2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MicroTESK: Automated Architecture Validation Suite Generator for Microprocessors

Mikhail Chupilko, Alexander Kamkin, Alexander Protsenko, Sergey Smolov, Andrei Tatarnikov
Ivannikov Institute for System Programming of Russian Academy of Sciences, Moscow, Russia
{chupilko, kamkin, protsenko, smolov, andrewt}@ispras.ru

***Abstract*—MicroTESK is a tool that automates construction of test program generators for microprocessors. The main part of each generator is the core implementing architecture-independent test generation methods. To produce tests for a specific instruction set architecture, the tool analyzes formal specifications of that architecture and extracts all necessary information, including, first of all, the instructions' syntax and semantics. The primary use case of the tool is to generate test programs from high-level test templates, or scenarios, provided by verification engineers. In this paper, we present a new facility, namely automatic generation of architecture validation suites, which allows creating simple tests in a "push-button" manner. Such tests exercise individual instructions and short sequences of dependent instructions by applying both random and directed values of the operands. MicroTESK and the underlying approach have been applied to the ARMv8, MIPS64, PowerPC, RISC-V, and x86 architectures.**

Keywords—microprocessors, instruction set architectures, formal specifications, functional verification, model-based testing, test program generation, architecture validation suites

I. INTRODUCTION

Test program generation and analysis of test program execution traces is the most widely used approach to *functional verification* of microprocessors. To generate test programs, special tools called *test program generators* (TPGs) are used. They implement a variety of test generation methods to exercise behavior of a microprocessor in "all possible" situations. An important requirement for modern TPGs is support of a wide range of *instruction set architectures* (ISAs). This implies that information on the ISA must be separated from the implementation of the test generation methods, i.e. the information makes a model of the ISA. Such approach is usually referred to as *model-based testing*.

Industrial TPGs such as Genesys-Pro [1] and RAVEN (Random Architecture Verification Engine) [2] follow the model-based approach. The main idea is that a TPG consists of the *core*, which implements architecture-independent test generation methods, and the *model*, which stores all information required to create test programs for the corresponding ISA. Assuming that the ISA model is specified, the primary use case is as follows: a verification engineer describes a *test template* in a special high-level language; the TPG analyzes the test template and constructs randomized assembly code that fits the description. The goal of our work is to bring further automation to the process to facilitate it. Having an ISA model available, some basic tests can be produced automatically in a "push-button" manner.

In this paper, we present MicroTESK (Microprocessor TEsting and Specification Kit), an open-source tool for constructing model-based TPGs [3]. It provides the reusable core and constructs ISA models by processing the *formal specifications*. To represent ISA specifications, the tool utilizes nML, a simple architecture description language [4]. Test templates are written in a Ruby-based language [5] (a Python extension is under development). The use of the well-tried languages is one of the distinctive features of the tool. Another one is a possibility to automatically generate basic test templates, so-called *architecture validation suites* (AVSs) [6]. This facility is especially useful at early stages of the microprocessor design process.

The rest of the paper is organized as follows. Section II reviews the related work addressing test program generation for microprocessors in general. Section III overviews the MicroTESK tool, describes tests being generated automatically, and considers how a MicroTESK-based TPG is developed and debugged. Section IV contains information on the supported architectures and a case study on applying the AVS generator in industrial settings. Section V concludes the paper and outlines the directions of future research.

II. RELATED WORK

Over the last decades, a lot of efforts have been invested into development of TPGs. The primary challenge is how to maximize the productivity of testing. This means to create AVSs that provide the necessary level of coverage with minimal effort. To address this challenge, the model-based approach was proposed [1]. It implies separation of a TPG into two main parts: (1) the model that represents a formal specification of the ISA under test and (2) the core that incorporates ISA-independent components implementing test generation methods. AVSs for such TPGs consist of test templates that formulate the properties of test programs to be generated in terms of the corresponding model. Test templates are processed with the core to produce tests satisfying the formulated properties. Such an approach allows reusing a TPG for multiple ISAs and generating large volumes of randomized tests sharing common properties on the basis of a single test template.

The best-known industrial TPGs that follow the model-based approach are Genesys-Pro [1] by IBM Research and RAVEN [2] by Obsidian Software (now owned by ARM). These TPGs are capable of generating random and constraint-based tests. They use ISA specifications (models) and test templates described in *domain-specific languages* (DSL) with some parts developed in C/C++. Both TPGs are proprietary and their vendors do not reveal details on how ISA specifications and test templates are developed. From publicly available information, it can be concluded that the models they use consist of three main parts: (1) description of instruction signatures; (2) information about test situations; (3) reference *instruction set simulator* (ISS). The main drawback is that these three parts are developed separately. Moreover, they are not monolithic (in sense of programming paradigm usage) and can include descriptions in multiple formats. This complicates their development and maintenance. Another issue is that despite the fact that the TPG core is ISA-independent and that AVSs for different ISAs solve common verification tasks, no facilities for AVS reuse are provided.

Besides TPGs there are a number of tools that use ISA specifications to support multiple platforms. Among them, there are ISSs and compilers (code generator back-ends). In general, they use C/C++ and DSL to describe the target platform configuration.

QEMU [7] is an ISS that supports multiple platforms. The supported ISAs are mapped to internal QEMU micro-instructions using code in C. Microprocessor state for different configurations (particular designs) and additional logic (e.g., memory management) are implemented as separate C libraries.

GNU Compiler Collection (GCC) [8] is a well-known compiler project with multiple ISA support. GCC machine descriptions consist of two parts: (1) patterns for instructions supported by the target machine and (2) headers defining macros that convey information about the target machine. Code is generated from a parse tree by applying the corresponding instruction patterns.

LLVM [9] is yet another retargetable compiler. Basically, a machine description in LLVM is a set of C++ classes generated from descriptions in the TableGen DSL. Registers and instruction templates are defined in the DSL, while coarse-grained compiler interaction has to be manually implemented in the corresponding classes.

In [10], experimental tools developed at Cadence are introduced. These tools construct ISSs, assemblers, and disassemblers based on ISA specifications in nML [4]. This is a DSL designed for describing the syntax and semantics of microprocessor instructions. Specifications in nML can be relatively easy created on the basis of ISA reference manuals.

It can be summarized that each tool with multiple platform support uses its own description format that suits its specific needs. In some cases, multiple formats are used for different parts of the description. The use of a single self-contained format for describing all aspects of the ISA would significantly simplify development and maintenance of TPGs. Among the above-considered approaches, the nML DSL is the most suitable description format. Specifications in nML can be derived from ISA reference manuals and they provide enough information to automatically generate AVSs.

III. MICROTESK APPROACH

MicroTESK is divided into two main parts: (1) the modeling framework that processes formal specifications and constructs a microprocessor model; (2) the testing framework that generates test programs on the basis of the model and test templates provided by users, i.e. verification engineers. The architecture of MicroTESK is shown in Figure 1.

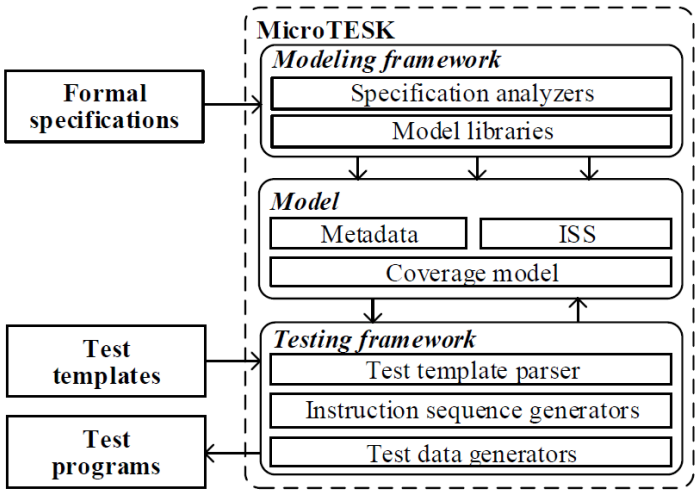

Fig. 1. The MicroTESK architecture

There are three main components of the model: (1) the *metadata* providing a catalogue of supported instructions; (2) the *instruction set simulator* (ISS) serving as a reference model; (3) the *coverage model* holding constraints describing execution paths of individual instructions.

The modeling framework conducts analyzing of formal specifications, extracting the necessary information, and construction of the model. To develop ISA specifications, one does it manually by means of the nML language [4]. The specifications describe data types, registers, memory, addressing modes, and instructions. Here is an nML specification of MIPS's ADD instruction.

```
op add(rd: R, rs: R, rt: R)
  syntax = format("add %s, %s, %s", rd.syntax, rs.syntax, rt.syntax)
  image  = format("000000%5s%5s%5s00000100000", rs.image, rt.image, rd.image)
  action = {
    if sign_extend(WORD, rs<31>) != rs<63..32> ||
       sign_extend(WORD, rt<31>) != rt<63..32> then
      unpredicted; // Precondition
    endif;
    temp33 = rs<31>::rs<31..0> + rt<31>::rt<31..0>;
    if temp33<32> != temp33<31> then
      exception("IntegerOverflow"); // Coverage item 1
    else
      mark("Normal"); // Coverage item 2
      rd = sign_extend(DWORD, temp33<31..0>);
    endif;
  }
```

Each instruction from ISA looks like an object with the following attributes: *syntax* (assembler format of the instruction), *image* (binary encoding), and *action* (what the instruction should do). To describe functionality of the instruction, the nML language supports bit-level commands (<a..b>, a::b, etc.), branching, and calling of other operations, including system ones (like *unpredicted* and *exception* in the above example).

Being rather simple although powerful enough in most cases, nML does not always have adequate facilities to describe complex *memory management units* (MMUs). For this purpose, a special mmuSL language extension is used. MMU specifications include address types, memory segments, buffers, tables, and overall control logic for handling loads and stores [11]. In the following example, an address type named VA with the only one attribute

(the virtual address itself) is declared. It should be notices that in general case the address type structure may contain a number of attributes.

```
address VA(
  value: 32 // Virtual Address itself
)
```

A memory segment represents a mapping from a set of addresses of some type to a set of addresses of another type. An example given below defines a segment SEG that maps a VA of the given set (**range**) to the physical address (PA). If PA is defined in the same way as VA, the segment performing flat translation with no use of TLBs and tables (**read**) would look as follows.

```
segment SEG (va: VA) = (pa : PA)
  range = (0x00000000, 0xffffffff)
  read = {
    pa.value = va.value;
  }
```

Buffers (TLBs, cache units, page tables, etc.) are specified with the following parameters: the associativity (ways), the number of sets (sets), the entry format (**entry**), the index calculation function (index), the tag calculation function (tag) and the data eviction policy (policy). Here comes a sample description of TLB accessed by VAs. The only attribute given here (**entry**) sets up the TLB record format, the others are omitted. The keyword **register** means that the buffer is mapped to the registers and can be read in the nML specification.

```
register buffer TLB (pa: PA)
  entry = (value : 32)
  ...
```

Specification of memory access instructions processing uses requesting of the segments and buffers. The syntax of mmuSL is similar to nML though allows using such constructs as follows.

- B(A).**hit** – the buffer B contains an entry for the address A,
- E = B(A) – the entry for the address A is read from the buffer B and assigned to E,
- B(A) = E – the entry E for the address A is written to the buffer B,
- and the like.

Here is a fragment of the MIPS MMU specification. It contains two attributes, **read** and **write**, which, respectively, define logic of loads and stores. This MMU is called each time when an access to memory happens in nML, triggering the whole of the MMU logic (address translation, caching, etc.) as shown below.

```
mmu MMU(va: VA) = (data: 32)
  var c: 3; // Cache policy
  var pa: PA;
  var cacheData: 256;
  var offset: 3;
  var l1Entry: L1.entry;
  read = {
    c = 3; // Default cache policy
    pa.value = va.value; // Let physical address be equal to logical address
    MMU_PA(pa) = pa.value; // PA is saved into nML register
    offset = pa.value<4..2>; // address is WORD-aligned

    if c<1..0> != 2 then // If address is cacheable
      if L1(pa).hit then // L1 Cache Access
        l1Entry = L1(pa);
        cacheData = l1Entry.DATA;
      else
        cacheData = M(pa); // Memory Access
        l1Entry.V = 1; // L1 Cache Update
        l1Entry.TAG = pa.value<31..12>;
        l1Entry.DATA = cacheData;
        L1(pa) = l1Entry;
      endif; // If the address hits the L1.
    else     // The address is uncacheable.
```

```
        cacheData = M(pa); // Memory Access
      endif; // If the address is cacheable.
      data = cacheData<...>; // Data Extraction
    }
    write = { ... }
  }
```

To make specification of ISA work for the purpose of test program generation, verification engineer should develop a set of Ruby-based [5] test templates. The test templates description language combines facilities for describing properties of test programs and features of a general-purpose programming language. ISA-specific constructs such as instruction wrappers are created on the fly by using the model metadata. Broadly speaking, test templates specify how to combine instruction sequences and what constraints to apply. For example, the Ruby code below describes all possible pairs of ADD and SUB instructions with “Normal” and “IntegerOverflow" constraints having been attached.

```
class ExampleTemplate < BaseTemplate
  def run
    block(:combinator => 'product') {
      iterate {
        add t0, t1, t2 do situation('Normal') end
        add t0, t1, t2 do situation('IntegerOverflow') end
      }
      iterate {
        sub t3, t4, t5 do situation('Normal') end
        sub t3, t4, t5 do situation('IntegerOverflow') end
      }
    }.run
  end
end
```

The generation process, conducted by the reusable MicroTESK TPG core adjusted by a test template, consists of the following stages:

1. constructing an abstract instruction sequence (no particular data);
2. solving constraints applied to the instructions and generating data;
3. creating initialization code that prepares the registers and the memory;
4. executing the instructions (including the initialization code) in the ISS;
5. creating self-checks based on the information provided by the ISS (optional);
6. printing the resulting instructions to an assembly file.

MicroTESK allows constructing complex instruction sequences by combining smaller parts. To solve constraints, the tool utilizes a number of built-in and external SAT- and SMT-solvers. Supported types of constraints include: (1) constraints on instruction operands; (2) constraints related to control flow; (3) floating-point constraints; (4) MMU-related constraints. The tool architecture facilitates integration of custom components for sequence processing and constraint solving (test data generation).

Each base template for the given ISA has an exception handlers section. Inside the section, the handlers are separated by means of memory replacement commands orgs; each branch has a mark (“Overflow”, “Unaligned”, etc.) to be called in user templates to make a specific situation. As for the interruption handling, it is up to the nML part to make the processing of events carefully. Each event processing is described in the correspondent nML operation (including reset, process context switching, etc.) and should be called explicitly from the test template even if it is caused by changing of input wires.

The template-based approach implemented in MicroTESK is rather popular as it automates many routine tasks related to construction of instruction sequences and test data. However, more automation would be desirable. Being in many respects similar to each other, microprocessors are often verified in a similar way. In other words, there are typical test scenarios that can be applied to different microprocessors. Our idea is to generate such kind of tests, namely AVS, automatically by analyzing ISA specifications.

We have implemented a MicroTESK extension that automatically produces the following types of tests:

- tests for individual instructions (execution units):
  - tests that for each instruction
    - randomize the operands;
    - try the boundary values;
    - cover all possible execution paths extracted from the specifications;
- tests for instruction sequences (pipeline control logic):
  - tests that enumerate short instruction sequences (pairs, triples, etc.),
  - and, optionally,
    - enumerate the execution paths of the instructions;
    - enumerate the dependencies between the instructions.

Here is an automatically constructed test template aimed at covering all execution paths of the `ADD` instruction.

```
sequence {
  add r(_), r(_), r(_) do testdata('all-paths') end
}.run
```

The test template is handled as follows. MicroTESK extracts all execution paths, including: (1) an integer overflow path; (2) a normal execution path. For each of them, it formulates the constraint and generates test data by solving that constraint (to do it, the tool uses external SMT solvers). As a result, it constructs test cases to cover each execution paths individually. Here is how an integer overflow test case may look like.

```
// Initialization code
lui r2, 0x8000
lui r3, 0x8000

// Target instruction
add r1, r2, r3 // IntegerOverflow exception
```

An automatically constructed test template aimed at testing multiple instructions has the following structure.

```
block(:combinator => '...', :compositor => '...', :permutator => '...') {
  # Group 1 (e.g., integer arithmetic)
  iterate {
    add r(_), r(_), r(_)
    ...
  }
  ...
  # Group N (e.g., loading / storing)
  iterate {
    sw r(_), _, r(_)
    ...
  }
}
```

IV. Case Study

MicroTESK has been used to construct TPGs for several ISAs, including ARMv8 (AArch64), MIPS64 (Revision 5), PowerPC (e500mc), RISC-V (Version 2.2), and x86 (x86-64). The TPGs for ARMv8, MIPS64, and RISC-V are industrial tools, while the rest are research prototypes. Table I provides information on the ISA specifications used for TPG construction and the efforts spent on their development.

The TPG development is reduced to specifying the corresponding ISA. The labor costs are approximately 2-5 instructions per person-day (depending on the ISA complexity). It should be noted that specifications can be reused when describing other designs of the same family. The nML and mmuSL languages allow marking specification elements with revisions to enable/disable those elements depending on the ISA version.

Table I. Information on ISA specifications

Instruction Set Architecture	ARMv8	MIPS64	PowerPC	RISC-V	x86 (x86-16)
Instruction count	1015	235	122	262	58
ISA specification size (LOC)	18178	3999	2766	3500	2585
MMU specification size (LOC)	2643	267	–	–	–
Efforts (person-months)	30	4	3	4	2.5

It is not a secret that each program contains bugs. After their development, the nML specifications may also have errors, such as incorrect assembler format of instructions and wrong results of instruction execution on the MicroTESK simulator. To detect such errors, we supply all the developed MicroTESK-based TPGs with *environments*. The environment includes the *reference model* of the ISA and the *comparator*. The comparator is the tool that compares results of test program simulation on the TPG and on the reference model. We use execution traces as simulation results. An execution trace contains events of the following kinds: instruction execution, write to register, write to memory, read from memory.

For every test template of the TPG the following actions are made:

1) The test program is generated. Upon generation the program is simulated on the MicroTESK internal simulator; the execution trace is generated.

2) The test program is compiled by the *toolchain* and is simulated on the reference model; another execution trace is generated. The QEMU emulator is used here. We have modified the QEMU output trace format to be compatible with the MicroTESK related one.

3) The execution traces from steps 1 and 2 are compared. We have implemented the Trace Matcher [12] tool as a comparator. The tool produces a report of discrepancies that were found during test program simulation.

A prototype of MicroTESK's AVS generator has been used to verify a MIPS-compatible microprocessor. The microprocessor implements 221 instructions, which can be divided into 13 groups: (1) integer arithmetic; (2) logical operations; (3) data transfer; (4) shifts; (5) branching; (6) empty operations; (7) loading and storing; (8) exceptions; (9) system operations; (10) floating-point arithmetic; (11) FPU data transfer; (12) type conversion; (13) FPU branching.

It is worth noting that the verification was carried out at a late design stage, when the microprocessor had been thoroughly tested and had been ready for tapeout. The generator constructed test programs by enumerating triples of instructions, functional branches for each of the instructions (integer overflow/normal execution, cache hit/miss, etc.), and dependencies between the instructions. Since the number of instructions is large enough, we used some heuristics to reduce the test size. The generated AVS contains approximately 37.5 million test cases, each being composed of 20-100 instructions (target instructions and initialization code). The total number of instructions in the generated test programs is more than 1 billion. As a result, 6 errors were found in the reference instruction set simulator and 9 errors were found in the RTL model. Here are two examples.

Failure 1. The first instruction divides the value of the register *r*1 by the value of the register *r*2 and saves the quotient and the remainder in the registers *LO* and *HI* respectively. The second instruction moves the contents of the register *r*3 into the register *HI*. The third instruction moves the contents of the register *HI* into the register *r*4. The failure is that *r*4 contains the result of the division instead of the contents written by the second instruction.

```
div  r1, r2 // LO := r1 / r2, HI := r1 % r2
mthi r3     // HI := r3
mfhi r4     // r4 := HI
```

Failure 2. The first instruction moves the contents of the register *r*1 into the FPU condition codes register (FCCR, $25). The second instruction moves the contents of the register *r*3 into the register *r*2 unless the condition code $*fcc*0 is false. The third instruction performs a floating-point operation that causes the *Inexact* exception. The failure occurs when the superscalar mode is enabled (in this mode, integer and floating-point instructions are

executed simultaneously) and the *Enables.Inexact* bit is set in the FPU control/status register (FCSR), and it manifests as a "hang" of the microprocessor.

```
ctc1  r1, $25       // FCCR := r1
movt  r2, r3, $fcc0 // if $fcc = 1 then r2 := r3
cvt.s $f1, $f2      // Inexact exception
```

V. Conclusions

The most widely used approach to functional verification of microprocessors is test program generation. The idea is to construct a huge volume of randomized assembly code that covers various situations in microprocessor behavior. There are tools, so-called TPGs, which automate that process. They represent knowledge of a target ISA (including the instructions' syntax and semantics) and are able to produce test programs based on high-level test templates. An obvious observation is that all microprocessors (at least, most of them) are similar to each other; thus, there should be general principles of test template development for different microprocessor architectures. The main goal of our work is to reveal such principles and to automate construction of typical tests based on ISA specifications.

We have extended the MicroTESK tool by the facility for generating basic test templates, so-called AVSs. That extension allows testing how the microprocessor executes individual instructions and short sequences of instructions. On the one hand, the approach reduces time to get first tests; thus, it may be useful at early design stages. On the other hand, it provides a certain degree of systematicness and thoroughness; thus, it may benefit at all stages. Our experience has shown that automatically generated tests can discover critical bugs even in mature designs. The MicroTESK tool, including the suggested extension, is open-source and distributed under the Apache License, Version 2.0.

In the nearest future, we are planning to reveal more industrial test design patterns to make our tool mature enough to bring microprocessor verification to a new level of automation. Another important direction is automating development of online TPGs, i.e. generators that synthesize and launch tests on the fly. Such tools are essential for post-silicon verification and for testing FPGA prototypes.

Acknowledgement

The authors would like to thank Russian Foundation for Basic Research (RFBR). The reported study was supported by RFBR, research project No18-07-01218.

References

[1] 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 & Test of Computers, 21(2), 2004. pp. 84--93.

[2] Random Architecture Verification Engine (RAVEN) — http://www.slideshare.net/DVClub/introducing-obsidian-software-andravengcs-for-powerpc

[3] Microprocessor Testing and Specification Kit (MicroTESK) — http://forge.ispras.ru/projects/microtesk

[4] M. Freericks, "The nML machine description formalism", Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.

[5] A. Tatarnikov, "Language for describing templates for test program generation for microprocessors" , Proceedings of ISP RAS, 28(4), 2016. pp. 81-102.

[6] L. Fournier, A. Koyfman, and M. Levinger. "Developing an Architecture Validation Suite — Application to the PowerPC Architecture", Design Automation Conference (DAC), 1999. pp. 189–194.

[7] Quick Emulator (QEMU) — http://www.qemu.org/

[8] GNU compiler collection (GCC) — https://gcc.gnu.org/

[9] Low Level Virtual Machine (LLVM) — https://llvm.org/

[10] M. Hartoog, J. Rowson, P. Reddy, S. Desai, D. Dunlop, E. Harcourt, N. Khullar. "Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign", Design Automation Conference (DAC), 1997. pp. 303–306.

[11] M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, A. Tatarnikov, "Specification-based test program generation for ARM VMSAv8-64 memory management units", Workshop on Microprocessor Test and Verification, 2015. pp.1-6.

[12] Trace Matcher tool — https://forge.ispras.ru/projects/traceutils