MicroTESK: An Extendable Framework for Test Program Generation

Alexander Kamkin*, Tatiana Sergeeva*, Andrei Tatarnikov*† and Artemiy Utekhin‡
* Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
† National Research University Higher School of Economics (NRU HSE)
‡ Moscow State University (MSU)
Email: {kamkin,leonsia,andrewt,utekhin}@ispras.ru

***Abstract*— Creation of test programs and analysis of their execution is the main approach to system-level verification of microprocessors. A lot of techniques have been proposed to automate test program generation, ranging from completely random to well directed ones. However, no "silver bullet" has been found. In good industrial practices, various methods are combined complementing each other. Unfortunately, there is no solution that could integrate all (or at least most) of the techniques in a single framework. Engineers are forced to use a number of tools, which leads to the following problems: (1) it is required to maintain duplicating data (each tool uses its own representation of the target design); (2) to be used together, tools need to be integrated (engineers have to deal with different formats and interfaces). This paper proposes a concept of an extendable framework (MicroTESK) that follows a unified methodology for defining test program generation techniques. The framework supports random and combinatorial generation and (what is even more important) can be easily extended with new techniques being implemented as the framework's plugins.**

I. Introduction

Being extremely complex, modern microprocessors require systematic activities for ensuring their *correctness* and *reliability*. Such activities are usually referred to as *verification* and *testing* [1]. The first of them, verification, is applied in the development stage and focuses on discovering *logical faults* in microprocessor designs (functional faults, interface faults, etc.). The second one, testing, is related to the manufacturing stage and deals with diagnosing *physical faults* in integrated circuits (stuck-at faults, bridging faults, etc.). The general approach to both tasks is based on execution of *verification/test programs*, which are assembly programs causing some *situations* in the design (internal events, component interactions, etc.) [2]. (Throughout the paper we will use the term *testing* to denote both verification and testing.)

By the present time, a great number of techniques for automated test program generation have been proposed. All of them can be subdivided in the following categories: (1) *random generation* [3], (2) *combinatorial generation* [2], (3) *template-based generation* [4] and (3) *model-based generation* [5]. The thing is that there is no a "silver bullet", which can effectively "fight" against all kinds of testing tasks. In real-life practice, different approaches are used together complementing and strengthening each other. It is a typical solution, for example, when general functionality of a microprocessor is tested by randomly generated programs, while critical logic is verified by advanced model-based techniques.

This work was supported in part by the Ministry of Education and Science of the Russian Federation under grant #8232 (06/08/2012).

Unfortunately, there is no framework that could accomodate a variety of test program generation techniques. Engineers have to use a number of tools with different input/output formats, and it is a big problem how to integrate them and keep their configurations in consistent states. It is not difficult to use different tools for solving loosely connected tasks, but settling tightly dependent problems by means of two or more tools might require deep knowledge of their internal interfaces. The root of the problem is that different tools use different representation of the target design, and often one representation is hidden from others. As a result, similar things are specified several times, duplicating data and complicating tests maintenance.

We propose a concept of an *extendable test program generation framework*, named MicroTESK [6]. The idea is to represent knowledge about the microprocessor under test (a *design/coverage model*) in a general way and to provide easy access to that knowledge to a number of test generators built as the *framework's plugins*. Being shared among various tools, a common model serves as a natural interface for their integration. Moreover, we have unified test generator interfaces, making it possible to use different tools for solving a testing task and even to combine them for doing complex jobs. Interaction between the framework and engineers is done with the help of *test templates* that specify test scenarios in a hierarchical manner (dividing testing tasks into smaller subtasks and linking each of them with an appropriate test generator).

The rest of the paper is organized as follows. Section 2 overviews existing test generation techniques and tools. Section 3 analyses the approaches described in Section 2 and formulates a concept of an extendable test program generation framework. Section 4 outlines the framework architecture and introduces two main components: a *modeling framework* and a *testing framework*. Section 5 considers the modeling framework and its components: a *translator* and a *modeling library*. Section 6 pays attention to the testing framework consisting of a *test template processor*, a *testing library* and a *constraint solver engine*. Section 7 concludes the paper.

II. Test Program Generation Techniques

There is a variety of techniques for test program construction. All of them can be divided into two types: (1) *manual test program development* and (2) *automated test program generation*. Nowadays, manually created tests are rarely used for systematic verification of microprocessors, but the approach is still in use for testing hardly formalizable and highly unlikely "corner cases" in microprocessor behavior. As for automated techniques, they can be subdivided into the following classes: (1) *random generation*, (2) *combinatorial generation*, (3) *template-based generation* and (4) *model-based generation*.

Random generation is the most common technique to produce complex (though unsystematic) test programs for microprocessor verification. Being easy to implement, the method, however, can create a significant workload to the microprocessor and is able to detect some high-quality bugs. One of the most famous generators of that type is RAVEN (Random Architecture Verification Engine) developed by Obsidian Software Inc (now acquired by ARM) [3]. To generate test programs, the tool not only applies *randomization*, but takes into account information about common microprocessor faults. RAVEN is built upon pre-developed and custom made modules that can be included into the generator to expand its functionality [3]. Unforfunately, due to the lack of publicly available information, technical details are unclear.

Another approach to test program construction is *combinatorial generation*. A brief analysis of microprocessor errata shows that many bugs can be detected by small test cases (2-5 instructions). Thus, it may be useful to systematically enumerate short sequences of instructions (including *test situations* for individual instructions and *dependencies* between instructions) [2]. The technique has been implemented in the first version of MicroTESK (ISPRAS). The tool supports hierarchical decomposition of a test program generator into *iterators* (each being responsible for iterating its own part of the test) and *combinators* (combining results of the inner iterators into complex test sequences). MicroTESK can also construct test programs with branch instructions (generation is done by enumeration of control flow graphs and bounded depth-first exploration of the execution traces) [7].

The next method is called *template-based generation*. A *test template* is an abstract representation of a test program, where *constraints* are used to specify possible values of instruction operands (instead of concrete values used in usual programs). When constructing a test program, a generator tries to find random solutions to the given constraint systems (such approach is usually referred to as *constraint-based random generation* [8]). Automating routine work, the method considerably increases productivity of engineers. The leading generator of that kind is Genesys-Pro (IBM Research) [4]. This is an architecture-independent tool that uses two types of input data: (1) a *model* (containing architecture-specific information) and (2) *test templates* (describing test scenarios). Genesys-Pro generates test programs in an instruction-wise manner: at each step, it selects an instruction to be put into a program and, then, formulates and solves a constraint system for the chosen instruction.

As opposed to the previously described approaches, *model-based generation* uses microprocessor models to compose test programs (or test templates). It is worthwhile clarifying the terminology. There are two main types of models used in microprocessor design and test: (1) *instruction-level models* (*behavioral models*) and (2) *microarchitectual models* (*structural models*). Models of the first type specify microprocessors as instruction sets (in other words, they describe a programmer's view to microprocessors: 'How to write programs for a microprocessor?'). Models of the second type define the internal structure of microprocessors (this is a computer engineer's point of view: 'How a microprocessor is organized inside?'). All test program generation methods and tools apparently use instruction-level models, but only few of them use microarchitectural models. The latter ones are called *model-based*. Let us consider some examples.

In [5], a method for directed test program generation has been proposed. It takes detailed microprocessor specifications written in the EXPRESSION language [9] and translates them into the SMV (Symbolic Model Verifier) description [10]. Specifications define the structure of the design (components and their interconnections), its behavior (semantics of the instructions) and mapping between the structure and the behavior. The key part of the work is a fault model describing typical errors for registers, individual operations, pipeline paths and for interactions between several operations. For each of the fault types, the set of concrete properties is generated, each of which is covered by a test case constructed by SMV (as a counterexample for the negation of the property). The generated test cases are mapped into test programs. As the authors say, the technique does not scale well on complex microprocessor designs. Thus, they suggest using the *template-based* approach as an addition. Test templates are developed by hand and describe sequences of instructions that create certain situations in the design's behavior (first of all, *pipeline hazards*). Test program generation is performed with the help of the graph model extracted from the specifications. It should be noticed that both approaches are based on rather accurate specifications, and it is better to apply them in the late design stages when the microarchitecture is stable.

In [11], a microprocessor is formally specified as an *operation state machine* (*OSM*). The OSM is a system of communicating extended finite state machines (EFSMs) modeling the microprocessor at two levels: (1) the *operational level* and (2) the *hardware level*. At the first level, movement of the instructions across the pipeline stages is described (each operation is specified by an EFSM). At the second level, hardware resources are modeled using so-called *token managers*. An operation EFSM changes its state by capturing/releasing tokens of the token managers. The pipeline model is defined as a concurrent composition of the operation EFSMs and resource EFSMs. Test programs are generated on-the-fly by traversing all reachable states and transitions of the joint OSM model.

III. Extendable Framework Concept

Let us analyze the test generation techniques and tools having been surveyed previously and formulate a concept of an extendable test program generation framework. It is worthwhile answering the questions: 'What is *extendability*?' and 'What is an *extendable* framework?' In general terms, *extendability* is a framework characteristic that shows how much effort it takes to integrate a new or existing component (in our case, a microprocessor model or a test generation engine) into the framework. Indeed, the less effort it is required, the more extendable the framework is. The object of this study is to suggest a framework architecture that would minimize the effort for creating new models/engines and plugging them into the framework.

There is a number of basic requirements that all kinds of extendable frameworks are expected to comply with. A system should be architected in such a way that there is a *core* (*platform*) and there are *extensions* (*plugins*) connected to the core via the *extension points*. Obviously, there should be well-defined interfaces between the core and its extensions as well as clear mechanisms for installing extensions into the framework and calling them for solving particular tasks. An optional requirement, which, we think, is essential for true extendable frameworks, is *open source*. The open source paradigm significantly simplifies creation and distribution of framework extensions.

There are also specific requirements to test program generation frameworks. Analyzing the approaches presented in the previous section, we can see that all of them use *instruction-level models* (either explicitly or implicitly). Evidently, to create a valid test program, one should know *instruction formats* and *instruction preconditions*. However, if more sophisticated programs need to be generated, more complicated models should be utilized (finite state machines, nets, etc.). In our opinion, the core should be formed around instruction-level models, while more specialized models/engines should be organized as framework extensions. Another suggestion is to divide the test program generation framework into two parts: (1) the *modeling framework* and (2) the *testing framework*, each having its own core and being extendable.

The core of the modeling framework allows describing *registers* (as variables storing fixed-size bit vectors), *memory* (as an array of machine words) and *instructions* (as atomic operations over registers and memory). More detailed specification is provided by so-called *model extensions*. There is a number of points that such extensions can be connected to (e.g., the memory access handler and the instruction execution handler). The framework supports standard extensions for specifying *memory management* (cache hierarchy, address translation mechanisms, etc.) and *pipelining* (interconnection between pipeline stages, control flow transfers, etc.). Note that the standard extensions, in turn, have extension points and can be easily customized by engineers (e.g., it is possible to define a cache replacement strategy or describe behavior of a pipeline stage).

The testing framework is comprised of engines of two types: (1) *test sequence generators* and (2) *test data generators*.

The main test sequence generators are *random* and *combinatorial generators* [2], [3]. It is explained by the fact that the modeling framework is organized around instruction-level models, which give no information on how to compose instructions into sequences for achieving particular testing goals. A useful feature is support for *test program composition* [12]. Given two test programs (or test templates) focusing on different (and more or less independent) situations, it may be interesting to shuffle them with the intention to cause situations to occur simultaneously or close to each other. More advanced *model-based generators* can be installed into the framework together with the corresponding models. Moreover, extension of the modeling framework shoud be always accompanied by the extension of the testing framework (if a new type of models is added into the framework, one should describe how to generate tests on the base of such models).

We think, it is a promising practice to use *constraint solvers* for test data generation (as it is done in Genesys-Pro [4]). It implies that test situations are expressed as *constraints* on the instruction operands and the microprocessor state. An important property of the approach is that situations can be easily combined by conjuncting the constraints. In contrast to Genesys-Pro, we suggest using general-purpose SMT solvers (like Yices [13] and Z3 [14]) supporting the unified SMT-LIB notation [15]. In addition, the generation core can be extended with *custom generators* (which are useful when situations are hardly expressible in terms of constraints). There is also a library of predefined generators including *random generators* and *directed generators* (e.g., for the floating-point arithmetic [16]).

IV. MicroTESK Framework Architecture

The MicroTESK framework is divided into two main parts: (1) the *modeling framework* and (2) the *testing framework*. The purpose of the modeling framework is to represent a model of the microprocessor under test (a *design model*) as well as model-based testing knowledge (a *coverage model*). The design/coverage model is extracted from *formal specifications* written in an *architecture description language* (*ADL*). The testing framework, for its turn, is responsible for generating test programs for the target microprocessor on the base of information provided by the model. Testing goals are defined in *test templates* written in a *template description language* (*TDL*).

The MicroTESK modeling framework consists of (1) a *translator* (analyzing formal specifications in an ADL and producing the microprocessor model) and (2) a *modeling library* (containing interfaces to be implemented by a model and classes to be used as building blocks) (see Figure 1). The translator includes two back-ends: (1) a *model generator* (constructing an executable design model) and (2) a *coverage extractor* (building a coverage model for the microprocessor instructions). In a similar fashion, the modeling library is split into *design* and *coverage libraries*.

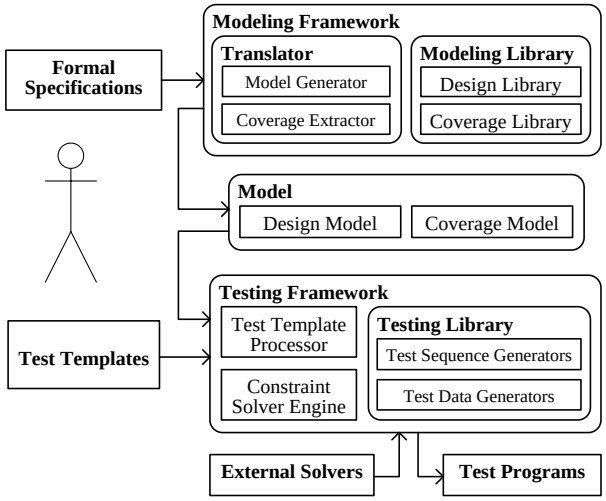

Figure 1. General structure of the MicroTESK framework

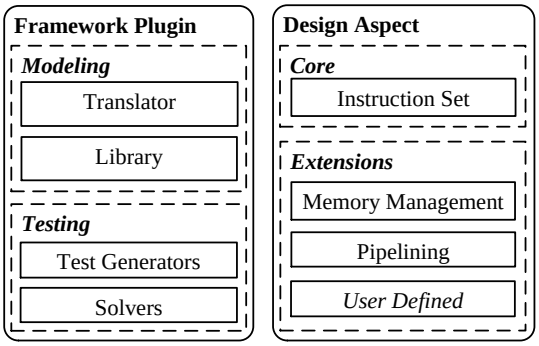

Figure 2. Design aspects and organization of a MicroTESK plugin

Components of the MicroTESK testing framework are as follows: (1) a *test template processor* (handling test templates written in a TDL and generating test programs), (2) a *testing library* (containing a wide range of *test sequence generators* and *test data generators* used by the test template processor) and (3) a *constraint solver engine* (providing the test generators of the testing library with a Java interface to external *SMT solvers*).

The framework components are not monolithic – they include some *core* functionality as well as *extensions* oriented to specific tasks. Such tasks are usually grouped according to the *design aspects* they deal with. All extensions related to the same aspect are united into a *framework plugin* (see Figure 2). To extend the framework with features aimed at modeling/testing a new design aspect, one should develop extensions for all of the framework components, including a *modeling library* (to provide building blocks for modeling the design aspect), a *testing library* (to let the framework know of how to test the design aspect) and a *specification language* coupled with a *translator* (to make it possible to express properties on the design aspect in a human-readable form).

The framework functionality is divided into the following aspects: (1) *instruction set*, (2) *memory management* and (3) *pipelining*. Forming the framework core, the first aspect is responsible for modeling microprocessor instructions and generating test programs on the base of the instruction-level models (random, combinatorial and template-based generators are in active use). Support for the next ones, memory management and pipelining, is implemented in the standard plugins. For other design aspects, custom plugins can be created and installed into the framework.

V. MicroTESK Modeling Framework

The purpose of the MicroTESK modeling framework is to represent knowledge about a microprocessor and share that knowledge with the testing framework. An engineer provides *formal specifications* of the microprocessor under test. The specifications are processed by the *translator* (including the front-end and two back-ends, the *model generator* and the *coverage extractor*). The translator produces the *model* by using the *modeling library* (comprising the *design* and *coverage libraries*). Let us consider the modeling framework components in more detail.

A. Translator

The *translator* processes *formal specifications* of the microprocessor and builds the *design/coverage model* (applying the model generator/coverage extractor and using the building blocks defined in the design/coverage library). Note that microprocessor specifications are written in a mixture of languages, each being responsible its own design aspect. The central part of specifications is related to the instruction set architecture; other parts describe memory management, pipelining, etc. As specifications are heterogeneous, the translator is actually represented as a set of tools processing their parts of specifications.

At the moment, Sim-nML [17], [18] is the only ADL supported by MicroTESK for specifying microprocessors at the instruction level. Sim-nML code specifying the integer addition instruction (`ADD`) from the MIPS instruction set [19] is shown below. Several things need to be emphasized: (1) specifications can use the predefined function `UNPREDICTABLE` to indicate the situations, where the design's behavior is undefined; (2) by analyzing control and data flows in instruction specifications one can automatically extract the coverage model; (3) instructions can be grouped together providing the framework with useful information to be used within test templates; (4) basing on such specifications, the framework is able to predict the result of the test 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_word<31..0>);
  endif;
}

syntax = format("add %s, %s, %s",
  rd.syntax, rs.syntax, rt.syntax)

op ALU = ADD | SUB | ...
```

B. Design Library

The *design library* is intended to represent a microprocessor model, which is used to simulate instruction execution and to keep track of the design state during test program generation. State tracking is essential for generating *self-checking tests* (i.e., programs with built-in checks of the microprocessor state). In addition to the *instruction simulator* (which simulates instructions and updates the model state) and the *state observer* (which provides access to the model state), the design model provides *meta-information* describing the design elements (registers, memory, instructions, etc.). The meta-information is the main interface between the modeling framework and the test template processor.

The design library has several *extension points* that allow engineers to connect their components. The set of extension points includes (1) the *memory access handler* and (2) the *instruction execution handler*. The handler of the first type is invoked every time a memory location is accessed for reading or writing. It may encapsulate memory management logic such as address translation and caching. The handler of the second type is launched when an instruction is executed. It is usually used to model the microprocessor pipeline – decomposition of instructions into microoperations and their scheduling.

C. Coverage Library

The *coverage library* is used to describe situations that can occur in a microprocessor (an overflow, a cache miss/hit, a pipeline bypass, etc.). Such a description (referred to as a *coverage model*) serves as a basis for generating test programs (especially, for creating test data for individual instructions of a program). Besides the *test situations*, the coverage model contains *grouping rules*, classifying microprocessor instructions according to some criteria (number of operands, resources being accessed, control flow structure, etc.). Similar to the design model, the coverage model provides meta-information on its elements, which is used by the test template processor.

Each test situation has a unique name that can be used in a test template to refer to the situation. There is a mapping of situation names onto test generators. Thus, the test template processor knows which engine to use to create a particular test case. To make the engine comprehend how it can be done, the situation include an engine-specific description of the condition/action causing the situation to occur. The most usable engine built-in into the framework uses the constraint-based description of situations and constraint solving [20].

VI. MicroTESK Testing Framework

The MicroTESK testing framework is responsible for generating test programs. An engineer provides a *test template* describing a test scenario for the microprocessor under test. The test template is handled by the *test template processor* by using the *engines* of the *testing library*: (1) it applies the *test sequence generators* to construct a *symbolic test program* (i.e., sequence of instructions annotated with test situations); (2) it requests the *test data generators* to generate concrete values of the instruction operands; (3) it instantiates the test program by inserting *control code* initializing the registers and the memory with the generated test data. Let us consider the testing framework components in more detail.

A. Test Template Processor

The *test template processor* is a runtime environment that handles a *test template*, chooses appropriate engines of the *testing library* and produces a *test program*. The supported TDL is organized as a Ruby [21] library. It allows describing instruction sequences in a way it is done in the assembly language (by using the *meta-information* provided by the *design model*) though supporting high-level scenario description constructs. The latter ones can be subdivided into two types: (1) *native Ruby constructs* (conditional statements, loops, etc.) and (2) *special MicroTESK constructs* (test sequence blocks, test situations, etc.). A simple test template example is given below.

```
# Assembly-Style Code
add r[1], r[2], r[3]
sub r[1], r[1], r[4]

# Ruby Control Statements
(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 Block
block(:engine => "random",
      :count  => 2013)
{
    add r[1], r[2], r[3]
    sub r[1], r[2], r[3]
        # Test Situation Reference
        do overflow end
}
```

An important notion used in test templates is a *test sequence block*. In fact, a test template is a hierarchical structure of test sequence blocks, each holding a set of instructions (or nested blocks) and specifying a *test sequence generator* (and its parameters) to be used to produce a test sequence. The test template processor constructs test sequences for the

nested blocks by applying the corresponding engines and then *combines/composes* the built sequences with the root engine (an example is given the section "*Test Sequence Generators*").

Another important feature of the test template processor is support for generation of *self-checking tests*. When constructing a test program, the test template processor can inject special pieces of code that check whether the microprocessor state is valid in the corresponding execution point. Such code (called a *test oracle*) compares data stored in the previously accessed registers and memory blocks with the reference data (calculated by the *instruction simulator*) and terminates the program if they do not match.

B. Test Sequence Generators

A *test sequence generator* is organized as an *iterator* of test sequences. In the simplest case, a test sequence generator returns a single test sequence for a single test sequence block. As blocks can be nested, generators can be *combined/composed* in a recursive manner. To do it, two strategies should be defined for each non-terminal block: (1) a *combinator* (describing how to combine the results of the inner iterators) and (2) a *compositor* (defining the method for merging several pieces of code together). Thus, a combinator produces the combinations of the inner test sequences, while a compositor merges those sequences into the one.

The *testing library* contains a variety of combinators and compositors. The most usable combinators are: (1) a *random combinator* (produces a number of random combinations of the inner iterators's results), (2) a *product combinator* (creates all possible combinations of the inner blocks' test sequences) and (3) a *diagonal combinator* (synchronously requests the inner iterators and joins their results). The set of implemented compositors include: (1) a *random compositor* (randomly mixes the inner test sequences), (2) a *catenation compositor* (catenates the inner test sequences) and (2) a *nesting compositor* (embeds the inner test sequences one into another). Note that engineers are allowed to add their own test sequence generators, combinators and compositors into the testing library and invoke them from test templates. Let us consider a simple example.

```
# Test Sequence 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 the example above, there is one top-level block containing two nested blocks, `A` and `B`. Block `A` consists of four instructions, `ADD`, `SUB`, `MULT` and `DIV`. Block `B` consists of `LD` and `ST`. The engine associated with `A` generates two sequences (`:count => 2`) of the length three (`:length => 3`) composed of the instructions listed in the block. The engine associated with `B` generates all permutations of the inner instructions (there are two permutations of two elements). The top-level engine produces all possible combinations of the nested blocks' sequences (`:combine => "product"`) and randomly mixes them (`:compose => "random"`). The result may 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
```

C. Test Data Generators

A symbolic test program produced by *test sequence generators* does not necessarily define values of all of the instruction operands (leaving some of them either undefined or deliberately ambiguous). The job of *test data generators* is to construct operand values on the base of the provided test situations. Test data generation relies on the *constraint solver engine* that constructs operand values by solving the corresponding constraints. To achieve a given test situation,

the *test template processor* selects an appropriate test data generator and requests the *design model* for the state of the involved design elements. After that, it initializes the closed variables of the constraint (variables whose values are defined by the previouly executed instructions) and calls the constraint solver engine to construct the free variables' values.

As soon as the operand values are constructed, the test data generator returns *control code*, which is a sequence of instructions that accesses the microprocessor resources associated with the instruction operands and brings them into the required states. For example, if an instruction operand is a register, control code writes the constructed value into that register. Following the concept of the *constraint-based random generation*, different calls of a test data generator may lead to different values of free variables. However, each generated set of values should cause the specified test situation.

D. Constraint Solver Engine

The *constraint solver engine* is a framework component that helps *test data generators* to construct *test data* by solving *constraints* specified in *test situations*. The engine is implemented as a collection of *solvers* encapsulated behind a generic interface. Solvers are divided into two major families: (1) *universal solvers* (handling a wide range of constraint types) and (2) *custom solvers* (aimed at specific test data generation tasks).

Universal solvers are built around external *SMT solvers* (like Yices [13] and Z3 [14])), which provide a rich constraint description language (supporting Boolean algebra, arithmetic, logic over fixed-size bit vectors and other theories) as well as effective decision procedures for solving such constraints. The MicroTESK framework uses Java Constraint Solver API [20] providing a generic interface to SMT-LIB-based constraint solvers [15]. The library allows dynamically creating constraints in Java, mapping them to the SMT-LIB descriptions, launching a solver and transferring results back to Java.

Some test situations are hardly expressible in terms of SMT constraints (e.g., situations in floating-point arithmetic, memory management, etc.). For such situations engineers are able to provide special custom solvers/generators. Note that custom solvers can also use SMT solvers to construct test data; though they usually implement non-trivial logic on forming a constraint system and interpreting its solution. When the design/coverage model is extended with a new type of knowledge, it often means a need to provide a corresponding custom solver. To facilitate extension of the constraint solver engine with new solvers, both universal and custom solvers implement uniform interfaces.

VII. Conclusion

We have suggested the extendable achitecture for test program generation framework. The proposed solution, named MicroTESK, can combine a wide range of microprocessor modeling and testing techniques. The central part of the framework is built around instruction-level models and random/combinatorial test program generators. More complicated types of models and test generation engines are supposed to be added as the framework's extensions. The goal of our work is not to create a "silver bullet" for microprocessor verification and testing (which, we believe, does not exist), but to organize a flexible, open-source environment being able to absorb a variety of useful approaches. Let us emphasize that the development having been launched at ISPRAS is based on the many-years experience of verifying industrial microprocessors. The work has not been finished, and there are a lot of things need to be done. In the nearest future, we are planning to implement the framework core and customize the generator for widely-spread microprocessor architectures, including ARM and MIPS. We are also working on MicroTESK's extensions for specifying/testing memory management mechanisms and pipeline control logic.

References

[1] M.S. Abadir, S. Dasgupta, *Guest Editors' Introduction: Microprocessor Test and Verification*. IEEE Design & Test of Computers, Volume 17, Issue 4, 2000, pp. 4–5.

[2] A. Kamkin. *Test Program Generation for Microprocessors*. Institute for System Programming of RAS, Volume 14, Part 2, 2008, pp. 23–63 (*in Russian*).

[3] http://www.arm.com/community/partners/display_product/rw/ProductId/5171/.

[4] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov and A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification*. IEEE Design & Test of Computers, Volume 21, Issue 2, 2004, pp. 84–93.

[5] P. Mishra and N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors*. ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 13, Issue 3, 2008, pp. 1–36.

[6] http://forge.ispras.ru/projects/microtesk.

[7] A. Kamkin. *Some Issues of Automation of Test Program Generation for Branch Units of Microprocessors*. Institute for System Programming of RAS, Volume 18, 2010, pp. 129–150 (*in Russian*).

[8] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus and G. Shurek. *Constraint-Based Random Stimuli Generation for Hardware Verification*. AI Magazine, Volume 28, Number 3, 2007, pp. 13–30.

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

[10] http://www.cs.cmu.edu/~modelcheck/smv.html.

[11] T.N. Dang, A. Roychoudhury, T. Mitra and P. Mishra. *Generating Test Programs to Cover Pipeline Interactions*. Design Automation Conference (DAC), 2009, pp. 142–147.

[12] A. Kamkin, E. Kornykhin and D. Vorobyev. s *Reconfigurable Model-Based Test Program Generator for Microprocessors*. Software Testing, Verification and Validation Workshops (ICSTW), 2011, pp. 47–54.

[13] B. Dutertre and L. Moura. *The YICES SMT Solver*. 2006 (http://yices.csl.sri.com/tool-paper.pdf).

[14] L. Moura and N. Bjørner. *Z3: An Efficient SMT Solver*. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337–340.

[15] D.R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial*. GrammaTech, Inc., Version 1.1, 2011.

[16] M. Aharoni, S. Asaf, L. Fournier, A. Koifman and R. Nagel. *FPgen – A Test Generation Framework for Datapath Floating-Point Verification*. High Level Design Validation and Test Workshop (HLDVT), 2003. pp. 17–22.

[17] M. Freericks, *The nML Machine Description Formalism*. Techical Report, TU Berlin, FB20, Bericht 1991/15.

[18] R. Moona, *Processor Models For Retargetable Tools*. International Workshop on Rapid Systems Prototyping (RSP), 2000, pp. 34–39.

[19] *MIPS64^{TM} Architecture For Programmers*. Volume II: The MIPS64^{TM} 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.