DOI: 10.15514/ISPRAS-2019-31(3)-11

Extracting Assertions for Conflicts in HDL Descriptions

[1,2,3,4]A.S. Kamkin, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>
[1]M.S. Lebedev, ORCID: 0000-0002-0207-7672 <lebedev@ispras.ru>
[1]S.A. Smolov, ORCID: 0000-0003-0173-3081 <smolov@ispras.ru>
[1]Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2]Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia
[3]Moscow Institute for Physics and Technology,
9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia
[4]National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia

Abstract. Data access conflicts may arise in hardware designs. One of the ways of detecting such conflicts is static analysis of hardware descriptions in HDL. We propose a static analysis-based approach to data conflicts extraction from HDL descriptions. This approach has been implemented in the Retrascope tool. The following types of conflicts are considered: simultaneous reads and writes, simultaneous writes, reading of uninitialized data, no reads between two writes. Conflict assertions are formulated as conditions on variables. HDL descriptions are automatically translated into formal models suitable for the nuXmv model checker. The translation process consists of the following steps: 1) preliminary processing; 2) Control Flow Graph (CFG) building; 3) CFG transformation into a Guarded Actions Decision Diagram (GADD); 4) GADD translation into a nuXmv format. Conflict assertions are automatically built using static analysis of the GADD model and passed to the nuXmv model checker. Bounded model checking is used to check whether these assertions are satisfiable. If true, counterexamples are generated and then translated to HDL testbenches by the Retrascope tool. The proposed approach was applied to several open source HDL benchmarks like Texas-97, Verilog2SMV, VCEGAR and mips16 modules. Potential conflicts have been detected for all of these benchmarks. Future work includes propagation of conflict assertions to the interface level (thus getting assertions on modules' communication protocols) and generation of built-in HDL checkers.

Keywords: hardware design; hardware description language; functional verification; static analysis; test generation; data access conflict; control flow graph; guarded action; guarded actions decision diagram; model checking.

For citation: A.S. Kamkin, M.S. Lebedev, S.A. Smolov. Extracting Assertions for Conflicts in HDL Descriptions. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 135-144. DOI: 10.15514/ISPRAS-2019-31(3)-11

Поиск конфликтов доступа к данным в HDL-описаниях

[1,2,3,4] А.С. Камкин, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>
[1] М.С. Лебедев, ORCID: 0000-0002-0207-7672 <lebedev@ispras.ru>
[1] С.А. Смолов, ORCID: 0000-0003-0173-3081 <smolov@ispras.ru>
[1] Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, г. Москва, Ленинские горы, д. 1
[3]Московский физико-технический институт,
141701, Россия, Московская обл., г. Долгопрудный, Институтский пер., д.9
[4]Национальный исследовательский университет «Высшая школа экономики»,
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. При проектировании модулей цифровой аппаратуры могут возникать конфликты доступа к данным. Одним из способов их выявления на ранних стадиях проектирования является статический анализ описаний цифровой аппаратуры (или HDL-описаний). В данной статье описывается метод поиска конфликтов доступа к данным в HDL-описаниях. Метод реализован в инструменте Retrascope и ориентирован на конфликты следующих типов: одновременные чтение и запись; одновременная запись; обращение к неинициализированным данным; отсутствие чтения между двумя актами записи. Конфликты задаются в виде условий (assertion) на внутренние переменные. Входное HDL-описание автоматически транслируется в формальную модель на языке, являющемся входным для инструмента проверки моделей nuXmv. Трансляция включает следующие этапы: 1) предварительная обработка; 2) построение графа потока управления; 3) трансформация графа потока управления в решающую диаграмму охраняемых действий (GADD-модель); 4) трансляция GADD-модели в формат инструмента nuXmv. Условия возникновения конфликтов строятся автоматически на основе статического анализа GADD-модели и передаются инструменту проверки моделей nuXmv. Найденные контрпримеры (последовательности значений входных сигналов, приводящие к достижению конфликта) автоматически транслируются инструментом Retrascope в тесты, которые могут быть исполнены на симуляторе. Предложенный метод поиска конфликтов был применен к ряду открытых тестовых наборов и модулей – Texas-97, Verilog2SMV, VCEGAR, mips16. Были выявлены потенциальные конфликты для всех указанных категорий. В качестве направлений дальнейших исследований рассматриваются вынос условий конфликтов на уровень входных сигналов (и получение, таким образом, сведений о протоколах взаимодействия между модулями), а также генерация встроенных проверок в коде HDL-описаний.

Ключевые слова: разработка аппаратуры; язык описания аппаратуры; функциональная верификация; статический анализ; генерация тестов; конфликт доступа к данным; граф потока управления; охраняемое действие; решающая диаграмма охраняемых действий; проверка модели.

Для цитирования: Камкин А.С., Лебедев М.С., Смолов С.А. Поиск конфликтов доступа к данным в HDL-описаниях. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 135-144 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-11

1. Introduction

Modern hardware designs contain multiple modules and processes operating on the common set of internal variables. In this case *conflicts*, i.e. illegal accesses from different processes to the same data, may appear. Requirements on how to operate with modules and avoid conflicts in a communication protocol can be described both in *formal* (machine-readable) and *informal* (human-readable) ways.

In this paper, a formal verification based approach to conflict extraction is proposed. The idea is to analyze an HDL description aimed at finding *data access conflicts* [1]. Both the conflicts and the target description are then automatically translated into the input format of a *model checking* tool. The tool generates counterexamples for the feasible conflicts.

2. Related work

In [1] several categories of data conflicts are described: *read after write* (RAW), *write after read* (WAR) and *write after write* (WAW). The HOL verification system [2] was used to check a RISC processor's pipeline. The formal specification of pipeline was implemented manually that is hard to be done for modern processors because of their complexity.

In [3], a GoldMine methodology is presented for automatic generation of hardware assertions. The method uses a combination of data mining and static analysis techniques. First, the HDL design is simulated to generate data about the design's dynamic behavior. Then, the generated data are mined for "candidate assertions" that are likely to be *invariants*. The data mining technique used is a decision-tree-based supervised learning algorithm. The candidate assertions are then passed through the Cadence Incisive Formal Verifier [4] tool to filter out the spurious candidates. The disadvantages of GoldMine are: 1) usage of commercial tool; 2) invariants' incompleteness because of random simulation usage at an early stage.

3. Assertion extraction method

We propose a new approach to data access conflicts extraction in HDL descriptions. Our goal is to detect conflicts and provide proofs that they may happen. The method is aimed at conflicts of the following types:

- *read-write* (RW): on the same clock tick one process writes the variable and the other process reads it;
- *write-write* (WW): on the same clock tick at least two processes write the same variable;
- *write-read-write* (WRW): we assume that a variable should be read between two writes;
- *undefined* (UNDEF): variable is read before it was written.

```
if (state_bank3[inst_addr] > 1) // C1
    prediction[3] <= 1;     // B1
else
    prediction[3] = 0;      // B2
if (state_bank2[inst_addr] > 1) // C2
    prediction[2] <= 1;     // B3
else
    prediction[2] <= 0;     // B4
if (state_bank1[inst_addr] > 1) // C3
    prediction[1] <= 1;     // B5
else
    prediction[1] <= 0;     // B6
```

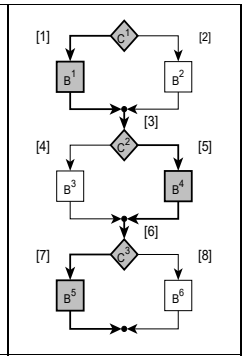

Fig. 1. Control Flow Graph Example

The method consists of the following steps: 1) *Control Flow Graph* (CFG) extraction; 2) transformation to *Guarded Actions Decision Diagram* (GADD); 3) process invariants and conflict assertions extraction; 4) invariants and assertions translation into an input format of a model checking tool; 5) counterexample generation. All method steps are made automatically. The CFG representation is built for every process of the HDL model using an abstract syntax tree traversal compiler-like approach [5]. From the structural view, CFG is a directed graph. Nodes of the graph contain HDL operators; edges of the graph mean control flows. On the left side of fig. 1 the fragment of Verilog code is shown; the related CFG is shown on the right side. Branch operators are shown as diamond nodes and called as C^i. Basic block operators are shown as rectangles and called as B^i. Graph edges contain the values that the branch conditions should be equal to for edges to be passed.

CFG is supposed to be acyclic: HDL loops with constant numbers of iteration are unrolled into sequences of operators.

The next step is the transformation of the CFG to a GADD that is a labeled DAG of guarded actions. A pair $\{\gamma, \delta\}$, where γ is a guard and δ is an action, is called a *guarded action* (GA) [6]. The main idea of the CFG-GADD transformation method is in extraction of branch-free sub-paths from the CFG. Every such sub-path (GA) contains a condition (*guard*) and a sequence of assignment operators (*action*). For action to be executed the guard should be satisfied. Actions are represented in the *static single assignment* (*SSA*) form [7]. To connect subsequent GAs into a complete CFG path an auxiliary *phase* variable is used.

To illustrate this step of the approach, let us take the previous example (see Fig. 1). The CFG model contains the following execution path: $C^1 \to B^1 \to C^2 \to B^4 \to C^3 \to B^5$. Path nodes are grey-colored in the fig. 1; path edges are highlighted too. The following GAs can be extracted from the path: $\{C^1, B^1\}$, $\{C^2, B^4\}$, $\{C^3, B^5\}$. Every GA corresponds to a unique value of the phase variable. The phase variable changes its value upon moving from one GA to another. On Fig. 1 related values of the phase variable are shown in brackets (the initial phase value is 0). Fig. 2 shows the example of GADD model from the previous example:

The main advantage of GADD model is path number reduction in comparison to CFG. In worst case (when CFG is a sequence of branch operators) the GADD has $O(n)$ paths, where n is the number of branches, but the CFG has $O(2^n)$.

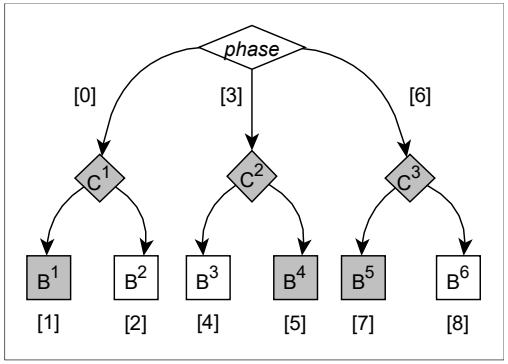

Fig. 2. GADD example

Then the GADD is transformed into the *invariants* of the processes, which represent the cycle-accurate behavior of the processes. The invariant is a logical formula and is a kind of a SSA representation of the whole process. Every GA of the GADD contains a unique phase variable value assignment. These unique values can be used as SSA version values of the variables. The phase variable is removed from the resulting formula because it does not affect the process behavior.

Each variable that is *defined* in a GA is labeled by the corresponding phase value. Each variable that is *used* in the GA is labeled by the set of phase values of the preceding GAs. For guards intermediate variables are introduced. To determine the values of the used variables, a backward search of the GADD is used: it is obvious that the variable value was defined in one of the preceding GAs or did not change from the previous cycle. After that, the process invariant formula is built as a conjunction of equality expressions representing each GA's guards and actions.

Let us see how a process invariant is built using a small example. Fig. 3 shows a part of the GADD and represents three guarded actions.

The guard conditions are: $z == a$, b and c respectively, and the actions contain definitions of variables x, $y$and unique definitions of $phase$. A set of the preceding phase values is $\{i, j\}$; z is a

variable; a, b and $c$are constants; f, g and h are functions defining the values of x and y; $V$is a set of process variables.

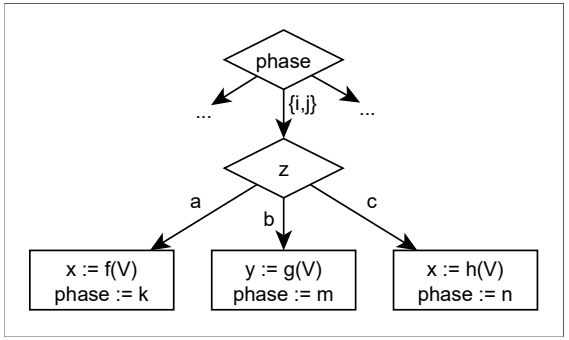

Fig. 3. Original part of the GADD

On the first step we label the variables by the corresponding phase values. The result of that is shown on fig. 4.

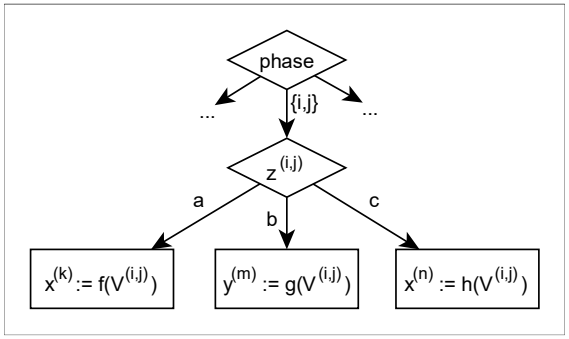

Fig. 4. GADD part with labeled variables

The used variables are now labeled by the preceding phase values $\{i, j\}$ and the defined variables are labeled by the corresponding phase values k, m, n. Phase definitions are removed.

Then we introduce and define a variable for each guard. The guard variable definition consists of a guard expression and a link to the preceding guards. This helps us restore the path from the beginning of the process to the corresponding guarded action. For example:

$$guard^{(k)} := (z^{(i,j)} == a)\ \&\ (guard^{(i)} \mid guard^{(j)})$$

When all the variables in all the GAs are labeled by phases, the remaining unknown used variables' values can be determined. Let us determine the value of $z^{(i,j)}$. So we traverse the GADD backward using the preceding phase values, starting from i and j (fig. 5). When a definition is found on some path (denoted def on fig. 5), the traversal of this path completes and the definition value is collected. If the beginning of the process is reached, the variable preserves its value from the previous cycle.

In the example on fig. 5 the variable z is defined on phases s and t or may not change its value. So the value of $z^{(i,j)}$ can be determined as follows:

$$z^{(i,j)} := guard^{(s)}\ ?\ z^{(s)} : guard^{(t)}\ ?\ z^{(t)} : z$$

On the final step the invariant formula is built. As it was mentioned, it is a conjunction of equality expressions for every labeled variable of the process including the guard variables:

$$x^{(k)} == f(V^{(i,j)})\ \&y^{(m)} == g(V^{(i,j)})\ \&x^{(n)} == h(V^{(i,j)})$$
$$\&guard^{(k)} == ((z^{(i,j)} == a)\ \&\ (guard^{(i)} \mid guard^{(j)}))$$
$$\&\ guard^{(m)} == ((z^{(i,j)} == b)\ \&\ (guard^{(i)} \mid guard^{(j)}))$$
$$\&guard^{(n)} == ((z^{(i,j)} == c)\ \&\ (guard^{(i)} \mid guard^{(j)}))$$
$$\&z^{(i,j)} == (guard^{(s)}\ ?\ z^{(s)} : guard^{(t)}\ ?\ z^{(t)} : z)\ \&\ \ldots$$

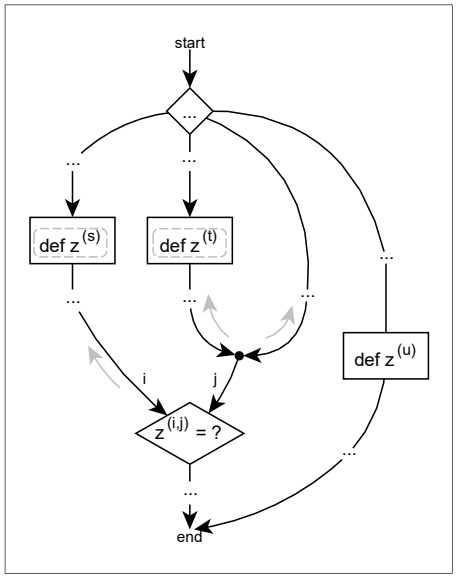

Fig. 5. Version value search (CFG view)

After the process invariant is built, the definition and usage conditions can be collected. They are collected only for internal and output variables of the HDL model, because input variables can be only used.

If a variable is *defined* (*used*) in the action of a GA, its definition (usage) condition equals the guard variable that corresponds to this GA. If a variable is used in the guard condition of a GA, its usage condition equals the disjunction of the corresponding guard variables of the preceding GAs. The variable definition (usage) condition of the whole process is the disjunction of the variable definition (usage) conditions of the GAs.

In our example, the definition conditions for variables $x$and y are:

$$def(x) = guard^{(k)} \mid guard^{(n)}$$
$$def(y) = guard^{(m)}$$

The usage condition for variable z is:

$$use(z) = guard^{(i)} \mid guard^{(j)}$$

Then the conditions are transformed into the assertions of conflict types described above. The assertions are represented as the *Linear-time Temporal Logic* (LTL) [8] formulas and state that the abovementioned conflicts never happen.

If, for example, a variable v is defined and used both in processes $p1$ and $p2$, the corresponding RW conditions are:

$$!\ \boldsymbol{F}\ (def_{p1}(v)\ \&use_{p2}(v))$$
$$!\ \boldsymbol{F}\ (def_{p2}(v)\ \&use_{p1}(v))$$

The corresponding WW condition:

$$! \, \boldsymbol{F} \, (def_{p1}(v) \, \& def_{p2}(v))$$

The corresponding WRW condition:

$$\boldsymbol{F} \, ((def_{p1}(v) \mid def_{p2}(v)) \\ \& \, (\boldsymbol{F} \, (use_{p2}(v) \mid use_{p1}(v)) \, \boldsymbol{U} \, (def_{p1}(v) \mid def_{p2}(v))))$$

The corresponding UNDEF condition:

$$\boldsymbol{G} \, (! \, (use_{p2}(v) \mid use_{p1}(v)) \, \boldsymbol{U} \, (def_{p1}(v) \mid def_{p2}(v)))$$

Invariants and assertions are then translated into the SMV model. Their translation is rather straightforward. It is only important to define the variable value in the next state of the model using the keyword *next*. This value equals the last version of the variable before the end of the process. For example, if the final phase values of a process are k, l, m, then the next state value of a variable *v* is defined as:

$$next(v) := v^{(k,l,m)}$$

The SMV model is checked by the nuXmv[9] tool using bounded model checking. Ifan assertion is violated, a counterexample is generated and a potential conflict is found. The counterexamples may be later translated into test scenarios for the original HDL description.

4. Case study

The method was implemented in the Retrascope [10] tool. It was applied to arange of Verilog designs from the Texas-97 [11], VCEGAR [12] and Verilog2SMV VIS [13] benchmarks and the 16-bit MIPS processor [14].Table 1 contains the results of the method's application: benchmark descriptions and generated assertions amount. Here *N* means total amounts of top-level modules.Most of the assertions denote only suspicious situations, so the results should be analyzed by a verification engineer to filter out the real data conflicts.

Table 1. Benchmark descriptions and potential conflicts.

Bench	N	LOC	Assertions			
			RW	WW	WRW	UNDEF
Texas'97	17/58	69539	408	26	211	211
VCEGAR	20/34	15144	315	25	167	167
Verilog2SMV	12/20	4494	78	0	62	62
mips16	5/12	1007	10	0	9	9

Example of a RW situation, which is not a conflict (*mips16/ID_stage.v*):

```
module ID_stage
…
wire  [2:0] ir_dest_with_bubble;
wire  [2:0] write_back_dest;

assign ir_dest_with_bubble = ( instruction_decode_en ) ?
     ir_dest : 0;
assign write_back_dest = ir_dest_with_bubble;
```

Signal *ir_dest_with_bubble* is defined in one process and is used in the other process at the same time.

Example of a WW situation, which seems to be a real conflict (*Texas97/MPEG/prefixcode.v*):


```
module start_code_prefix(start, done…);
…
reg monitor;
…
always @(posedge read_signal) begin
 monitor=start;
…
end
always if( start==0) begin
…
 monitor=0;
end
```

Variable *monitor* is defined simultaneously, if *read_signal* rises and at the same time *start* equals 0.

Example of an UNDEF situation, which is also not a conflict (*mips16/ID_stage.v*):

```
module ID_stage

…
reg   [15:0]  instruction_reg;
…
always@(posedge clk or posedge rst) begin
 if (rst) begin
    instruction_reg<= 0;
 end
 else begin
    if (instruction_decode_en) begin
       instruction_reg <= instruction;
    end
 end
end
assign ir_op_code = instruction_reg[15:12];
```

Register *instruction_reg* is undefined from the start of simulation until the *clk* or *rst* rising edge.

5. Conclusion and future work

In this paper, the approach to data access conflicts extraction from HDL descriptions has been proposed. We extract assertions from the source code and automatically translate them into the input format of the model checker. The tool generates counterexamples that are proofs of conflicts' reachability. We have implemented the approach in the Retrascope toolkit and applied it to several open source HDL benchmarks.

One direction for future research is to propagate assertions from internal variables' to interface variables. Such assertions can be used to improve protocols of unknown third-party modules or even to reconstruct protocols. Another direction is the generation of *checkers*, i.e. HDL wrappers for target modules that check their behavior through simulation.

References

[1]. S.Tahar, R. Kumar. Formal Verification of Pipeline Conflicts in RISC Processors. In Proc. of the European Design Automation Conference (EURO-DAC), 1994, pp. 285-289.

[2]. M. Gordon,T. Melham. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, 1993, 492 p.

[3]. S. Hertz, D. Sheridan, S. Vasudevan. Mining Hardware Assertions with Guidance From Static Analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 6, 2013, pp. 952-965.

[4]. Cadence Incisive Formal Verifier. https://community.cadence.com//CSSharedFiles/forums/storage/22/10078/IncisiveFV_ds.pdf

[5]. A.V. Aho, R. Sethi, J.D. Ullman. Compilers: principles, techniques, and tools. Addison-Wesley, 1986, 796 p.
[6]. J. Brandt, M. Gemunde, K. Schneider, S. Shukla, J.-P. Talpin. Integrating system descriptions by clocked guarded actions. In Proc. of Forum on Specification and Design Languages (FDL), 2011, pp. 1-8.
[7]. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems, vol. 13, issue 4, 1991, pp. 451-490.
[8]. A. Pnueli. The temporal logic of programs. In Proc. of the 18th Annual Symposium on Foundations of Computer Science, 1977, pp. 46-57.
[9]. nuXmv model checker. https://nuxmv.fbk.eu
[10]. Retrascope toolkit. https://forge.ispras.ru/projects/retrascope
[11]. Texas-97 Verification Benchmarks. https://ptolemy.berkeley.edu/projects/embedded/research/vis/texas-97
[12]. VCEGAR benchmark collection. https://www.cprover.org/hardware
[13]. Verilog2SMV tool. https://es-static.fbk.eu/tools/verilog2smv
[14]. Educational 16-bit MIPS Processor. https://opencores.org/projects/mips_16

Information about authors / Информация об авторах

Александр Сергеевич КАМКИН – ведущий научный сотрудник отдела технологий программирования Института системного программирования им. В.П. Иванникова Российской академии наук (ИСП РАН). Также он читает лекции в Московском государственном университете им. М.В. Ломоносова (МГУ), Московском физико-техническом институте (МФТИ) и Высшей школе экономики (НИУ ВШЭ). Кандидат физико-математических наук (2009). Область научных интересов: архитектура микропроцессоров, проектирование цифровой аппаратуры, верификация и тестирование цифровой аппаратуры, генерация тестовых программ для микропроцессоров, верификация подсистем управления памятью микропроцессоров, статический и динамический анализ HDL-описаний.

Alexander Sergeevich KAMKIN is a leading researcher at the Software Engineering Department of Ivannikov Institute for System programming of the Russian Academy of Sciences (ISP RAS). He is also a lecturer at Moscow State University (MSU), Moscow Institute of Physics and Technology (MIPT) and Higher School of Economics (HSE). His research interests include hardware design, functional verification, test program generation, and formal methods. He has a MS in computer science (MSU, 2003) and a PhD in computer science (ISP RAS, 2009). He is an expert of RAS and one of the organizers of Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE).

Михаил Сергеевич ЛЕБЕДЕВ – инженер 2й категории отдела технологий программирования ИСП РАН. Область научных интересов: архитектура микропроцессоров, проектирование цифровой аппаратуры, верификация и тестирование цифровой аппаратуры, статический и динамический анализ HDL-описаний.

Mikhail Sergeevich LEBEDEV is an engineer at the Software Engineering Department of ISP RAS. He has a MS in hardware engineering (MEPhI, 2011). His research interests include hardware design, functional verification, formal methods and static analysis.

Sergey Aleksandrovich SMOLOV is a junior researcher at the Software Engineering Department of ISP RAS. He has a MS in computer science (MIPT, 2010). His research interests include functional verification, formal methods and static analysis.

Сергей Александрович СМОЛОВ – младший научный сотрудник отдела технологий программирования ИСП РАН. Область научных интересов: функциональная верификация, формальные методы, статический анализ.