An Approach to Test Programs Generation for Microprocessors Based on Pipeline Hazards Templates

Alexander Kamkin, Dmitry Vorobyev
Institute for System Programming of the Russian Academy of Sciences
25, A.Solzhenitsyn Street, Moscow, 109004, Russia
E-Mail: {kamkin, vorobyev}@ispras.ru

Abstract — **In this paper we describe an approach to automated test programs generation intended for microprocessor verification. The approach is based on formal specification of microprocessor ISA and description of pipeline hazards templates. The use of formal specifications allows automating development of test program generators and systematizing control logic verification. Since the approach is underlain by high-level descriptions, all specifications and templates developed, as well as the constructed test programs, can be easily reused when the processor's microarchitecture changes. It makes it possible to apply the methodology in early stages of a microprocessor development cycle when the design is frequently modified.**

I. Introduction

Functioning of a modern pipelined microprocessor is implemented in a very difficult way. Pipeline can concurrently process multiple instructions, which, in addition, can interact each other via shared resources. At every cycle of execution a microprocessor makes lots of decisions on hazards resolution, branches processing, exceptions handling, and so on. The microprocessor mechanisms responsible for controlling instructions execution are called *control logic*.

Control logic is a key component of a microprocessor; that is why it should be designed and verified thoroughly, not missing any detail. However, the common practice is to produce tests manually or using random generation techniques, which is obviously inefficient and unsystematic. The other kinds of methods are based on *cycle-accurate models*. Such approaches are aimed at detailed verification of control logic, but the problem is that accurate models are very hard to develop and maintain.

The approach suggested in this work is thought to be somewhere between random generation techniques and techniques based on cycle-accurate models. The approach uses formal specification (modeling) of a microprocessor instruction set (ISA, Instruction Set Architecture). Of course, instruction-level models are less informative in comparison with cycle-accurate ones, but they have a number of practical advantages. First, they are much easier to develop, and second, they can be reused even if the microarchitecture is considerably altered.

This work was supported by the RFBR (grant 08-01-00889-a).

The rest of the paper is organized as follows. Section II is a survey of the related work. Section III introduces the main concepts of the suggested approach. The description of the approach is given in Section IV. Section V considers a case study. Finally, Section VI concludes the paper.

II. Related Work

In the paper [1] two mutually complementary techniques are described. The first one is a test generation technique basing on *model checking*, while the second one uses *template-based procedures*. Source information for the both approaches is a microprocessor specification written in EXPRESSION [2]. Basing on the specification, a generic structure-behavior model in SMV [3] is automatically derived. For this model, a test developer defines a *fault model*. For each element of the fault model the negation of the corresponding property is produced. Then, a counter-example is generated using the SMV model checker. As the authors say, model checking does not scale on complex designs. So, the technique employing templates is used as an addition.

Templates are developed by hands and describe sequences of instructions that create special situations in microprocessor behavior (in the first place, pipeline hazards). Generation is performed with the help of the graph model extracted from the specification. The template-based technique requires greater efforts, but it scales well. It should be noticed that both approaches are based on rather accurate specifications, and it is better to apply them in late design stages, when the microarchitecture is stable. Otherwise, there would be a need to modify the specification to keep up its consistency.

In the approach [4], pipeline structure is formally specified in the form of state machine, which is called OSM (Operation State Machine). OSM describes control logic at two levels, called *operational* and *hardware* levels. At the first level, "movement" of instructions through the pipeline stages is described (every operation is described by a separate state machine). At the second level, hardware resources are modeled using so-called *token managers*. An operation state machine changes states by capturing and moving tokens. A pipeline model is defined as a composition of operation state machines and resource state machines. The goal of testing is to traverse all the transitions of the joint

automaton. Like the previous techniques, this approach is based on accurate specifications.

The paper [5] considers the test program generation tool Genesys-Pro (IBM Research). A generator is composed of two main components, an *engine* (which does not depend on target architecture) and a *model* (which describes microprocessor-specific knowledge). A verification engineer develops *templates*, which specify structure of test programs and properties they should satisfy. Genesys-Pro transforms each template into a set of constraints and builds a test program using *constraint solving* techniques. The tool is quite universal and applicable to different microprocessor architectures. However, so far as development of test templates is done by hand, tests maintenance is hard enough.

In the paper [6] a technique for test generation basing on FSM traversal is described. In one of the stages the technique uses Genesys (the previous version of Genesys-Pro). A test developer creates a microprocessor model using the SMV language. After this, a set of paths covering all the edges of the state graph extracted from the model is built. Each path (so-called *abstract test*) is translated into a Genesys template. The technique allows achieving good coverage of control logic, but it has two principal shortcomings. First, one needs a skilled expert to develop a microprocessor model. Second, to be able to map abstract tests into Genesys templates, a complex description has to be done.

Summing up, all the techniques reviewed can be divided into two classes: *techniques based on accurate models* [1,4,6] and *techniques based on templates* [5]. The techniques of the first class allow achieving high quality of verification. However, it is not practical to use them in early design stages. Template-based techniques do not have this shortcoming, but they are unsystematic and cannot guarantee high quality of verification.

III. Foundations of the Suggested Approach

The suggested approach is based on *combinatorial model-based generation* [7]. It uses *formal specifications* of microprocessor ISA, which describe instructions regardless of their processing on a pipeline. Description of each instruction includes a mnemonic, list of operands, precondition, latency, and semantics in the imperative form. Besides instructions, *test situations* and *dependencies between instructions* are formally described. Test programs are generated automatically by combining test situations and dependencies for finite sequences of instructions.

A. Structure of a Test Program

Test program is a sequence of *test cases*. The key part of a test case is a *test action*, which is a specially prepared sequence of instructions intended to create a certain situation in microprocessor behavior (hazard, exception, and so on). A test action is prepared by *initializing instructions* and followed by a *test oracle* (sequence of instructions that checks correctness of a microprocessor state after execution of the test action). Thus structure of a test program can be described by the expression:

$$Test = \{\langle Pre_i, Action_i, Post_i \rangle\}_{i=0,n-1},$$

where Pre_i is the initializing instructions of the ith test case, $Action_i$ is the test action, and $Post_i$ is the test oracle. In elementary case a test program consists of a singular test case without a test oracle ($Test = \langle Pre, Action \rangle$).

The assembler code below is a test program fragment that contains one test case (we use MIPS ISA [8] to illustrate ideas of the approach).

```
// Initialization of sub[0]: IntegerOverflow=true
// s5[rs]=0xffffffffc1c998db, v0[rt]=0x7def4297
lui s5, 0xc1c9
ori s5, s5, 0x98db
lui v0, 0x7def
ori v0, v0, 0x4297

// Initialization of add[1]: Exception=false
// a0[rs]=0x1d922e27, a1[rt]=0x32bd66d5
...
// Initialization of div[2]: DivisionByZero=true
// a2[rs]=0x48f, a1[rt]=0x0
...

// Dependencies: div[2].rt[1]-sub[0].rd[0]

// Test action: 2010
sub a1, s5, v0 // IntegerOverflow=true
add t7, a0, s3 // Exception=false
div a2, a1     // DivisionByZero=true
```

In this fragment, *Action* is represented as a sequence of three instructions: `sub`, `add` and `div`. There is a register dependency between the `rt` operand of the `div` instruction and the `rd` operand of the `sub` instruction. This dependency implies using the same register for each of the operands. Initializing instructions *Pre* load values into the independent input registers of all instructions of the test action (see preparation of the instruction `sub`, for example). *Post* is empty here.

B. Test Templates

Test template is an abstract representation of a test action where constraints (test situations and dependencies) are specified instead of concrete instructions and their operands' values. Generally speaking, each template defines a testing purpose (situation that should be tested). The goal of test program generation is to construct a representative set of test templates. One of the possible templates for the example above is shown below.

```
IADDInstruction R, ?, ? @ IntegerOverflow=true
IADDInstruction ?, ?, ? @ Exception=false
IDIVInstruction ?, R    @ DivisionByZero=true
```

The template is composed of three instructions. The first two instructions belong to the equivalence class `IADDInstruction`, while the third one belongs to `IDIVInstruction`. For the first instruction the situation `IntegerOverflow=true` is given (instruction should throw the integer overflow exception). The situation of the second instruction is `Exception=false` (absence of exceptions). The third instruction should raise division by zero (`DivisionByZero=true`). In addition, there is a dependency between the first and the third instructions (the first register of the first instruction is equal to the second register of the third instruction). Independent operands of the instructions, i.e., operands that are not revolved by the dependency are denoted as `?`.

Test templates are allowed to be *parameterized*. The example below demonstrates a template with four parameters: `$FirstInstruction` (equivalence class of the first instruction), `$Situation` (test situation of the first instruction), `$ThirdInstruction` (equivalence class of the third instruction), and `$Dependency` (dependency of the third instruction on the first and the second instructions).

```
$FirstInstruction @ $Situation
IADDInstruction   @ IntegerOverflow=false
$ThirdInstruction @ $Dependency
```

C. Test Situations

Speaking about control logic verification, test situations related to execution of instructions on a pipeline are of interest. As a rule, processing of an instruction is performed in the same way for all values of the operands (of course, if exceptions are not taken into account). Thereby, for an instruction that can raise N exceptions, N+1 test situations are usually defined: *Exception=false*, $Exception_0=true$, …, $Exception_{N-1}=true$. For an instruction which handling depends on the operands values one should specify all possible paths of its execution.

Branch instructions are examined in a particular way. A test situation of a branch instruction includes a target address and truth values of the condition (if the branch instruction is conditional). In general case, a test situation is as follows: *Target=Label*, $Trace=\{C_0, \ldots, C_{M-1}\}$, where *Label* is a target label (address) and C_i is a truth value of the branch condition for the ith time of the instruction execution.

D. Dependencies between Instructions

Dependencies between instructions are thought to have a key role in creation of pipeline hazards. There are two main types of dependencies: *register dependencies* and *address dependencies* (*data dependencies*). Register dependencies are expressed as equality of registers being used as operands of two instructions. Such dependencies can be of the following types:

- *read-read* — both instructions read from the same register;
- *read-write* — the first instruction reads from a register, while the second one writes into it;
- *write-read* — the first instruction writes into a register, while the second one reads from it;
- *write-write* — both instructions write into the same register.

Address dependencies have more complex structure and are related to internal organization of memory management units [9]. Some examples of the address dependencies are itemized below.

- *VAEqual* — equality of virtual addresses;
- *TLBEqual* — equality of TLB entries;
- *PAEqual* — equality of physical addresses;
- *CacheRowEqual* — equality of cache rows.

IV. The Suggested Approach

Basing on documentation analysis, a verification engineer marks out situations “interesting” from the control logic point of view (different types of pipeline hazards). For each hazard type its *generalized specification* is developed, which is a parameterized template creating the corresponding hazard situation. Such templates are also called *pipeline hazards templates* or *basic templates*. Basic templates are usually of a small size, because hazards between instructions occur if instructions are close to each other. To be able to generate tests, one should define *iterators* of templates' parameters. After this, a generator constructs test programs using different values of the parameters and combining templates together.

A. Specification of Hazards

Let us examine a general scheme of pipeline hazards specification. All of the situations derived from the documentation are classified by their types (see Fig. 1). The four main types are *exceptions*, *data hazards*, *structural hazards*, and *control hazards*.

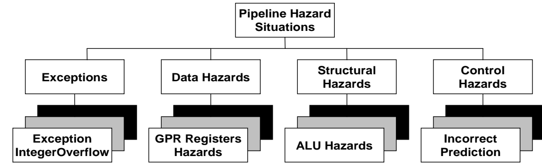

Figure 1. Classification of pipeline hazard situations

Generally, all situations of the same type are described by one basic template. The difference between two single-type specifications is connected with various constraints for template parameters – parameters domain is divided by a verification engineer into a number of equivalence classes, and this serves as a basis for the further construction of iterators (see Fig. 2).

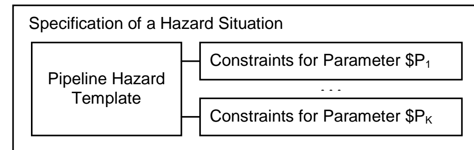

Figure 2. Specification of a hazard situation

1) Specification of Exceptions

Exception is a special event that signals that something goes wrong during instruction execution. When an exception is raised, control flow is switched to a special routine, called *exception handler*, and all the instructions loaded after the instruction throwing the exception are flushed. The typical errors related to exception handling are incorrect setting of an exception signal (incorrect calculation of an exception condition) and incorrect flushing of loaded instructions.

There are two main strategies for exception handling in test programs. First, if an exception is raised, execution is switched to the next instruction of the test action. Second, execution is switched to the test oracle passing the rest

instructions of the test action. To check pipeline flushing mechanisms, the second strategy is preferable. Generalized specification of an exception is given by the template below.

```
$PreInstructions
$ExceptionInstruction @ $ExceptionType
$PostInstructions
```

The template uses the following parameters:

- `$PreInstructions` — a sequence of instructions that precedes an exception (pre-instructions should not raise exceptions);
- `$ExceptionInstruction` — an instruction that raises an exception;
- `$ExceptionType` — an exception type;
- `$PostInstructions` — a sequence of instructions that succeeds an exception (post-instructions should be flushed).

Here is an example of a concrete test action that corresponds to the template given.

```
dadd   r25, r30, r7
lb     r22, 0(r4) // TLBInvalid=true
daddiu r5,  r18, 13457
```

The sequence `$PreInstructions` consists of the only instruction `dadd`. `$ExceptionInstruction` is instantiated by the instruction `lb` that raises the exception `TLBInvalid` (`$ExceptionType`). The sequence `$PostInstructions` includes the only instruction `addiu`.

2) *Specification of Data Hazards*

Data hazards are situations in which different instructions try to access the same data and at least one instruction tries to write them. Thereby, to describe data hazards, one should use "read-write", "write-read", and "write-write" types of dependencies. The typical error related to data hazard resolution is incorrect implementation of pipeline interlocks resulting in data flow integrity violation. Generalized specification of a data hazard is given by the template below.

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```

The template uses the following parameters:

- `$PreInstructions` — a sequence of instructions that precedes a dependency (pre-instructions should not raise exceptions);
- `$FirstInstruction` and `$SecondInstruction` — a pair of dependent instructions that causes a data hazard;
- `$Dependency` — a dependency between instructions that causes a data hazard;
- `$InnerInstructions` — a sequence of instructions between dependent instructions (inner-instructions should not raise exceptions and produce hazards);
- `$PostInstructions` — a sequence of instructions that succeeds a data hazard (post-instructions are usually suspended with the dependent instruction).

Here is an example of a concrete test action that corresponds to the template given.

```
madd.s   $f18, $f6,  $f28, $f10
add.s    $f8,  $f17, $f3
ceil.l.s $f2,  $f18  // Data hazard
div.s    $f23, $f13, $f24
```

3) *Specification of Structural Hazards*

Structural hazards occur when several instructions try to access the same unit of a microprocessor (or some other resource). Usually, such kinds of hazards happen when two similar multi-cycle instructions are located closely to each other. In some cases an additional data dependency is required to create a structural hazard between instructions. The typical error related to structural hazard resolution is the same as for data hazards (incorrect implementation of pipeline interlocks). Generalized specification of a structural hazard is absolutely the same. A concrete example is given below.

```
div.s $f11, $f27, $f3
add.s $f28, $f7,  $f30
div.d $f23, $f1,  $f20 // Structural hazard
add.d $f18, $f2,  $f25
```

4) *Specification of Control Hazards*

Control hazards are related to branch instructions. Depending on microprocessor organization, execution of a branch instruction can result in pipeline *stalling* or *flushing*. Errors of control hazard resolution relate to pipeline interlocks, branch prediction and other control logic mechanisms. Generalized specification of a control is given by the template below.

```
$PreInstructions
$BranchInstruction @ $Target, $Trace
$DelaySlots
$PostInstructions
```

The template uses the following parameters:

- `$PreInstructions` — a sequence of instructions that precedes a branch instruction;
- `$BranchInstruction` — a branch instruction;
- `$Target` — a target address of a branch instruction;
- `$Trace` — an execution trace of a branch execution (sequence of truth values of a branch condition);
- `$DelaySlots` — instructions in delay slots;
- `$PostInstructions` — a sequence of instructions that succeeds a branch instruction.

Here is an example of a concrete test action that corresponds to the template given.

```
L:addi r1, r1,  1
  beq  r1, r0,  L // Target=L, Trace={1, 0}
  dadd r7, r12, r23
```

B. Test Programs Generation

Let us consider how test programs are generated on the base of test templates. Test actions are divided into two types: *simple test actions* (which correspond to a single basic template) and *composite test actions* (which are constructed by composition of several basic templates).

1) *Constructing Simple Test Actions*

Simple test actions are targeted at creation of one hazard situation. A technique for their construction is easy and based on using *basic templates* and *iterators*. For each situation derived from documentation, a set of test actions is built. Test actions are constructed by iterating parameters

values and combining them to each other (commonly, all possible combinations are used) (see Fig. 3).

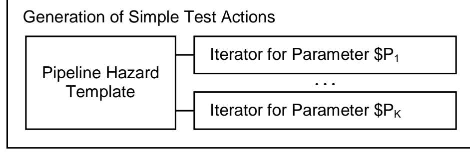

Figure 3. Generation of simple test actions for a pipeline hazard

Let us examine simple test action construction for a structural hazard on FPU (Floating Point Unit) being described by the following basic template:

```
$PreInstructions
$FirstInstruction
$InnerInstructions
$SecondInstruction @ $Dependency
$PostInstructions
```

Some constraints on parameters values are defined for this hazard. For example, the hazard occurs only between instructions being executed more than one cycle. It is also obvious that size of `$InnerInstructions` should not exceed latency of `$FirstInstruction` subtracted by two. In addition, equivalence classes of dependent instructions can be given. Assume that the template's parameters are setup with the following values.

```
FMULInstruction     : {mul.s, mul.d}
FDIVInstruction     : {div.s, div.d}
IADDInstruction     : {add, sub}

$PreInstructions    : {}
$FirstInstruction   : FMULInstruction, FDIVInstruction
$SecondInstruction  : FMULInstruction, FDIVInstruction
$Dependency         : class($FirstInstruction) ==
                      class($SecondInstruction)
$InnerInstruction   : {IADDInstruction}
$PostInstructions   : {}
```

Given the parameters values, two test actions are generated:

```
$PreInstructions    →
$FirstInstruction   →  mul.d $f12, $f3,  $f21
$InnerInstructions  →  sub   r6,   r15,  r3
$SecondInstruction  →  mul.s $f9,  $f23, $f7
$PostInstructions   →

$PreInstructions    →
$FirstInstruction   →  div.s $f18, $f28, $f4
$InnerInstructions  →  add   r25,  r13,  r27
$SecondInstruction  →  div.s $f5,  $f12, $f10
$PostInstructions   →
```

2) *Constructing Composite Test Actions*

The aim of composite test actions as opposed to simple test actions is creation of several "simultaneous" pipeline hazards. Composite actions allow testing complex situations in microprocessor behavior (*parallel hazards*, *nested hazards*, *parallel exceptions*, and so on). Construction of composite actions is performed by composition of several basic templates.

Let T be a template of an arbitrary type, T_E be a template of an exception, T_H be a template of a data or structural hazard, and, finally, T_C be a control hazard template. The main composition operations are given below.

a) Overlapping: $T=T_{H1}|T_{H2}$

T_H.*PreInstructions* = T_{H1}.*PreInstructions* ≡ T_{H2}.*PreInstructions*
T_H.*FirstInstruction* = T_{H1}.*FirstInstruction* ≡ T_{H2}.*FirstInstruction*
T_H.*SecondInstruction* = T_{H1}.*SecondInstruction* ≡ T_{H2}.*SecondInstruction*
T_H.*Dependency* = T_{H1}.*Dependency* & T_{H2}.*Dependency*
T_H.*InnerInstructions* = T_{H1}.*InnerInstructions* ≡ T_{H2}.*InnerInstructions*
T_H.*PostInstructions* = T_{H1}.*PostInstructions* ≡ T_{H2}.*PostInstructions*

b) Shift: $T_H=T_{H1}\downarrow T_{H2}$

T_H.*PreInstructions* = T_{H1}.*PreInstructions*
T_H.*FirstInstruction* = T_{H1}.*FirstInstruction*
T_H.*SecondInstruction* = T_{H1}.*SecondInstruction*
T_H.*Dependency* = T_{H1}.*Dependency* & T_{H2}.*Dependency*
T_H.*InnerInstructions* = {T_{H1}.*InnerInstructions*, T_{H2}.*FirstInstruction*, T_{H2}.*InnerInstructions*}
T_H.*PostInstructions* = {T_{H1}.*PostInstructions*, T_{H2}.*SecondInstruction*, T_{H2}.*PostInstructions*}
T_{H1}.*PostInstructions* ≡ T_{H2}.*PreInstructions*

c) Concatenation: $T=T_1\rightarrow T_2$

T.PreInstructions = T_1.*PreInstructions*
T.MainParameters = T_1.*MainParameters*[2]
T.PostInstructions = T_2
Type of a template T matches with the type of a template T1

d) Nesting: $T_H=T_{H1}[T]$

T_H.*FirstInstruction* = T_{H1}.*FirstInstruction*
T_H.*SecondInstruction* = T_{H1}.*SecondInstruction*
T_H.*Dependency* = T_{H1}.*Dependency*
T_H.*PreInstructions* = T_{H1}.*PreInstructions*
T_H.*InnerInstructions* = T
T_H.*PostInstructions* = T_{H1}.*PostInstructions*

To clearify the semantics of composite templates, let us consider an example where the overlapping is used to compose a data hazard and a structural hazard:

```
$PreInstructions1,2                  → add.s  $f28, $f7,  $f30
$FirstInstruction1,2                 → div.s  $f11, $f27, $f3
$InnerInstructions1,2                → dsub   r25,  r30,  r7
$SecondInstruction1,2
  @ $Dependency1 & $Dependency2      → div.d  $f23, $f11, $f20
$PostInstruction1,2                  → lb     r22,  0(r4)
```

It is intuitively obvious how to iterate test actions for a given composite test template. Some parameters of different basic test templates are identified. After this, iterators are specified for resultant parameters (commonly, iterators developed for simple templates are used). Generation of composite test templates is perform by enumeration of different syntactical structures consisting of a small number of basic templates connection by composition operations.

V. CASE STUDY

The suggested approach was applied to verification of control logic of two arithmetical coprocessors, floating point coprocessor (CP1) and complex arithmetic coprocessor (CP2). Coprocessors have common control flow with CPU and use three execution channels (functional pipelines):

- channel of floating point arithmetic;

[2] *MainParameters* is set of template parameters excluding *PreInstructions and PostInstructions*.

- channel of RAM operations;
- channel of on-chip memory operations.

The control logic supports solving of different types of hazards. In both coprocessors memory exceptions can occur. In addition, in CP1 arithmetic exceptions can be raised. The CPU implements static branch prediction and speculative execution mechanisms.

The test actions were composed of four instructions of different types. Structure of the test situations and dependencies was very much the same as it is described in the paper, but some particular features of the microarchitecture were taken into account and additional data dependencies were included.

TABLE I. CASE STUDY INFORMATION

CPU revision	Code volume (LOC)	Number of instructions	Affected instructions	Affected code (LOC)
Coprocessor CP1				
8	28500	113	—	—
20	28650	114	94	485 (1.7%)
Coprocessor CP2				
8	4950	15	—	—
20	11550	59	5	45 (0.9%)
29	14350	100	18	165 (1.4%)

Table I shows code volume (including ISA specifications and pipeline hazards templates), number of implemented instructions, number of instructions affected by the revision and volume of the affected code. As it is seen in the table, when the microprocessor is modified, a small part of the generated code has to be changed. It took us less than half an hour to alter the code.

The generated test programs detected a considerable number of errors in both coprocessors which had not been found by randomly generated test programs.

VI. CONCLUSION

Verification of a pipeline microprocessor is a very difficult task that cannot be carried out without using automation techniques. In the paper the technique for automated test programs generation is described. As opposed to the common approaches, like manual development and random generation, the suggested methodology has a high level of automation and allows systematically testing control logic of a microprocessor. At the same time, the methodology differs from the approaches that use accurate models by the possibility of using it in early design stages when microarchitecture is frequently revised.

Using accurate models of control logic is reasonable in late stages of a microprocessor design cycle when control logic is stable. Due to the high informativeness of accurate models, such approaches allow finding errors which are really hard to detect. Moreover, accurate models make it possible to create more compact set of tests. In the future we are planning to extend the approach to support accurate models as well. This would make the generator to be more flexible – it would be applicable to both early and late design stages unifying the verification process.

REFERENCES

[1] P. Mishra, N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors*. ACM Transactions on Design Automation of Electronic Systems, 2008.

[2] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration*. Technical Report 1998-29, University of California, Irvine, 1998.

[3] www.cs.cmu.edu/~modelcheck/smv.html.

[4] T.N. Dang, A. Roychoudhury, T. Mitra, P. Mishra. *Generating Test Programs to Cover Pipeline Interactions*. Design Automation Conference, 2009.

[5] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification*. Design and Test of Computers, 2004.

[6] S. Ur, Y. Yadin. *Micro-Architecture Coverage Directed Generation of Test Programs*. Design Automation Conference, 1999.

[7] A. Kamkin. *Test Program Generation for Microprocessors*. Institute for System Programming of RAS, 2008. (in Russian)

[8] *MIPS64™ Architecture For Programmers. Revision 2.0.* MIPS Technologies Inc., 2003.

[9] D. Vorobyev, A. Kamkin. *Test Program Generation for Memory Management Units of Microprocessors*. Institute for System Programming of RAS, 2009. (in Russian)