ISSN 0361-7688, Programming and Computer Software, 2014, Vol. 40, No. 1, pp. 1–9. © Pleiades Publishing, Ltd., 2014.
Original Russian Text © A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko, 2014, published in Programmirovanie, 2014, Vol. 40, No. 1.

Extensible Environment for Test Program Generation for Microprocessors

A. S. Kamkin, T. I. Sergeeva, S. A. Smolov, A. D. Tatarnikov, and M. M. Chupilko

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia

e-mail: kamkin@ispras.ru, leonsia@ispras.ru, ssedai@ispras.ru, andrewt@ispras.ru, chupilko@ispras.ru

Received August 10, 2013

Abstract—Development of test programs and analysis of the results of their execution is the basic approach to verification of microprocessors at the system level. There is a variety of methods for the automation of test generation, starting with the generation of random code and ending with directed model-based test generation. However, there is no cure-all method. In practice, combinations of various complementary techniques are used. Unfortunately, no solution for the integration of various test generation methods into a unified environment is currently available. To test a microprocessor, verification engineers are forced to use many different test generators, which results in a number of difficulties, such as (1) the necessity to ensure the compatibility of tool configurations (in each tool, a specific description of the target microprocessor is used, which leads to duplication of information); (2) the necessity to develop utilities for integration tools (different tools have different interfaces and use different data formats). This paper describes a concept of extensible environment for test program generation for microprocessors. This environment provides a unified approach for test generation; it supports widespread test generation techniques, and can be extended by new testing tools. The proposed concept was partially implemented in MicroTESK (Microprocessor TEsting and Specification Kit).

DOI: 10.1134/S0361768814010046

1. INTRODUCTION

To ensure the *correctness* and *reliability* of microprocessor operation, a set of activities is used. Such activities are known as *verification* and *testing* [1]. Verification is performed at the stage of design; it is intended for detecting logical errors in the microprocessor design. Testing is performed at the manufacturing stage; it is intended for detecting physical defects in integrated circuits. To accomplish these tasks, *test programs* (or just tests) are used; these are special sequences of microprocessor instructions resulting in the occurrence of various situations in its operation (internal events, interaction between components, and others) [2]. Below, we use the term *testing* both for verification and testing.

By the present time, various methods for automated test generation have been proposed. They can be classified as follows: (1) *random generation* (generation of pseudorandom programs) [3], (2) *combinatorial generation* (systematic enumeration of programs of bounded length) [2], (3) *template-based generation* (generation of programs based on a given high-level description of test scenarios) [4]; (4) *model-based generation* (generation of programs that cover a certain class of situations in the microprocessor model) [5]. It is clear that there is no universal method for all testing problems. In practice, a combination of various approaches, which complement and enhance each other, is used. Typically, the generic operation of the microprocessor is verified using random tests, while the most important modules and subsystems are verified using model-based tests.

Unfortunately, no unified environment for the integration of various test generation methods is currently available. To test a microprocessor, verification engineers have to use many tools with different formats of input and output data; this gives rise to difficulties in their integration and maintaining the compatibility of their configurations. When independent works are performed (for example, for the generation of tests of different kinds), different tools can easily be used. However, a complex task may require the use of several tools (for example, if a test consisting of parts of different types is to be generated); in this case, a nontrivial integration procedure can be needed. Furthermore, different tools use different incompatible descriptions of the microprocessor and test scenarios; as a result the same objects are specified several times thus complicating test generation support.

In this paper, we propose a concept of extensible test generation environment for microprocessors. This environment is built around a *microprocessor model* that provides knowledge about the target microproces-

sor (its instruction set, registers, etc.) in a certain form. This model can be used by *test generators*, which are extensions of the environment. All test generators implement identical interfaces that allow one to specify generation parameters (such as a subset of the instruction set to be used, probabilities of various instructions, and the like) and get access to the generation results (internal representation of tests). Engineers interact with the environment using *test templates* that hierarchically specify the test generators to be used, parameters of their operation, and methods used to merges the test generation results into a unified test program.

The paper is organized as follows. In Section 2, a review of the available test generation methods and tools is given. In Section 3, these approaches are analyzed, and the concept of extensible test generation environment is described. Section 4 deals with the architecture of the environment and gives an overview of its basic components—*modeling environment* and *testing environment*. In Section 5, the modeling environment and its components *compiler* and *modeling library* are considered. Section 6 describes the testing environment and its components *test template processor*, *test library*, and *constraint resolution mechanism*. Section 7 summarizes the results and indicates directions for further studies.

2. TEST GENERATION METHODS

A variety of test generation methods have been proposed. They can be subdivided into two large classes: (1) *manual development* and (2) *automatic generation*. Presently, manual development is used for testing the situations in microprocessor behavior that are difficult to formalize or are hardly probable. However, manual development is rarely used for systematic verification of microprocessors. Automatic generation methods can be classified as follows: (1) *random generation*, (2) *combinatorial generation*, (3) *template-based generation*, and (4) *model-based generation*.

Random generation is the most widespread method for generating complex even though not directed tests for microprocessor verification. Despite the ease of implementation, this method can generate an intensive flow of stimuli thus helping detect nontrivial bugs. A well-know generator of this type is RAVEN (Random Architecture Verification Engine) developed by the Obsidian Software company (which was later purchased by ARM). This tool not only uses random code generation, but also takes into account typical design errors. RAVEN is based on predeveloped test generators, but it can be extended with user-defined components [3]. Unfortunately, no implementation details of this tool are available.

Another approach to test generation is *combinatorial generation*. The analysis of bugs in microprocessors shows that many of them can be detected using short test examples consisting of 2–5 instructions. Therefore, it is reasonable to search through short instruction sequences (including *test situations* for individual instructions and *dependences* between pairs of instructions) [2]. This method was implemented in the first version of MicroTESK (Microprocessor Testing and Specification Kit) developed in the Institute for System Programming, Russian Academy of Sciences. This tool supports hierarchical decomposition of the test generator into *iterators* (of which each is responsible for the search through a specific part of the test) and *composers* (which combine the results produced by internal iterators into more complex test sequences). In addition, MicroTESK can generate tests containing *control transfer instructions*. For that purpose, various control flow graphs are constructed and possible execution paths of limited length are searched through for each of them [7].

The next approach is *template-based generation*. A *test template* is an abstract representation of the test program in which the specific values of instruction operands (as is the case in an ordinary program) are replaced with *constraints* (*symbolic values*). When generating a test, the generator tries to find a random solution of the system of constraints (this approach is often called *constraint-based random generation* [8]). Due to the automation of a bulk of routine work, this method considerably improves the performance of verification engineers. The most well-known generator of this type is Genesys-Pro (IBM Research) [4]. This tool uses two types of input data: (1) a *model*, which describes the microprocessor architecture, and (2) *test templates*, which specify testing scenarios. Genesys-Pro generates a test sequentially; that is, an instruction to be included in the test program is chosen at each step, and then the system of constraints imposed on its operands is formulated and solved.

In distinction from the approaches described above, the *model-based generation* uses formal models of microprocessors to generate tests (or test templates). To clarify the terminology, we note that there are two types of microprocessor models: (1) *instruction-level* (*behavioral*) *models* and (2) *microarchitecture* (*structural*) *models*. The models of the first type specify the instruction set of a microprocessor (this is a view from outside). The models of the second type specify the structure organization of a microprocessor (this is a view from inside). All test generation methods use instruction-level models explicitly or implicitly; however, only few methods use microarchitecture models (exactly these methods are called model-based). Below we consider some of such approaches.

In [5], a directed test generation method was proposed. It uses detailed specifications of the microprocessor represented in the EXPRESSION language [9] and then translates them into SVM (Symbolic Model Verifier) [10]. Specifications determine the microprocessor structure (its components and connections between them), its behavior (semantics of the instructions), and the relationship between the structure and

behavior. The key part of paper [5] is the *fault model*, which describes typical design errors (errors in individual operations, in interaction of concurrently executing operations, etc.). Given the fault model, a set of formulas is generated for the microprocessor. Each formula determines the condition under which a particular error occurs. For each formula, a test is constructed using SVM (and the model checking method underlying SVM); in fact, this is a counterexample for the negation of the formula. Then, this counterexample is transformed into a test program. In the opinion of the authors of [5], this method cannot be scaled for complex microprocessors. To complement this method, the test template approach is proposed. Test templates are developed manually so as to describe chains of instructions that specify certain situations in the microprocessor behavior (primarily, pipeline conflicts). Tests are generated using a graph model of the microprocessor derived from the specifications.

In [11], the microprocessor microarchitecture is represented in the form of an operation state machine (OSM). The OSM includes two layers: (1) *operational level* and (2) *hardware level*. At the first level, the logic of the step-by-step execution of operations is described—each operation is described by an individual extended finite state machine (EFSM). At the second level, hardware resources of the microprocessor are described—each resource is assigned a token. The tokens are controlled by special token *managers*. As transitions in the OSM are performed, the OSMs can capture and free tokens using token managers. The microprocessor model is defined as a combination of operation and resource state machines. Test programs are generated by traversing all reachable states and transitions of the combined OSM.

3. CONCEPT OF EXTENSIBLE ENVIRONMENT FOR TEST GENERATION

Let us analyze the test generation methods and tools described above and define the concept of extensible environment for test generation. *Extensibility* is a characteristic of the environment indicating how much effort is needed to integrate into it a new or existing component (in our case, this is a microprocessor model or test generator). The less effort is needed, the better the extensibility of the environment. The aim of the present paper is to propose an architecture of the test generation environment that minimizes the effort needed to create new components and integrate them into the environment.

Extensible environments are typically built on the following principle. They have a *kernel* (*platform*) and *extensions* (*plugins*) interacting with the kernel at predetermined *extension points*. It goes without saying that the interfaces between the kernel and extensions, as well as the methods of installation of the extensions and their invocation from the environment for accomplishing specific tasks must be clearly defined. In our opinion, open source code positively affects the environment extensibility because the availability of code can simplify the development of extensions.

All the approaches to test generation use *instruction-level models* explicitly or implicitly (to create a test, one must know *preconditions of instructions* and their *assembler format*); however, the more complex test problems, the more complex models are needed to solve them. In our opinion, the kernel of the test generation environment should be based on instruction-level models and the corresponding test generation methods (random, combinatorial, and template-based test generation). More specific models and generators based on them should be in the form of extensions. The test generation environment can be conveniently represented as consisting of two parts: (1) *modeling environment* and (2) *testing environment*.

The kernel of the modeling environment makes it possible to describe *registers* (variables that store bit vectors of fixed size), *memory* (an array of words), and *instructions* (atomic actions on registers and memory). More detailed specifications are represented by *extensions of the model*, which are connected to the environment through the *extension points* (memory access handler, instruction invocation handler, etc.). The modeling environment supports standard extensions for the description of typical subsystems, such as *memory control unit* (description of the cache memory hierarchy, address translation mechanism, etc.) and *pipeline control unit* (descriptions of the pipeline stages, control flow switches, etc.). For standard extensions, one can install additional extensions (for example, to specify the strategy of cache row replacement or to describe the behavior of pipeline stages).

The testing environment consists of generators of two types: (1) *test sequence generators* and (2) *test data generators*.

The main types of test sequence generators are random and combinatorial generators [2, 3]. This is because the modeling environment is based on instruction-level models, which do not allow the use of more intricate test generation methods. An important property of the testing environment is the support of composition of test programs [12]. Suppose that there are two test programs (or two test templates) aimed at the creation of different and relatively independent situations in the operation of the microprocessor. Then, a combination of these programs can lead to the simultaneous (or close in time) occurrence of these situations. More complex model-based test generators can be installed in the environment together with the corresponding modeling tools. Note that the extension of the modeling environment is typically accompanied with an extension of the testing environment (when a new type of models is added, one should indicate how tests based on them can be generated).

A promising approach to test data generation is the generation based on constraint resolution (as is done in Genesys-Pro [4]). It is assumed that test situations

are represented in terms of *constraints* on the values of instruction operands and on the state of the microprocessor. An advantage of this approach is the capability of combining test situations by combining their constraints. In distinction from Genesys-Pro with its specialized constraint solver, we propose to use universal solvers, such as Yices [13] or Z3 [14], which support the SMT-LIB language [15]. In addition, the kernel of the generator can be extended by *user-defined test data generators*, which is useful in the case when test situations are difficult to represent in the form of constraints. The testing environment includes a library of built-in random and directed test generators (for example, for testing floating point arithmetic units [16]).

4. ARCHITECTURE OF THE MICROTESK ENVIRONMENT

The MicroTESK test generation environment includes two basic parts: (1) *modeling environment* and (2) *testing environment*. The purpose of the modeling environment is to describe the target microprocessor (describe the *microprocessor model*) and to specify the *test coverage model*. The model (the microprocessor model and test coverage model) is derived from formal specifications written in the *architecture description language* (ADL). In turn, the testing environment is responsible for the generation of test programs for the target microprocessor based on the information provided by the model. The goals of testing are defined in test *templates* written in the *template description language* (TDL).

The MicroTESK modeling environment consists of the following components: (1) *compiler*, which analyzes formal specifications in ADL and creates a model, and (2) *modeling library*, which contains interfaces that must be implemented in the model and standard blocks from which the model can be constructed (see Fig. 1). The compiler includes two backend components: (1) *Model generation module*, which constructs the executable model of the microprocessor, and (2) *coverage extraction module*, which derives a test coverage model for microprocessor instructions. Respectively, the modeling library consists of the *microprocessor modeling library* and (2) *coverage description library*.

The MicroTESK testing environment consists of the following components: (1) *test template processor*, which generates test programs from templates written in TDL; (2) *test library*, which contains a variety of *test sequence generators* and *test data generators* used by the test template processor; and (3) *constraint solver*, which provides to the test data generators an interface for interacting with external SMT solvers.

The components of the environment are not solid assemblies; they can be extended by modules designed for executing specific tasks. All the extensions related to one microprocessor subsystem are usually joined into one plugin (see Fig. 2). To extend the environment with modeling and testing tools for a certain subsystem, extensions of the components of the environment must be developed, including the *modeling library* (containing standard blocks for modeling this subsystem), *testing library* (containing generators of tests aimed at the verification of this subsystem), *specification language*, and *compiler* (which make it possible to describe the subsystem under examination in a certain language).

The functional capabilities of the environment can be subdivided according to the following basic subsystems: (1) *instruction set*, (2) *memory management*, and (3) *pipeline management*. The tools of the environment corresponding to the first subsystem form the kernel of the environment; they are intended for modeling the microprocessor instructions and generating test programs based of instruction-level models (random, combinatorial, and template-based generators are used). The modeling and testing of memory and pipeline management mechanisms are supported by standard extensions of the environment. The other subsystems are supported by user-defined extensions.

5. MICROTESK MODELING ENVIRONMENT

The MicroTESK modeling environment is designed for representing knowledge about the target microprocessor and for transferring this knowledge to the testing environment. The general procedure of the modeling environment operation is as follows: (1) a verification engineer develops *formal specifications* of the microprocessor; (2) these specifications are processed by the *compiler*, which creates the model of the microprocessor using the *modeling library*. Let us consider the components of the modeling environment in more detail.

5.1. Compiler

The compiler processes the *formal specifications* of the microprocessor and builds a *microprocessor mode*l and *test coverage model* using, respectively, the model generation unit and the coverage extraction unit; inbuilt modeling libraries of microprocessors and coverage descriptions are also used. Note that microprocessor specifications may be written in several languages of which each is responsible for a certain subsystem. The main part of specifications is related to the instruction set of the microprocessor; the other parts describe memory management, pipeline management, and other subsystems. Due to the inhomogeneity of specifications, the compiler is actually a collection of tools of which each processes a corresponding part of specifications.

Presently, MicroTESK supports only one ADL language for the specification of the instruction set; this is Sim-nML [17, 18]. Below, a code describing the integer addition (ADD) from the MIPS instruction set [19] in this language is presented. We want to note the following facts. (1) The function UNPREDICTABLE

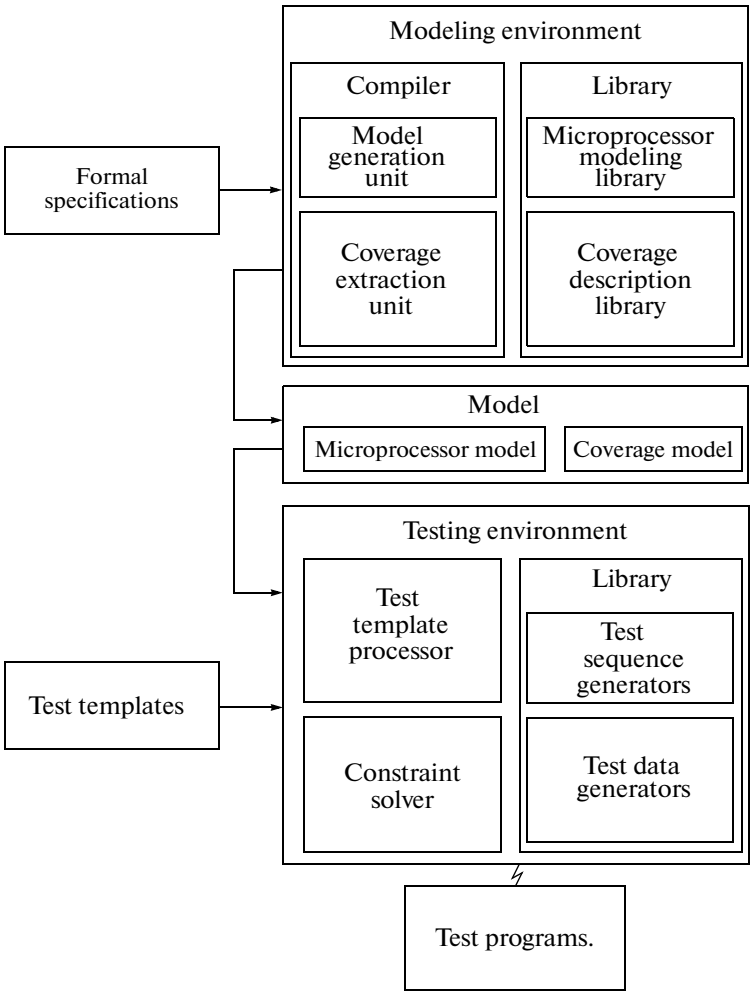

Fig. 1. Block diagram of the MicroTESK modeling environment.

may be used in specifications to indicate the situations in which the microprocessor behavior is undetermined. (2) By analyzing control flows in instruction specifications, one can automatically extract a model of test coverage. (3) Instructions can be joined into groups forming abstract instructions, which can be used in test templates. (4) Specifications are executable and deterministic; as a result, the environment can predict the results of program execution [12].

```
op ADD(rd: GPR, rs: GPR, rt: GPR)
action = {
  if(NotWordValue(rs) || NotWordValue(rt))
  then
    UNPREDICTABLE();
  endif;
  tmp = rs<31..31>::rs<31..0> + rt<31..31>::rt<31..0>;
  if(tmp<32..32> != tmp<31..31>)
  then
    SignalException("IntegerOverflow");
  else
    rd = sign_extend(tmp<31..0>);
  endif;
}
syntax = format("add %s,%s,%s",
    rd.syntax, rs.syntax, rt.syntax)
op ALU = ADD | SUB | ...
```

5.2. Microprocessor Modeling Library

The microprocessor modeling library is designed for creating executable models of microprocessors that are used by the environment for interpreting test programs and for tracking the state of the model in the course of test generation. The state tracking is needed for the generation of *self-verifiable tests* (programs with built-in checks of the microprocessor state). Thus, the microprocessor model includes an *instruction interpreter* and *functions for accessing the state of the model*. In addition, the model provides *metadata* that describe the elements of the microprocessor that are visible to programs, such as registers, memory, instructions, etc. Metadata is the main interface

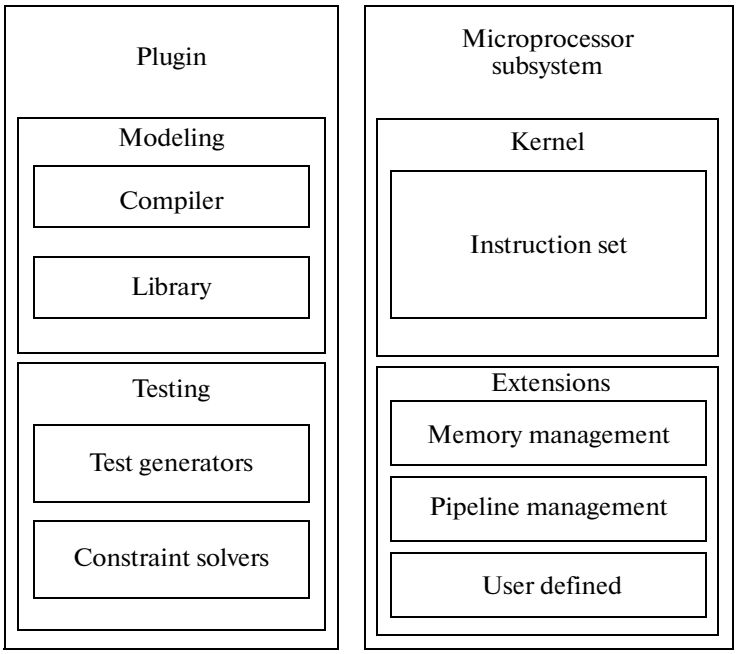

Fig. 2. Subsystems and organization of plugins in MicroTESK.

between the modeling environment and the test template processor.

The design library has several *extension points* for adding plugins. The set of extension points includes (1) the *memory access handler* and (2) the *instruction execution handler*. Handler (1) is invoked at each memory access for reading or writing. It encapsulates the memory management logic, including address translation and caching. Handler (2) is invoked each time an instruction starts executing. Using this handler, one can simulate the microprocessor pipeline by subdividing instructions into microoperations and scheduling their execution.

5.3. Coverage Description Library

The *coverage description library* makes it possible to determine situations that can occur in the microprocessor operation (overflow, cache hit, cache miss, pipeline conflict, etc.). The set of *test situations* called *test coverage model* provides a basis for test generation, mainly, for test data generation for individual instructions. In addition to *test situations*, the test coverage model includes *rules for instruction grouping*, which classify the microprocessor instructions according to certain categories (by operand types, used resources, structure of the control flow, and so on). As well as the microprocessor model, the test coverage model provides metadata concerning its elements.

Each test situation is assigned a unique name, which can be used to indicate it. The situation names are associated with generators; therefore, the test template processor knows which generator should be used for creating one or another test situation. The description of each situation includes the complete information needed for the generator to create it. The built-in test data generator uses the description of situations in the form of constraints (test data are generated by resolving constraints) [20].

6. MICROTESK TESTING ENVIRONMENT

The MicroTESK testing environment is responsible for generating test programs. The generic operation of the testing environment is as follows. (1) An engineer creates a *test template* and feeds it at the entry of the environment; this template describes a microprocessor testing scenario. (2) The *test template processor* applies *test sequence generators* for this template and builds a test program in *symbolic form*, where specific values of operands are replaced by names of test situations. (3) The processor invokes test data generators to create specific values for instruction operands. (4) The resultant test program is supplemented with *control code*, which initializes registers and memory with the generated data. Let us consider the components of the testing environment in more detail.

6.1. Test Template Processor

The test template processor transforms *test templates* into *test programs* using test sequence and test data generators registered in the environment. The supported TDL is organized as a library in Ruby [21]. This language allows one to describe sequences of instructions in assembler using the *metadata* provided

by the *microprocessor model*. In addition, this language supports high-level constructs for describing test scenarios, which can be classified into two types: (1) *built-in Ruby instructions* (conditional expressions, loops, etc.) and (2) *special MicroTESK instructions* (blocks for generating test sequences, references to test situations, etc.) Here is a simple example of a test template.

```
# Assembler code
add r[1], r[2], r[3]
sub r[1], r[1], r[4]
# Ruby control constructs
(1..3).each do |i|
    add r[i], r[i+1], r[i+2]
    sub r[i], r[i], r[i+3]
end
# Test sequence generation block
block (:engine ⇒ ”random”, :count ⇒ 2013)
{
    add r[1], r[2], r[3]
    sub r[1], r[2], r[3]
        # Reference to test situation
        do overflow end
}
```

An important construct used in test templates is the *test sequence generation block*. The test template is a hierarchical structure consisting of test sequence generation blocks. Each block contains instructions and nested blocks and specifies a *test sequence generator* and its parameters. The test template processor creates test sequences for the nested blocks using the appropriate generators and then combines the resultant test sequences and composes a test program from them (an example is given in Subsection 6.2).

Note that the test template processor supports the creation of self-verifying tests. When building a test program, it can add special code (called *test oracle*) that verifies the correctness of the microprocessor state at the corresponding point of the program. The test oracle compares the data in the memory and registers with the reference values calculated by the *instruction interpreter*; if these values do not coincide, the oracle reports an error.

6.2. Test Sequence Generators

Each *test sequence generator* implements a method iterating through chains of instructions. Each block of the test template is assigned a test sequence generator. Since blocks may be nested, generators can be recursively combined. For this purpose, each nonterminal block must determine two strategies: (1) *combining* the results produced by the nested generators and (2) *composition* (merging) of several instruction chains into a unified test sequence.

Testing library contains predefined combination and composition strategies. The following combining methods are supported: (1) *random combination* (random combinations of the results produced by generators are created), (2) *Cartesian product* (all possible combinations of the results produced by generators are created), (3) *diagonal of the Cartesian product* (to create a combination, the nested generators are called synchronously). The supported composition methods are as follows: (1) *random composition* (test sequences are mixed randomly), (2) *concatenation* (test sequences are concatenated), (3) *nesting* (test sequences are nested one into another). Users may add test sequence generators and combination and composition strategies to the environment. Consider a simple example.

```
# Test sequence generation block
block(:combine ⇒ “product”, :compose ⇒ “random”) {

    # Nested block A
    block(:engine ⇒ “random”, :length ⇒ 3, :count ⇒ 2) {
        add r[a], r[b], r[c]
        sub r[d], r[e], r[f]
        mult r[g], r[h]
        div r[i], r[j]
    }
    # Nested block B
    block(:engine ⇒ “permutate”) {
        ld r[k], r[l]
        st r[m], r[n]
    }
}
```

In this example, the upper level block contains two nested blocks A and B. Block A consists of four instructions ADD, SUB, MULT, and DIV. Block B consists of two instructions LD and ST. The generator of test sequences associated with A produces two sequences (:count ⇒ 2) of length 3 (:length ⇒ 3) composed of the instructions listed above in a random fashion (:engine ⇒ “random”). The generator associated with B produces all permutations of the specified instructions (:engine ⇒ “permutate”). The upper level generator creates all possible combinations of the results produced by the nested blocks (:combine ⇒ “product”) and mixes them in a random fashion (:compose ⇒ “random”). The result of processing such a template can look as follows.

```
    # Combination (1,1)
    sub r[d], r[e], r[f]        # Block A
    ld r[k], r[l]               # Block B
    div r[i], r[j]              # Block A
    st r[m], r[n]               # Block B
    add r[a], r[b], r[c]        # Block A

    # Combination (1,2)
    st r[m], r[n]           # Block B
    sub r[d], r[e], r[f]    # Block A
    ld r[k], r[l]           # Block B
```

```
div r[i], r[j]          # Block A
add r[a], r[b], r[c]    # Block A

# Combination (2,1)
mult r[g], r[h]         # Block A
mult r[g], r[h]         # Block A
ld r[k], r[l]           # Block B
add r[a], r[b], r[c]    # Block A
st r[m], r[n]           # Block B

# Combination (2,2)
mult r[g], r[h]         # Block A
st r[m], r[n]           # Block B
mult r[g], r[h]         # Block A
ld r[k], r[l]           # Block B
add r[a], r[b], r[c]    # Block A
```

6.3. Test Data Generators

Test data generators should produce values of the operands of instructions based on the test situations specified in the template. MicroTESK supports generation of test data using *resolution of constraints*. To calculate the values of instruction operands, the *test template processor* chooses a test data generator corresponding to the situation and requests the state of the resources used in the test situation (registers, memory, etc.) from the *microprocessor model*. Then, the processor initializes the corresponding variables and invokes the constraint solver.

As soon as the values of the operands have been obtained, *control code* is added to the test program, which initializes the corresponding resources of the microprocessor. For example, if the operand of an instruction is a register, then the control code puts the corresponding value into it. Following the *concept of constraint-based random generation*, different calls to test data generators may produce different sets of operand values (however, each set must satisfy the prescribed system of constraints).

6.4. Constraint Resolution Mechanism

The constraint resolution mechanism is implemented using a set of solvers, which are accessed via a unified interface. Constraint solvers are classified into two main types: (1) *universal solvers* that can be used for a wide class of constraints and (2) *user-defined solvers* designed for specific test data generation.

The first type includes SMT solvers based, for example, on Yices [13] and Z3 [14]. They support Boolean operations, integer arithmetic, operations on bit vectors of fixed length, etc. The interaction with universal constraint solvers is implemented using the library Java Constraint Solver API [20]. This library makes it possible to create constraints in the form of Java objects, represent them in SMT-LIB [15], and invoke an external SMT solver.

Consider an example of a constraint written in SMT-LIB that describes the situation of overflow in the addition instruction of 32-bit words. First, types (*define-sort*), functions (*define-fun*), and variables (*declare-const*) are declared. Next, preconditions for the variable values and the target constraint are specified (*assert*).

```
; Types and functions
(define-sort Int_t () (_ BitVec 64))
(define-fun INT_ZERO () Int_t (_ bv0 64))
(define-fun INT_BASE_SIZE () Int_t (_ bv32 64))
(define-fun INT_SIGN_MASK () Int_t
   (bvshl (bvnot INT_ZERO) INT_BASE_SIZE))
(define-fun IsValidPos ((x!1 Int_t)) Bool
   (ite (= (bvand x!1 INT_SIGN_MASK)
   INT_ZERO) true false))
(define-fun IsValidNeg ((x!1 Int_t)) Bool
   (ite (= (bvand x!1 INT_SIGN_MASK)
   INT_SIGN_MASK) true false))
(define-fun IsValidSignedInt ((x!1 Int_t)) Bool
   (ite (or (IsValidPos x!1) (IsValidNeg x!1)) true
   false))

; Variables
(declare-const a Int_t)
(declare-const b Int_t)

; Preconditions
(assert (IsValidSignedInt a))
(assert (IsValidSignedInt b))

; Constraint
(assert (not (IsValidSignedInt (bvadd a b))))
```

Some test situations are difficult to describe using constraints (this is the case for floating point arithmetic [22], memory management [23], and some others). To describe and handle such situations, user-defined test data generators may be added to the environment.

7. CONCLUSIONS

Architecture of extensible environment for the generation of test programs for microprocessors is proposed. The proposed approach was implemented in the MicroTESK environment (Institute for System Programming, Russian Academy of Sciences). The developed platform makes it possible to join various methods of microprocessor modeling and testing. This environment is based on instruction-level models; it supports random, combinatorial, and template-based generation of test programs. More complex types of models and test generators can be added to the environment as plugins. The formal specification of the microprocessor instruction set is done in the language Sim-nML. Based on the analysis of these specifications, a model (of the microprocessor and test cover-

age) is built. Templates of test programs are described in Ruby so that instructions, test situations, and other elements of the models can be accessed. The language makes it possible to describe complex situations of testing microprocessors, and it supports composition of several test programs. In near future, we plan to use MicroTESK to develop test generators for widely used microprocessor architectures, including ARM and MIPS. We also work on extensions of the environment with tools for modeling and testing typical subsystems of microprocessors (address translation buffers, cache modules, control units, etc.).

ACKNOWLEDGMENTS

This work was supported in part by the Ministry for Education and Science of the Russian Federation, project no. 8232 as of August 6, 2012.

REFERENCES

1. M.S. Abadir and S. Dasgupta, Guest editors' introduction: Microprocessor test and verification, *IEEE Design & Test Comput.*, 2000, vol. 17, no. 4, pp. 4–5.
2. Kamkin, A.S., Test program generation for microprocessors, in *Trudy Instituta Sistemnogo Programmirovaniya Ross. Akad. Nauk*, 2008, vol. 14, part 2, pp. 23–63.
3. http://www.arm.com/community/partners/displayproduct/rw/ProductId/5171/
4. Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M., and Ziv. A., Genesys-Pro: Innovations in test program generation for functional processor verification, *IEEE Design & Test Comput.*, 2004, vol. 21, no. 2, pp. 84–93.
5. Mishra, P. and Dutt, N., Specification-driven directed test generation for validation of pipelined processors, *ACM Trans. Design Autom. Electron. Syst. (TODAES)*, 2008, vol. 13, no. 3, pp. 1–36.
6. http://forge.ispras.ru/projects/microtesk
7. Kamkin, A.S., Some issues of automation of test program generation for branch units of microprocessors, in *Trudy Instituta Sistemnogo Programmirovaniya Ross. Akad. Nauk*, 2010, vol. 18, pp. 129–150.
8. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., and Shurek, G., Constraint-based random stimuli generation for hardware verification, *AI Magazine*, 2007, vol. 28, no. 3, pp. 13–30.
9. Grun, P., Halambi, A., Khare, A., Ganesh, V., Dutt, N., and Nicolau, A., EXPRESSION: An ADL for system level design exploration, *Technical Report 1998-29*, Univ. of California, Irvine, 1998.
10. http://www.cs.cmu.edu/~modelcheck/smv.html
11. Dang, T.N., Roychoudhury, A., Mitra, T., and Mishra, P., Generating test programs to cover pipeline interactions, in *Design Automation Conference (DAC)*, 2009, pp. 142–147.
12. Kamkin, A., Kornykhin, E., and Vorobyev, D., Reconfigurable model-based test program generator for microprocessors, in *Software Testing, Verification, and Validation Workshops (ICSTW)*, 2011, pp. 47–54.
13. Dutertre, B. and Moura, L., The YICES SMT solver, 2006. http://yices.csl.sri.com/tool-paper.pdf
14. Moura, L. and Bjoslashrner, N., Z3: An efficient SMT solver, in *Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.
15. Cok, D.R., The SMT-LIBv2 Language and Tools: A Tutorial, GrammaTech, Inc., 2011, Version 1.1.
16. Aharoni, M., Asaf, S., Fournier, L., Koifman, A., and Nagel, R., FPgen—A test generation framework for datapath floating-point verification, in *High Level Design Validation and Test Workshop (HLDVT)*, 2003, pp. 17–22.
17. Freericks, M., The nML machine description formalism, *Technical Report, Bericht 1991/15*, Technische Universitaumlautt Berlin, 1991.
18. Moona, R., Processor models for retargetable tools, in *Int. Workshop on Rapid Systems Prototyping (RSP)*, 2000, pp. 34–39.
19. *MIPS64™ Architecture for Programmers*, Vol. II: *The MIPS64™ Instruction Set,* Document Number: MD00087, Revision 2.00, June 9, 2003.
20. http://forge.ispras.ru/projects/solver-api.
21. http://www.ruby-lang.org.
22. Kamkin, A.S. and Chupilko, M.M., Testing microprocessor floating point arithmetic modules for conformity to the IEEE 754 standard, in *Trudy Instituta Sistemnogo Programmirovaniya Ross. Akad. Nauk*, 2008, no. 2, pp. 7–22.
23. Kornykhin, E.V., Generation of test data for verification of caching mechanisms and address translation in microprocessors, *Program. Comput. Software*, 2010, vol. 36, no. 1, pp 28–35.

Translated by A. Klimontovich