memory sharing.

cache lines change their states according to the coherence protocol, this way of sharing memory allows loading memory coherence mechanisms without the overhead of ensuring the determinism: multiple cores use the same cache line, however, the order of memory accesses of the core-owner is guaranteed by the hardware according to the used memory type and other cores do not affect the content of the memory fragment.

In comparison with true sharing, false sharing does not provide real operation with shared data. Thus, some ways of changing the cache lines states are not checked. In addition, using true sharing can change the dynamics of the test execution.

For these reasons, the combination of both approaches was proposed: mainly to organize data separation in tests by false sharing, but at the same time to implement operation with common data in some volume. This is achieved by forming a testing sequence from a parameterized number of sections. The full test structure is shown in Fig. 2. At the beginning of each section, all of the cores are synchronized. After synchronization procedures, a random sequence of VLIWs with different combinations of memory access instructions is generated for each core. Within each section of the test sequence, cache lines are split between the cores using false sharing in different ways. Therefore, when switching between test sections occurs, some memory fragments are passed to other cores-owners for management. Thus, the operating with shared memory fragments by several microprocessor cores is implemented in cores synchronization procedures and during transitions between the test sections.

V. Description of memory accesses

A special structure – a memory map has been developed to describe memory accesses. The memory map contains a list of address ranges corresponding to memory fragments and maps the memory fragments to their parameter sets:

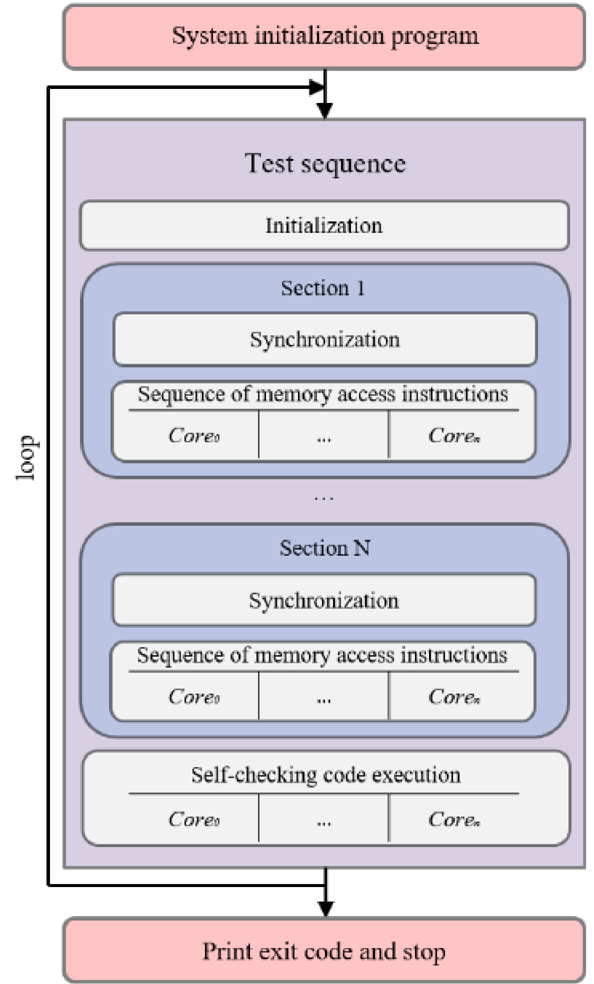

Fig. 2. Test structure.

$$[a_k^{begin}, a_k^{end}] \mapsto \{I_k, F_k, M_k, c_k, r_k^{st/ld}, p_k^u\}, k = \overline{1, K} \quad (2)$$

where:
a_n^{begin} – the address of the nth memory fragment beginning,
a_n^{end} – the address of the nth memory fragment ending,
I_n – the nth set of memory access instruction types,
F_n – the nth set of memory access instruction formats,
M_n – the nth set of MAS,
c_n – the nth core-owner,
$r_n^{st/ld}$ – the nth store-to-load ratio,
p_n^u – the nth priority of use,
K – the number of the memory fragments in the memory map.
All addresses are presented in terms of bytes. The memory map describes non-overlapping memory fragments:

$$a_1^{begin} \leq a_1^{end} < a_2^{begin} \leq a_2^{end} < \ldots < a_K^{begin} \leq a_K^{end} \quad (3)$$

Adjacent memory fragments with equivalent parameter sets are combined into a single memory fragment in the memory

map. Priorities of use memory fragments allow managing the frequency of memory accesses to each memory fragment, as well as store-to-load ratios control the frequency of using different types of memory accesses. This allows to configure spatio-temporal profiles of cores with different memory areas in a flexible way [9].

The need to ensure a deterministic state of memory imposes restrictions on the combination of using memory access types. Using coherent and non-coherent memory accesses to the memory fragments located in the same cache line generally results in an undefined cache line memory value. For this reason, the set of valid memory access types for each memory fragment should be selected for deterministic reasons.

A memory map is used to describe the requests generated in a test section. For multi-section tests, a separate memory map for each section of the test should be used. In this case, ensuring deterministic transitions between the test sections by adding additional procedures to the test is necessary.

VI. Code generator

The code generator implements the described in the memory maps memory accesses as a ready-made Assembly test in accordance with the test structure presented in Fig. 2.

At the beginning of the test, the code generator defines initialization parameters for the parameterized system initialization program. Depending on the test generation parameters, default or random settings for memory subsystem devices can be used.

The test sequence is executed in a loop. The number of iterations of the loop is parameterized. Due to the operation of the cache memory, different dynamic of the test program execution at different iterations is achieved. Consequently, with a slight increase in the size of the test, the execution time and the number of situations being tested can be increased many times. This is convenient when performing a test on an FPGA-based prototype, where the test load time is significant.

Memory access instructions are generated based on the memory map of the current test section. The algorithm for generating a test sequence using a memory map runs independently for each core. To check the correctness of the store-to-load bypass in the L1D cache, random generation of the load instruction corresponding to the previous store instruction by address, format and MAS is provided. Random generation of "wait" instructions is also implemented. For core c the algorithm consists of the following steps:

1) Retrieving the set A of memory fragments owned by core c;
2) An empty VLIW creating;
3) Constructing a memory fragments probability distribution based on the priorities of use p^u for memory fragments $a \in A$;
4) If bypass-load is scheduled to be inserted into this VLIW, restore the memory access type t, the memory access address $\widetilde{a}$ and go to step 13;
5) With specified probability of wait instruction generation randomly choose an instruction from the set of acceptable wait inctructions and go to step 14;
6) Random choice of the memory fragment $a_m \in A$ according to the constructed memory fragments probability distribution;
7) Constructing a store-to-load probability distribution based on the store-to-load ratio $r_m^{st/ld}$;
8) Random choice of the instruction type $i \in I_m$ according to the constructed store-to-load probability distribution;
9) Retrieving the set $\widetilde{F_m} \subset F_m$ of suitable memory access instruction formats $f : size(f) \leq size(a_m), \forall f \in \widetilde{F_m}$;
10) Retrieving the set T_m of suitable memory access types using (2) in accordance with the requirements of the instruction set architecture:
$$T_m \subset i \times \widetilde{F_m} \times M_m \quad (4)$$
11) Random choice of a memory access type $t \in T_m$;
12) Random choice of the memory access address $\widetilde{a} \in a_m$ considering the memory access alignment and the borders of a_m;
13) Selecting the data register for the generated memory access instruction;
14) Placing the generated instruction in the current VLIW or in a new empty VLIW if the placing in the current VLIW is impossible;
15) If i is store, with the specified probability of store-to-load bypassing save the memory access address $\widetilde{a}$, the memory access type t and randomly choose the VLIW to bypass-load insertion based on the specified VLIW skipping range;
16) Repeating steps 4-16 until the specified number of memory access instructions is reached.

The register and memory random values initialization code is placed in the beginning of the first section of the test for each core. In this case, a prohibition is introduced on the generation of load instructions, only uninitialized memory fragments are written with store instructions of the maximum possible format.

The mode of generation of unaligned addresses is optional, its use is determined by the generation parameters of the test.

Code generation for each test section is performed independently using different memory maps. The code generator ensures the correctness of the transition between test sections. If the memory access types of a cache line are changed from coherent to non-coherent, the cache line should be flushed out of all the caches when switching between the test sections. Before starting a new test section, the code generator places the cores synchronization procedure and a sequence of cache line flush instructions for such cache lines. Changing the memory access types of a cache line from non-coherent to coherent does not violate determinism; therefore, no additional code is generated in this case. As a result of using this approach, the functionality of evicting the specified cache lines is additionally checked.

The final step of the test is checking whether the register and memory values match the reference values. The self-checking code is generated independently for each core. The memory map of the last section of the test is used as a description of requests. In this way, each core checks its own memory fragments. If a discrepancy between the test data and the reference data is detected, the test execution is terminated with the output of diagnostic information to the user (exit code or debug printing when executed on an FPGA-based prototype).

VII. Automation of the memory maps formation

Memory maps are a large and detailed description of memory accesses. Manual compilation of memory maps requires high labor costs. In practice, describing memory requests in high detail, up to fragments, is not always necessary. In order to minimize the effort involved in creating memory maps for large memory areas with the same allowed memory access types, this process has been automated.

The user is given the opportunity to describe memory maps at a more general level: the description is made for arbitrary memory areas. In accordance with each memory area A_k the parameter set are placed:

$$A_k \mapsto \{I_k, F_k, M_k, C_k, S_n, r_k^{st/ld}, p_k^u\}, k = \overline{1, K} \quad (5)$$

where:
I_n – the nth set of memory access instruction types,
F_n – the nth set of memory access instruction formats,
M_n – the nth set of MAS,
C_n – the nth set of cores-owners,
S_n – the nth set of memory fragment sizes,
$r_n^{st/ld}$ – the nth store-to-load ratio,
p_n^u – the nth priority of use,
K – the number of described memory areas.
No restrictions are imposed. The resulting parameter set of overlapping memory areas is a union of the overlapping memory areas parameter sets. All conflicts will be resolved automatically. If the conflicts cannot be resolved, test generation ends with an error message.

Then the symbol A is used for the described memory areas set. The algorithm for generating a memory map consists of the following steps:

1) Splitting of the memory area set A into K fragments $\widetilde{A_k} \in A, k = \overline{1, K}$ of cache line size and alignment with the parameter set inheritance as $\{\widetilde{I_k}, \widetilde{F_k}, \widetilde{M_k}, \widetilde{C_k}, \widetilde{S_n}, r_k^{st/ld}, \widetilde{p_k^u}\}$;
2) Analysis of the MAS sets $\widetilde{M_k}, k = \overline{1, K}$ for simultaneous presence of coherent and non-coherent types of MAS in M_k sets;
3) Elimination of coherent or non-coherent types of MAS in a random way for each M_k set, which contains coherent and non-coherent MAS simultaneously;
4) Splitting the fragments $\widetilde{A_k}, k = \overline{1, K}$ into fragments $a_n^k \in \widetilde{A_k}, n = \overline{1, N(k)}$ of $\widetilde{S_k}$ sizes in a random way, where $N(k)$ is a resulting number of memory fragments a_n^k in $\widetilde{A_k}$;
5) Mapping the memory map parameter set $\{\widetilde{I_k}, \widetilde{F_k}, \widetilde{M_k}, \widetilde{c_{nk}}, r_k^{st/ld}, \widetilde{p_k^u}\}$ with random core-owner $\widetilde{c_{nk}} \in \widetilde{C_k}$ to each memory fragment a_n^k.

The result of this algorithm is a memory map that describes the memory requests for a test section.

The memory maps are generated independently for each section of the test using the presented algorithm. Ensuring true sharing of memory and variations in request types used in different sections of the test is achieved due to the randomness of the memory map generation algorithm. The code generator provides determinism support during transitions between the test sections.

To check the functionality of individual caches, memory areas are selected based on the organization of the target cache. Requests to lines with the same indexes lead to evictions from the cache memory, and the number of lines used must exceed the associativity of the target cache memory. To automate the compilation of memory areas aimed at creating evictions in various cache memory levels, a memory areas generator has been developed. The memory areas generator allows generating a memory area that corresponds to a given number of lines of the target cache level with the same indexes and different tags.

VIII. Results

A method for describing memory accesses using a memory map was developed. The test generator of self-checking Assembly tests for memory coherence verification of "Elbrus" architecture microprocessors implementing the memory accesses described in the memory maps was developed in C++.

The memory areas generator and the memory maps generator have been developed for the convenience of creating memory maps that used for checking the functionality of individual cache memories. Automating the formation of memory maps allows reducing the volume and complexity of describing the memory maps. In this case, the user can simultaneously use an arbitrary number of memory areas generators and manually specify memory areas. Moreover, the parameters for generating memory maps can be set separately for each obtained memory area. The implementation of the described tools for automating memory maps generation does not lead to the loss of the ability to describe requests at the fragment level, but in some cases significantly simplifies the process of configuring the test generator.

Currently, the developed test generator is used for verification of RTL models and FPGA-based prototypes of developing microprocessors with general-purpose cores of "Elbrus" architecture version 6. As a result of using the test generator, 74 logical errors were detected in the following hardware units: L1D-cache, L2-cache, L3-cache, Translation Lookaside Buffer (TLB), Memory Access Unit (MAU), On-Chip Network (OCN), Home Memory Unit (HMU), Memory Controller (MC), EFUSE. In addition, 3 malfunctions were found in the hardware of the FPGA-based prototype. These malfunctions are not errors of the original design and are specific only to the implementation of the FPGA-based prototype.

Logical errors were manifested in the following manner: data corruption, deadlock, RTL assertion failure (available only in RTL-simulation).

Conducting verification on the FPGA-based prototype allows using tests with a large number of the test sequence itterations (> 100) and, therefore, obtaining a variety of the test sequence execution dinamics, which is difficult to achieve with RTL-simulation. For this reason, some logical errors were found only during verification on the FPGA-based prototype. Detailed statistics on detected logical errors are presented in table I.

The plan for further development of the test generator consists of work in the following areas:

- Test scenarios development;
- Support for virtual addressing;
- Development of algorithms for verification of memory consistency.

TABLE I
STATISTICS ON DETECTED LOGICAL ERRORS

Unit	Number of logical errors	
	Detected on the FPGA-based prototype	***Total***
L1D-cache	2	19
L2-cache	1	10
L3-cache	2	10
TLB	0	5
MAU	0	13
OCN	0	4
HMU	0	6
MC	1	6
EFUSE	0	1

REFERENCES

[1] Ermakov S. G., Kim A. K., Perekatov V. I. Mikroprocessory i vychislitel'nye kompleksy semejstva "Elbrus", 2012 (in Russian).

[2] Lebedev D. A., Petrochenkov M. V. Test environment for verification of multi-processor memory subsystem unit". Trudy ISP RAN [Proc. ISP RAS], vol. 31, pp. 67–76, 2019.

[3] A. Adir [et al.]. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. IEEE Des. Test. 2004. V. 21, N. 2. pp. 84-93.

[4] Frolov P.V. Random System-Level Test Generation for Elbrus Architecture Microprocessors. Voprosy radioelektroniki, 2014, no. 3. (in Russian).

[5] Tanenbaum, Andrew S., Structured computer organization / Andrew S. Tanenbaum, Todd Austin. – 6th ed.

[6] Yurlin S.V., Bychkov I.N. FPGA prototyping for functional verification of multi-core processors. Problemy razrabotki perspektivnykh mikro- i nanoelektronnykh sistem (MES), 2014, no. 4, pp. 45–50 (In Russian).

[7] A.N. Meshkov, M.P. Ryzhov, V.A. Shmelev. The developement of the verification tools

[8] K. L. Gurin, A. N. Meshkov, A. V. Sergin, M. A. Yakusheva. Memory architecture development in the "Elbrus" series computer models. Voprosy radioelektroniki, 2010, no. 3. (in Russian).

[9] V. A. Agafonov. Ispol'zovanie karty pamyati pri generatsii sistemnykh testov podsistemy pamyati mikroprotsessora. Trudy 61-i Vserossiiskoi nauchnoi konferentsii MFTI. 19–25 noyabrya 2018 goda. Radiotekhnika i komp'yuternye tekhnologii, p. 21, 2018 (in Russian). of the Elbrus-2S microprocessor. Voprosy radioelektroniki, ser. EVT, 2014, no. 3, pp. 5-17 (in Russian).