Language for Describing Templates for Test Program Generation for Microprocessors

A.D. Tatarnikov <andrewt@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Test program generation and simulation is the most widely used approach to functional verification of microprocessors. High complexity of modern hardware designs creates a demand for automated tools that are able to generate test programs covering non-trivial situations in microprocessor functioning. The majority of such tools use test program templates that describe scenarios to be covered in an abstract way. This provides verification engineers with a flexible way to describe a wide range of test generation tasks with minimum effort. Test program templates are developed in special domain-specific languages. These languages must fulfill the following requirements: (1) be simple enough to be used by verification engineers with no sufficient programming skills; (2) be applicable to various microprocessor architectures and (3) be easy to extend with facilities for describing new types of test generation tasks. The present work discusses the test program template description language used in the reconfigurable and extensible test program generation framework MicroTESK being developed at ISP RAS. It is a flexible Ruby-based domain-specific language that allows describing a wide range of test generation tasks in terms of hardware abstractions. The tool and the language have been applied in industrial projects dedicated to verification of MIPS and ARM microprocessors.

Keywords: microprocessors; functional verification; test program generation; test templates; domain-specific languages.

DOI: 10.15514/ISPRAS-2016-28(4)-5

For citation: Tatarnikov A.D. Language for Describing Templates for Test Program Generation for Microprocessors. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 4, 2016. pp. 77-98. DOI: 10.15514/ISPRAS-2016-28(4)-5

1. Introduction

Functional verification is acknowledged to be the bottleneck in microprocessor design cycle. According to various estimates, it accounts for more than 70% of overall project time and resources. In the current industrial practice, function verification mainly relies on *test program generation (TPG)* which is done by special automation tools [1]. Generated *test programs (TP)* are instruction sequences aimed to trigger

certain events in the microprocessor design under verification. TPG tools are aimed to provide a high level of test coverage by applying a rich set of generation methods. As modern microprocessors are getting more and more complex, new more advanced methods emerge. A common problem for TPG tool developers is how to overcome the complexity and make it easy to apply the growing set of methods to a wide range of microprocessor designs.

One of possible ways to increase the flexibility of a TPG tool is to separate generation logic from descriptions of test cases. This method is known as *template-based* generation. The key idea of the method is that test programs are generated on the basis of abstract descriptions called test program templates or *test templates (TTs)*. The method helps generate high-quality tests directed towards specific situations or classes of situations. TTs specify methods to be used for constructing instruction sequences and constraints on instruction operand values which must be satisfied to make certain events to fire. Test data are generated by finding random solutions to the given constraint systems. Such approach is usually referred to as *constraint-based* random generation [2]).

The *template-based* approach is implemented in a number of TPG tools including MicroTESK [3], a reconfigurable [4] and extensible [5] TPG framework being developed at ISP RAS. The framework uses *formal specifications* to construct TPG tools for specific microprocessor designs. A constructed TPG tool is separated into two main components: (1) an architecture-independent test generation core and (2) an architecture specification, or a model. The approach called *model-based* [1] helps significantly reduce the efforts to support a new microprocessor architecture by reusing the core. The core is designed as a set of generation engines which can be easily extended with plugins implementing new TPG methods. Test programs are generated by processing TTs that describe verification tasks in terms of the model and the generation methods implemented by the core.

This paper describes the test template description language (TTDL) used in MicroTESK. This is a domain-specific language implemented as a set of Ruby [6] libraries, which is easy adaptable to changing configurations. Facilities for describing instruction calls for a specific ISA are dynamically added and are based on information provided by the model. Also, the MicroTESK TTDL provides a rich set of facilities for describing verification tasks which are common for all microprocessor configurations. When MicroTESK is extended with new TPG methods, support for these features is added in the TTDL by providing new Ruby libraries.

The rest of the paper is divided into five sections. Section 2 contains a brief survey of the existing TPG tools that follow the template-based approach. Section 3 formulates the requirements for a TTDL imposed by MicroTESK that led to creating the described TTDL. Section 4 provides a detailed description of the architecture and facilities of the MicroTESK TTDL. Section 5 contains a case study of applying the TTDL for describing test cases in industrial projects. Section 6 discusses the results and outlines directions of future research and development.

2. Related work

Functional verification has always been a major issue for the research community. Over the last decades, a lot of TPG methods and tools have emerged. The template-based approach described in this paper has been applied in a number of tools developed by different teams. This section gives an overview of the most significant of existing TPG tools and discusses strong and weak points of their TTDLs.

IBM Research has been one of the major contributors in the field of TPG for microprocessors during the last decades. Genesys-Pro [1], one of their most recent tools, uses TTs to describe TPG tasks as constraint satisfaction problems (CSP) [2] and generates test data by solving these CSPs. Constraints can be used to specify such aspects of functionality as boundary conditions, exceptions, cache hits/misses, etc. The TTDL used by Genesys-Pro is a completely impendent domain-specific language which provides a rich set of features. The language features it offers can be divided into four groups: (1) basic instruction statements, (2) sequencing-control statements, (3) standard programming constructs, and (4) constraint statements. By combining these constructs, users can compose complex TTs with a degree of randomness varied from completely random to completely directed. The main advantage of the language is that it is designed for describing test scenarios and it does not confuse verification engineers with any unnecessary programming constructs. At the same time, being not based on existing languages, it does not take advantage of well-tried constructs that can help organize TTs into reusable libraries. This can be important as industrial testbenches usually contain thousands lines of code. Also, it is unclear how easy the language can be extended with new constructs for describing new types of TPG tasks.

Another company that has made a significant contribution in development of TPG tools is Obsidian Software (now acquired by ARM) [7]. Their tool RAVEN (Random Architecture Verification Engine) [8] generates random and directed tests based on TTs. Test templates are focused on coverage grids and use constraints to formulate specific coverage goals. There is no detailed information available on this technology. It is known that TTs can be either generated by the tool's GUI or created as text. The language must suit well for the TPG tasks that can be accomplished with RAVEN. However, the question whether it is suitable for more general tasks stays open.

Also, Samsung Electronics created a TPG framework called RDG (Random Diagnostics Generator) [9] for testing reconfigurable processors. It uses TTs created in the C++ language to specify instructions that will be used in a TP and constraints on their input values that should be satisfied in order to meet testing goals. This approach takes advantage of power and performance of C++, but requires solid programming skills which are not common for verification engineers.

Finally, MicroTESK [3] version 1.0 used TTs written using Java libraries [10]. This is not convenient as verification engineers are forced to deal with Java abstractions such as classes and interfaces, which are not related to verification tasks. Moreover, details of language implementation must be hidden from users in order to be able to change it without breaking existing TTs. This motivated to create a new domain-specific language for the new version of MicroTESK.

3. Requirements for TTDL

Requirements for a TTDL can be divided in two groups: (1) general requirements for a TTDL; (2) requirements related to integration into the MicroTESK framework. Let us first consider the general requirements that are common for all TTDLs. A TTDL used to describe scenarios for random and directed tests must provide facilities:

1) to describe instructions calls and data definitions using syntax similar to the one used in assembly code;
2) to manage memory allocations in the same way as in the assembly language;
3) to fill memory with data generated according to specific rules;
4) to compose instruction sequences using a wide range of methods (random, combinatorial, etc.) and to merge these sequences;
5) to specify random values and the degree of their randomness described by distributions;
6) to select instructions at random with the specified degree of randomness;
7) to specify constraints on instruction arguments;
8) to describe initialization code that places generated test data to proper registers or memory addresses;
9) to specify code of self-checks that check validity of the resulting state of the microprocessor;
10) to describe exception handlers;
11) to specify conditions for generating different code depending on the context;
12) to insert comments and custom text into generated TPs;
13) to reuse existing TTs and their parts;
14) to split generated TPs into multiple files.

This list is not complete, but it is enough to conclude that the TTDL must be a domain-specific language that provides constructs for the listed facilities.

Another important consideration is that it must be integrated into MicroTESK. First of all, MicroTESK is written in Java and its generation engines operate with Java objects. Therefore, the result of TT processing must be a hierarchy of Java objects that then will be passed to TPG engines. The front-end of a TTDL processor can be implemented using two approaches: (1) creating a Java-based parser for the new language or (2) reusing an existing Java-based parser for one of the popular programming languages. A crucial requirement for the second approach is that the language must be easy to extend with new domain-specific constructs.

Now let us consider the requirements imposed by reconfigurability and extensibility of MicroTESK:

1) *Reconfigurability* means that it can be applied to microprocessors with different ISAs. Consequently, facilities used to describe instruction calls must be changeable. Ideally, they must be added dynamically depending on

the information provided in the model that describes the configuration of the design under verification.

2) *Extensibility* means that the set of supported TPG methods can be extended by adding plugins implementing new methods. Often it will require adding new constructs in the TTDL. Thus, it must be possible to dynamically add language constructs depending on the installed plugins.

In other words, a crucial requirement for the MicroTESK TTDL is the ability to dynamically change the set of supported language constructs. Obviously, changes in the tool configuration must not involve modification of the TTDL processor. Creating a flexible language processor from scratch is a challenging task. A simpler solution would be to reuse a parser of an existing language.

Having considered several possible alternatives, it was decided to use JRuby [11], a Java-based implementation of the Ruby language, as a front-end of the TTDL processor. Ruby was selected because of its support for *metaprogramming* [12], which allows adding new language features at runtime. Thus, the created TTDL combines basic programming constructs provided by the Ruby core with constructs for describing TTs provided by MicroTESK. The TTDL front-end is implemented as a set of Ruby libraries that define language facilities for the above mentioned requirements. Facilities that depend on the current configuration are dynamically added using metaprogramming.

It is also worth mentioning that scripting languages like Ruby are quite popular among verification engineers, who often use them to create in-house test generators. So, another advantage of using Ruby is that it can make the TTDL easier to learn.

4. TTDL Description

4.1 Language Processor Architecture

The job of the TTDL processor is to build a hierarchy of Java objects describing a TT and to pass it to the MicroTESK generation engines for further processing. The TTDL processor is divided into a *Ruby-based front-end* and *Java-based back-end*. The back-end is implemented as set of factories for creating Java objects that correspond to specific entities of a TT. The front-end is represented by Ruby libraries that provide language constructs for describing these entities and perform interaction with the back-end to build corresponding Java objects. In other words, a language feature is defined by a Ruby module that specifies its syntax and a Java module that describes corresponding entities and provides means of constructing them. New language features can be supported by providing corresponding modules.

The TTDL contains features that are configuration dependent. This includes facilities for describing instruction calls, which are determined by the model built by MicroTESK from ISA specifications. These language features are managed by a special Ruby module that uses metaprogramming to define corresponding constructs at runtime based on the information provided by the model.

4.2 Test Template Structure

A TT is a program in Ruby executed by MicroTESK with the help of JRuby to build Java objects that formulate tasks for the TPG engines implemented by the tool core. More technically, it is a subclass of the *Template* base class provided by the MicroTESK library. All domain-specific language constructs are implemented as methods of this class. The *Template* class is not monolithic, it unites a set of Ruby modules responsible for various features into a single class. Language extensions are also implemented as modules to be included in the base class. Configuration-specific methods are dynamically defined when the class is loaded.

The listing below shows the structure of a TT class:

```
require ENV['TEMPLATE']
class MyTemplate < Template
  def initialize
    super
    # Initialize settings here
  end

  def pre
    # Place your initialization code here
  end

  def post
    # Place your finalization code here
  end

  def run
    # Place your testing task description here
  end
end
```

The first line imports the *Template* base class from the location specified by the *TEMPLATE* environment variable. The exact location depends on the configuration and is determined automatically.

Classes describing TTs define four methods:

- *initialize* - configures TT settings if there is a need to override the default;
- *pre* - defines ISA-specific constructs and specifies initialization code to be inserted in the beginning of TPs;
- *post* - specifies finalization code to be inserted in the end of TPs;
- *run* - contains descriptions of test cases to be generated.

The methods will be filled with constructs described further.

4.3 Managing Memory Allocation

It may be required to place code and data sections of generated TPs at specific memory locations. The assembly language provides special directives to accomplish this task. The TTDL offers similar constructs. An important note is that MicroTESK

simulates TPs in the process of their generation. Consequently, these constructs not only specify directives to be placed into TPs, but also manage memory allocation in the simulator.

The TTDL provides the following methods for managing addresses, which are applicable to both code and data sections:

- *align* - aligns the allocation address by the amount n passed as an argument, which by default means 2n bytes.
- *org* - sets the allocation origin, which is required to increase the allocation address. It is possible to set an *absolute* or *relative* origin. The former can be specified as org n and means an offset by n bytes from the base virtual address. The latter can be specified as *org :delta=>n* and means an offset by n bytes from the most recent allocation address.
- *label* - associates the specified label with the current address.

The listed methods rely on the following TT settings:

- *align_format* - specifies textual format for the align directive;
- *org_format* - specifies textual format for the org directive;
- *base_virtual_address* - specifies the base virtual address for memory allocation;
- *base_physical_address* - specifies the base physical address for memory allocation;
- *alignment_in_bytes* - specifies how the alignment amount should be interpreted.

The first four settings are initialized with default values in the *initialize* method of the *Template* base class as shown below and can be changed in the current TT class:

```
@org_format = ".org 0x%x"
@align_format = ".align %d"
@base_virtual_address = 0x0
@base_physical_address = 0x0
```

The last setting is implemented as a method that can be overridden to change its behavior:

```
def alignment_in_bytes(n) 2 ** n end
```

4.4 Defining Random Distributions

Many TPG tasks involve selection based on random distribution. The TTDL provides the following methods to define random distributions:

- *range* - creates an object describing a range of values and its weight, which are specified by the *value* and *bias* attributes. Values can be one of the following types:
  - *single* value;

- *range* of values;
- *array* of values;
- *distribution* of values.

The *bias* attribute can be skipped which means default weight. Default weights are used to specify an even distribution based on ranges with equal weights.

- *dist* - creates an object describing a random distribution from a collection of ranges.

The code below illustrates how to create weighted distributions for integer numbers:

```
simple_dist = dist(
  range(:value => 0, :bias => 25),          # Value
  range(:value => 1..2, :bias => 25),       # Range
  range(:value => [3, 5, 7], :bias => 50)  # Array
  )
composite_dist = dist(
  range(:value=> simple_dist, :bias => 80), # Distribution
  range(:value=> [4, 6, 8], :bias => 20)       # Array
  )
```

4.5 Describing Data Definitions

Data definitions are based on assembler-specific directives, which are not described by the microprocessor model and, therefore, must be configured in TTs. The configuration information includes textual format of the directives and mappings between data types used by the assembler and the microprocessor model. Data directives are configured using the *data_config* construct, which must be placed in the pre method. Here is an example:

```
data_config(:text=>".data", :target=>"MEM") {
  define_type :id=>:byte, :text=>".byte", :type=>card(8)
  define_type :id=>:half, :text=>".half", :type=>card(16)
  define_type :id=>:word, :text=>".word", :type=>card(32)
  define_space :id=>:space, :text=>".space", :fillWith=>0
  define_ascii :id=>:ascii, :text=>".ascii", :zero=>false
  define_ascii :id=>:asciiz, :text=>".asciiz", :zero=>true
}
```

The *data_config* method has the following parameters:

- *text* - specifies the textual format of a directive that marks the beginning of a data section;
- *target* - specifies the memory array defined in the model to which data will be placed during simulation;
- *base_virtual_address* (optional, *0* by default) - specifies the base virtual address for data sections.

Distinct data directives are configured using special methods that must be called inside the *data_config* block. All of these methods share two common parameters: *id* and *text*. The first specifies the keyword to be used in a TT to address the directive and the second specifies how it will be printed into the TP. Here is the list of methods:

- *define_type* - defines a directive to allocate memory for a data element of the data type specified by the *type* parameter;
- *define_space* - defines a directive to allocate memory filled with a default value specified by the *fillWith* parameter;
- *define_ascii_string* - defines a directive to allocate memory for an ASCII string terminated or not terminated with zero depending on the *zero* parameter.

The above example defines directives *byte*, *half*, *word*, *ascii* (non-zero terminated string) and *asciiz* (zero terminated string) that place data in the memory array *MEM* defined in the microprocessor model.

Once data directives have been configured, data sections can be defined using the *data* construct. Data definitions can be of two kinds depending on the context:

1) *Global data* that are available to all test cases generated from the given TT. They are defined in the root of the *pre* or *run* methods. Global data are placed into the simulator's memory during initial processing of a TT.
2) *Test case level data* that are defined and used by specific test cases. Such data are placed into the simulator's memory when the test case is being generated.

The data method has two optional parameters:

- *global* - a flag that states that the data definition should be treated as global regardless of the context.
- *separate_file* - a flag that specifies whether the generated data definitions should be placed into a separate source code file.

Here is an example of a data definition:

```
data(:global => true, :separate_file => false) {
  org 0x00001000
  label :byte_values
  byte 1, 2, 3, 4
  label :word_values
  word 0xDEADBEEF, 0xBAADF00D
}
```

The above code defines global data: four byte values and two word values. Memory is allocated at offset *0x00001000*. Data values are aligned by their size (1 and 4 bytes). Labels *byte_values* and *word_values* point at the beginning of the byte and the word arrays correspondingly.

4.6 Describing Instruction Calls

To describe instruction calls, the TTDL provides runtime methods that are defined using the metaprogramming facilities of Ruby on the basis of information provided by the model. Methods have the same names and parameters as operations describing corresponding instructions, which are defined in ISA specifications. Operations use parameters of three kinds:

1) *Immediate values* that represent constants.
2) *Addressing modes* that encapsulate logic of reading and writing data to memory resources. Usually they provide access to registers or memory.
3) *Operations* that specify operations to be performed as a part of execution of the current operation. They are used to describe complex instructions composed of several operations (e.g. VLIW instructions).

For example, a call to the add instruction from the MIPS ISA [13], which adds two general-purpose registers *t0 ($8)*, *t1 ($9)* and *t2 ($10)* described by the *reg* addressing mode, can be specified in the following way:

```
add reg(8), reg(9), reg(10)
```

The TTDL supports creating *aliases* for addressing modes and operations invoked with certain arguments. Aliases help make TTs more human-readable. They are created by defining Ruby functions with corresponding names. The code below shows how to create aliases for the registers from the previous example:

```
def t0 reg(8) end
def t1 reg(9) end
def t2 reg(10) end
```

Now the arguments of the add instruction can be specified using alises:

```
add t0, t1, t2
```

Also, the TTDL provides the *pseudo* function that can be used to specify calls to *pseudo instructions* that do not have corresponding operations in ISA specifications. They print user-specified text, but are not simulated by the generator. Here is an example:

```
pseudo 'syscall'
```

4.7 Defining Groups

Addressing modes and operations can be organized into *groups*. Groups are used when it is required to randomly select an addressing mode or operation from the specified set. Groups can be defined in ISA specifications or in TTs. To do this in TTs, the *define_mode_group* and *define_op_group* functions are used. Both functions take the *name* and *distribution* arguments that specify the group name and the distribution used to select its items.

For example, the code below defines an instruction group called *alu* that contains instructions *add*, *sub*, *and*, *or*, *nor*, and *xor* selected randomly according to the specified distribution:

```
alu_dist = dist(
  range(:value => 'add', :bias => 40),
  range(:value => 'sub', :bias => 30),
  range(:value => ['and', 'or', 'nor', 'xor'], :bias => 30)
)
define_op_group('alu', alu_dist)
```

The following code specifies three calls that use instructions randomly selected from the *alu* group:

```
alu t0, t1, t2
alu t3, t4, t5
alu t6, t7, t8
```

4.8 Describing Instruction Call Sequences

Instruction call sequences are described using block-like structures. Each block specifies a sequence or a collection of sequences. Blocks can be nested to construct complex sequences. The algorithm used for sequence construction depends on the type and the attributes of a block.

An individual instruction call is considered a primitive block describing a single sequence that consists of a single instruction call. A single sequence that consists of multiple calls can be described using the *sequence* or the *atomic* construct. The difference between the two is that an atomic sequence is never mixed with other instruction calls when sequences are merged. The code below demonstrates how to specify a sequence of three instruction calls:

```
sequence {
  add t0, t1, t2
  sub t3, t4, t5
  or t6, t7, t8
}
```

A collection of sequences that are processed one by one can be specified using the *iterate* construct. For example, the code below describes three sequences consisting of one instruction call:

```
iterate {
  add t0, t1, t2
  sub t3, t4, t5
  or t6, t7, t8
}
```

Sequences can be combined using the *block* construct. The resulting sequences are constructed by sequentially applying the following engines to sequences returned by nested blocks:

- *combinator* - builds combinations of sequences returned by nested blocks. Each combination is a tuple of length equal to the number of nested blocks.
- *permutator* - modifies combinations returned by combinator by rearranging some sequences.

- *compositor* - merges (multiplexes) sequences in a combination into a single sequence preserving the initial order of instructions calls in each sequence.
- *rearranger* - rearranges sequences constructed by compositor.
- *obfuscator* - modifies sequences returned by rearranger by permuting some instruction calls.

Each engine has several implementations based on different methods. It is possible to extend the list of supported methods with new implementations. Specific methods are selected by specifying corresponding block attributes. When they are not specified, default methods are applied. The format of a block structure for combining sequences looks as follows:

```
block(
  :combinator => 'combinator-name',
  :permutator => 'permutator-name',
  :compositor => 'compositor-name',
  :rearranger => 'rearranger-name',
  :obfuscator => 'obfuscator-name') {
  # Block A. 3 sequences of length 1: {A11}, {A21}, {A31}
  iterate { A11; A21; A31 }
  # Block B. 2 sequences of length 2: {B11, B12}, {B21, B22}
  iterate { sequence { B11, B12 }; sequence { B21, B22 } }
  # Block C. 1 sequence of length 3: {C11, C12, C13}
  iterate { sequence { C11; C12; C13 } }
}
```

The default method names are: *diagonal* for combinator, *catenation* for compositor, and *trivial* for permutator, rearranger and obfuscator. Such a combination of engines describes a collection of sequences constructed as a concatenation of sequences returned by nested blocks. For example, sequences constructed for the block in the above example will be as follows: *{A11, B11, B12, C11, C12, C13}*, *{A21, B21, B22, C11, C12, C13}* and *{A31, B11, B12, C11, C12, C13}*.

4.9 Specifying Test Situations

Test situations are associated with specific instruction calls and specify methods used to generate their input data. There is a wide range of data generation methods implemented by various data generation engines. Test situations are specified using the *situation* construct. It takes the situation name and a map of optional attributes that specify situation-specific parameters. For example, the following line of code causes input registers of the add instruction to be filled with zeros:

```
add t1, t2, t3 do situation('zero') end
```

When no situation is specified, a default situation is used. This situation places random values into input registers. It is possible to assign a custom default situation for individual instructions and instruction groups with the *set_default_situation* function. For example:

```
set_default_situation 'add' do situation('zero') end
```

Situations can be selected at random. The selection is based on a distribution. This can be done by using the *random_situation* construct. For example:

```
sit_dist = dist(
  range(:value => situation('add.overflow')),
  range(:value => situation('add.normal')),
  range(:value => situation('zero')),
  range(:value => situation('random', :dist => int_dist))
  )
add t1, t2, t3 do random_situation(sit_dist) end
```

Unknown immediate arguments that should have their values generated are specified using the "_" symbol. For example, the code below states that a random value should be added to a value stored in a random register and the result should be placed to another random register:

```
addi reg(_), reg(_), _ do situation('random') end
```

4.10 Selecting Registers

Unknown immediate arguments of addressing modes are a special case and their values are generated in a slightly different way. Typically, they specify register indexes and are bounded by the lenght of register arrays. Often such indexes must be selected from a specific range taking into account previous selections. For example, registers are allocated at random and they must not overlap. To be able to solve such tasks, all values passed to addressing modes are tracked. The allowed value range and the method of value selection are specified in configuration files. Values are selected using the specified method before the instruction call is processed by the engine that generates data for the test situation. The selection method can be customized by using the *mode_allocator* function. It takes the allocation method name and a map of method-specific parameters. For example, the following code states that the output register of the add instruction must be a random register which is not used in the current test case:

```
add reg(_ mode_allocator('free')), t0, t1
```

Also, the TTDL allows customizing the allowed range for selected values. It is possible to exclude some elements from the range by using the *exclude* attribute or to provide a new range by using the *retain* attribute. For example:

```
add reg(_ :exclude=>[1, 5, 7]), t0, t1
add reg(_ :retain=>8..15), t0, t1
```

Addressing modes with specific argument values can be marked as free using the *free_allocated_mode* function. To free all allocated addressing modes, the *free_all_allocated_modes* function can be used.

4.11 Describing Preparators

Preparators describe instruction sequences that place data into registers or memory accessed via the specified addressing mode. They are inserted into TPs to set up the

initial state of the microprocessor required by test situations. It is possible to overload preparators for specific cases (value masks, register numbers, etc). Preparators are defined in the *pre* method using the *preparator* construct, which uses the following parameters describing conditions under which it is applied:

- *target* - the name of the target addressing mode;
- *mask* (optional) - the mask that should be matched by the value in order for the preparator to be selected;
- *arguments* (optional) - values of the target addressing mode arguments that should be matched in order for the preparator to be selected;
- *name* (optional) - the name that identifies the current preparator to resolve ambiguity when there are several different preparators that have the same target, mask and arguments.

It is possible to define several *variants* of a preparator which are selected at random according to the specified distribution. They are described using the *variant* construct. It has two optional parameters:

- *name* (optional) - identifies the variant to make it possible to explicitly select a specific variant;
- *bias* - specifies the weight of the variant, can be skipped to set up an even distribution.

Here is an example of a preparator what places a value into a 32-bit register described by the *REG* addressing mode and two its special cases for values equal to *0x00000000* and *0xFFFFFFFF*:

```
preparator(:target => 'REG') {
  variant(:bias => 25) {
    data {
      label :preparator_data
      word value
    }
    la at, :preparator_data
    lw target, 0, at
  }
  variant(:bias => 75) {
    lui target, value(16, 31)
    ori target, target, value(0, 15)
  }
}
preparator(:target => 'REG', :mask => '00000000') {
  xor target, zero, zero
}
preparator(:target => 'REG', :mask => 'FFFFFFFF') {
  nor target, zero, zero
}
```

Code inside the *preparator* block uses the *target* and *value* functions to access the target addressing mode and the value passed to the preparator.

The TTDL provides the *prepare* function to explicitly insert preparators into TPs. It can be used to create composite preparators. The function has the following arguments:

- *target* - specifies the target addressing mode;
- *value* - specifies the value to be written;
- *attrs* (optional) - specifies the preparator name and the variant name to select a specific preparator.

For example, the following line of code places value *0xDEADBEEF* into the *t0* register:

```
prepare t0, 0xDEADBEEF
```

4.12 Describing Self-Checks

TPs can include self-checks that check validity of the microprocessor state after a test case has been executed. These checks are instruction sequences inserted in the end of test cases which compare values stored in registers with expected values. If the values do not match control is transferred to a handler that reports an error. Expected values are produced by the MicroTESK simulator. Self-check are described using the *comparator* construct which has the same features as the *preparator* construct, but serves a different purpose. Here is an example of a comparator for 32-bit registers and its special case for value equal to *0x00000000*:

```
comparator(:target => 'REG') {
  prepare target, value
  bne at, target, :check_failed
  nop
}
comparator(:target => 'REG', :mask => "00000000") {
  bne zero, target, :check_failed
  nop
}
```

4.13 Describing Test Cases

A TP can be described by the following formula: $\Pi = \Pi_{start} \cdot \{\langle \pi_{start}, x_i, \pi_{stop} \rangle\}_{i=1,n} \cdot \Pi_{stop}$, where:

- Π_{start} is a TP prologue that consists of instructions aimed for microprocessor initialization;
- $\langle \pi_{start}, x_i, \pi_{stop} \rangle$ is a test case that specifies an individual stimulus and consists of:
  - π_{start} is a test case prologue that performs all necessary preparations for the test case;

- x_i is a test case action that contains the main code of the test case;
- π_{stop} is a test case epilogue that performs finalization actions for the test case such as self-checks.

- Π_{stop} is a TP epilogue that consists of instructions aimed for microprocessor finalization;
- *n* is the number of test cases in a TP.

The TTDL provides means of describing each part of a TP. Π_{start} and Π_{stop} are described in the *pre* and *post* methods of a TT class correspondingly. Test cases are specified in the *run* method.

Test cases are described by block constructs specifying one or more sequences of instruction calls. Each sequence is a separate test case. It is possible to process a block multiple times. This makes sense when sequences use randomization. In this case, it results different test cases based on the same description. For example, the code below describes five test cases based on the same sequence of three calls. Input data for the calls are generated at random and will be different for all test cases.

```
def run
  sequence {
    add t0, t1, t2
    sub t3, t4, t5
    or t6, t7, t8
  }.run 5
end
```

π_{start} that contains preparators for input registers and π_{stop} that contains self-checks will be generated by the tool automatically. Also, it is possible to specify additional prologue and epilogue for test cases. They will be inserted between automatically generated prologue and epilogue and main code of the test cases. They are specified using the *prologue* and *epilogue* blocks nested into the sequence block. The syntax looks like this:

```
sequence {
  prologue { ... }
  ...
  epilogue { ... }
}.run n
```

When instruction sequences are merged by nesting blocks, prologue and epilogue of nested blocks wrap sequences returned by these blocks.

Test cases can be processed by different TPG engines. A specific engine can be selected by passing the *engine* parameter to the block construct that describes the test cases.

4.14 Describing Exception Handlers

TPs must contain handlers of exceptions that may occur during their execution. Exception handlers are described using the *exception_handler* construct. This

description is also used by the MicroTESK simulator to handle exceptions. Separate exception handlers are described using the *section* construct nested into the *exception_handler* block. The *section* function has two arguments: *org* that specifies the handler's location in memory and *exception* that specifies names of associated exceptions. For example, the code below describes a handler for the *IntegerOverflow*, *SystemCall* and *Breakpoint* exceptions, which resumes execution from the next instruction:

```
exception_handler {
  section(:org =>0x380, :exception => ['IntegerOverflow', 'SystemCall', 'Breakpoint']) {
    mfc0 ra, cop0(14)
    addi ra, ra, 4
    jr ra
    nop
  }
}
```

4.15 Printing Text

TPs are printed in textual form to source code files. The printed text includes various supplementary messages such as comments and separators. They are generated by MicroTESK engines or specified by users in TTs. The format of printed text is set up using the following settings:

- *sl_comment_starts_with* - starting characters for single-line comments. Default value is "//".
- *ml_comment_starts_with* - starting characters for multi-line comments. Default value is "/*".
- *ml_comment_ends_with* - terminating characters for multi-line comments. Default value is "*/".
- *indent_token* - indentation token. Default value is "\t".
- *separator_token* - token used in separator lines. Default value is "=".

The settings are initialized with default values in the *initialize* method of the *Template* class can be redefined in the *initialize* method of a TT.

The TTDL provides functions for printing custom text messages. Text messages are printed either into the generated source code or into the simulator log. Here is the list of supported functions:

- *newline* - adds the new line character into the TP;
- *text* - adds text into the TP;
- *trace* - prints text into the simulator execution log;
- *comment* - adds a comment into the TP;
- *start_comment* - starts a multi-line comment;
- *end_comment* - ends a multi-line comment.

The *text*, *trace* and *comment* functions print formatted text. They take a format string and an array of objects to be printed, which can be constants or memory locations. To specify locations to be printed (registers, memory), the *location* function should be used. It takes the name of the memory array and the index of the selected element. For example, the code below prints a constant value and a value stored in a register in the hexadecimal format:

```
text 'Constant: 0x%X', 0xDEADBEEF
text 'Register: 0x%X', location('GPR', 8)
```

5. Case Study

MicroTESK and its TTDL have been applied in industrial projects to generate TPs for MIPS64 [13] and ARMv8 [14] microprocessors. Table 1 provides characteristics of the MIPS64 and ARMv8 specifications used to configure MicroTESK for generating TPs for these designs.

Table 1. Industrial application of the proposed TTDL and supporting tool

Project	**MIPS64**	**ARMv8**
Number of instructions	102	207
ISA specification size (lines of code)	70	143
MMU specification size (lines of code)	134	637
Efforts (person-months)	101	809

Created tests include:

- tests for arithmetical instructions;
- tests for floating-point instructions;
- tests for branch instructions;
- tests for memory access instructions.

To describe tests for branch and memory instruction, the TTDL was extended with additional constructs based on existing ones. The language was evolving in the process of working on the projects. Some features were changed and some were added. A number of language features came as requirements from customers. The approach based on using dynamic languages such as Ruby to create TTDLs has proved its flexibility. The TTDL allowed describing test cases in a format which is maximally close to assembly language for corresponding microprocessors. This allows verification engineers to concentrate on verification problems instead of issues related to the use of a specific programming language.

5. Conclusion

A concept of a TTDL for a reconfigurable and extensible TPG framework has been considered. The proposed solution was implemented in the MicroTESK [3] framework. The developed TTDL is based on the Ruby [6] language and uses its metaprogramming facilities to dynamically add configuration-dependent language

constructs. The language is integrated into MicroTESK, which is a Java-based tool, with the help of JRuby [11]. Facilities of the TTDL can be extended by adding new Ruby libraries.

Directions for further research and development are to apply the described principles to create TTDLs based on other programming languages. First of all, it is Python and its Java-based implementation called Jython. It provides facilities similar to those of Ruby and is also popular among verification engineers. For this reason, it would be advantageous to provide a Python-based version of the TTDL for those who are more comfortable with this language.

Another task is development of a TTDL based on C++. It will be a part of a large research project dedicated to *on-line* generation. An on-line TPG tool is represented by a binary image with basic functions of an operating system, which is loaded directly to a microprocessor chip where it generates and executes test stimuli. The tool will be created by MicroTESK from C++ libraries based on formal specifications. For further unification of TPG tools, it is important that TTs for on-line generation are developed using the same principles.

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, 2004. pp. 84–93.

[2]. 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.

[3]. MicroTESK page. http://forge.ispras.ru/projects/microtesk

[4]. A. Kamkin, E. Kornykhin, D. Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors. International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47–54.

[5]. A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko. Extensible Environment for Test Program Generation for Microprocessors. Programming and Computer Software, 40(1), 2014. pp. 1-9.

[6]. Ruby site: http://www.ruby-lang.org

[7]. E.A. Poe. Introduction to Random Test Generation for Processor Verification. Obsidian Software, 7 pp, 2002.

[8]. RAVEN test program generator. Available at: http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc

[9]. Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. Random Test Program Generation for Reconfigurable Architectures. 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.

[10]. A. Kamkin. Test Program Generation for Microprocessors. Trudy ISP RAN / Proc. ISP RAS], vol. 14, part 2, 2008, pp. 23-64 (in Russian).

[11]. JRuby site: http://www.jruby.org

[12]. Flanagan D., Matsumoto Y. The Ruby Programming Language. OReilly Media, Sebastopol, 2008.
[13]. MIPS64TM Architecture For Programmers. Volume II: The MIPS64TM Instruction Set, Document Number: MD00087, Revision 2.00, June 9, 2003.
[14]. ARM Architecture Reference Manual. ARM DDI 0487A.f, ARM Corporation, 2015. 5886 p.

Язык описания шаблонов для генерации тестовых программ для микропроцессоров

А.Д. Татарников <andrewt@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Генерация тестовых программ на языке ассемблера и проверка корректности результатов их выполнения является наиболее широко применяемым подходом к функциональной верификации микропроцессоров. Данная задача решается при помощи специальных автоматизированных средств, называемых генераторами тестовых программ. Высокая сложность современных электронных устройств создает потребность в автоматизированных средствах, способных генерировать тестовые программы, покрывающие нетривиальные ситуации в их работе. Большинство таких средств используют в качестве входных данных шаблоны тестовых программ, которые позволяют описывать тестовые сценарии в абстрактном виде. Такой подход предоставляет инженерам-верификаторам возможность описывать широкий спектр задач генерации, затрачивая минимальные усилия. Шаблоны тестовых программ разрабатываются на специальных предметно-ориентированных языках. Такие языки должны удовлетворять следующим требованиям: (1) они должны быть достаточно простыми для использования инженерами-верификаторами, не обладающими серьезными навыками программирования; (2) они должны быть применимы для широкого спектра микропроцессорных архитектур и (3) они должны быть легко расширяемы для поддержки описания новых типов задач генерации. В данной работе рассматривается язык описания шаблонов тестовых программ, который был создан для расширяемой среды генерации тестовых программ MicroTESK, разрабатываемой в ИСП РАН. Это гибкий предметно-ориентированный язык, основанный на языке Ruby, который позволяет описывать широкий набор задач генерации в терминах абстракций цифровой аппаратуры. Среда генерации MicroTESK и язык описания тестовых шаблонов успешно применяются в промышленных проектах по верификации микропроцессоров на базе архитектур MIPS и ARM.

Ключевые слова: микропроцессоры; функциональная верификация; генерация тестовых программ; тестовые шаблоны; предметно-ориентированные языки.

DOI: 10.15514/ISPRAS-2016-28(4)-5

Для цитирования: Татарников А.Д. Язык описания шаблонов для генерации тестовых программ для микропроцессоров. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 77-98. DOI: 10.15514/ISPRAS-2016-28(4)-5

Список литературы

[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, 2004. pp. 84–93.

[2]. 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.

[3]. Инструмента MicroTESK. http://forge.ispras.ru/projects/microtesk

[4]. A. Kamkin, E. Kornykhin, D. Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors. International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47–54.

[5]. Камкин А.С., Сергеева Т.И., Смолов С.А., Татарников А.Д., Чупилко М.М. Расширяемая среда генерации тестовых программ для микропроцессоров. Программирование, № 1, 2014, стр. 3-14.

[6]. Язык Ruby: http://www.ruby-lang.org.

[7]. E.A. Poe. Introduction to Random Test Generation for Processor Verification. Obsidian Software, 7 pp, 2002.

[8]. Инструмент RAVEN: http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc.

[9]. Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. Random Test Program Generation for Reconfigurable Architectures. 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.

[10]. А.С. Камкин. Генерация тестовых программ для микропроцессоров. Труды ИСП РАН, 14(2), 2008. С. 23-63.

[11]. Интерпретатор JRuby: http://www.jruby.org.

[12]. Flanagan D., Matsumoto Y. The Ruby Programming Language. OReilly Media, Sebastopol, 2008.

[13]. MIPS64TM Architecture For Programmers. Volume II: The MIPS64TM Instruction Set, Document Number: MD00087, Revision 2.00, June 9, 2003.

[14]. ARM Architecture Reference Manual. ARM DDI 0487A.f, ARM Corporation, 2015. 5886 p.