Institute for System Programming of the Russian Academy of Sciences

MicroTESK User Guide (UNDER DEVELOPMENT)

Moscow 2016

Contents

Chapter 1

Overview

Chapter 2

Installation

2.1 System Requirements

MicroTESK is a set of Java-based utilities that are run from the command line. It can be used on ***Windows***, ***Linux*** and ***OS X*** machines that have ***JDK 1.7 or later*** installed. To build MicroTESK from source code or to build the generated Java models, ***Apache Ant version 1.8 or later*** is required. To generate test data based on constraints, MicroTESK needs the ***Microsoft Research Z3*** or ***CVC4*** solver that can work on the corresponding operating system.

2.2 Installation Steps

To install MicroTESK, the following steps should be performed:

1. Download from `http://forge.ispras.ru/projects/microtesk/files` and unpack the MicroTESK installation package (the `.tar.gz` file, latest release) to your computer. The folder to which it was unpacked will be further referred to as the installation directory (`<installation dir>`).

2. Declare the `MICROTESK_HOME` environment variable and set its value to the path to the installation directory (see the Setting Environment Variables section).

3. Set the `<installation dir>/bin` folder as the working directory (add the path to the `PATH` environment variable) to be able to run MicroTESK utilities from any path.

4. Note: Required for constraint-based generation. Download and install constraint solver tools to the `<installation dir>/tools` directory (see the Installing Constraint Solvers section).

2.2.1 Setting Environment Variables

Windows

1. Open the `System Properties` window.

2. Switch to the `Advanced` tab.

3. Click on `Environment Variables`.

4. Click `New..` under `System Variables`.

5. In the `New System Variable` dialog, specify variable name as `MICROTESK_HOME` and variable value as `<installation dir>`.

6. Click `OK` on all open windows.

7. Reopen the command prompt window.

Linux and OS X

Add the command below to the `~.bash_profile` file (***Linux***) or the `~/.profile` file (***OS X***):

```
export MICROTESK_HOME=<installation dir>
```

To start editing the file, type `vi ~/.bash_profile` (or `vi ~/.profile`). Changes will be applied after restarting the command-line terminal or reboot. You can also run the command in your command-line terminal to make temporary changes.

2.2.2 Installing Constraint Solvers

To generate test data based on constraints, MicroTESK requires external constraint solvers. The current version supports the Z3 and CVC4 constraint solvers. Constraint executables should be downloaded and placed to the `<installation dir>/tools` directory.

Using Environment Variables

If solvers are already installed in another directory, to let MicroTESK find them, the following environment variables can be used: `Z3_PATH` and `CVC4_PATH`. They specify the paths to the Z3 and CVC4 excutables correspondingly.

Installing Z3

- ***Windows*** users should download Z3 (32 or 64-bit version) from the following page:`http://z3.codeplex.com/releases` and unpack the archive to the `<installation dir>/tools/z3/windows` directory. Note: the executable file path is `<windows>/z3/bin/z3.exe`.

- ***UNIX*** and ***Linux*** users should use one of the links below and and unpack the archive to the `<installation dir>/tools/z3/unix` directory. Note: the executable file path is `<unix>/z3/bin/z3`.

Debian x64	`http://z3.codeplex.com/releases/view/101916`
Ubuntu x86	`http://z3.codeplex.com/releases/view/101913`
Ubuntu x64	`http://z3.codeplex.com/releases/view/101911`
FreeBSD x64	`http://z3.codeplex.com/releases/view/101907`

- ***OS X*** users should download Z3 from `http://z3.codeplex.com/releases/view/101918` and unpack the archive to the `<installation dir>/z3/osx` directory. Note: the executable file path is `<osx>/z3/bin/z3`.

Installing CVC4

- ***Windows*** users should download the latest version of CVC4 binary from `http://cvc4.cs.nyu.edu/builds/win32-opt/` and save it to the `<installation dir>/tools/cvc4/windows` directory as `cvc4.exe`.
- ***Linux*** users download the latest version of CVC4 binary from `http://cvc4.cs.nyu.edu/builds/i386-linux-opt/unstable/` (32-bit version) or `http://cvc4.cs.nyu.edu/builds/x86_64-linux-opt/unstable/` (64-bit version) and save it to the `<installation dir>/tools/cvc4/unix` directory as `cvc4`.
- ***OS X*** users should download the latest version of CVC4 distribution package from `http://cvc4.cs.nyu.edu/builds/macos/` and install it. The CVC4 binary should be copied to `<installation dir>/tools/cvc4/osx` as `cvc4` or linked to this file name via a symbolic link.

2.3 Installation Directory Structure

The MicroTESK installation directory contains the following subdirectories:

arch	Microprocessor specifications and test templates
bin	Scripts to run modeling and test generation tasks
doc	Documentation
etc	Configuration files
gen	Generated code of microprocessor models
lib	JAR files and Ruby scripts to perform modeling and test generation tasks
src	Source code of MicroTESK

2.4 Running

To generate a Java model of a microprocessor from its nML specification, a user needs to run the `compile.sh` script (Unix, Linux, OS X) or the `compile.bat` script (Windows). For example, the following command generates a model for the miniMIPS specification:

```
sh bin/compile.sh arch/minimips/model/minimips.nml
```

NOTE: Models for all demo specifications are already built and included in the MicroTESK distribution package. So a user can start working with MicroTESK from generating test programs for these models.

To generate a test program, a user needs to use the `generate.sh` script (Unix, Linux, OS X) or the `generate.bat` script (Windows). The scripts require the following parameters:

- model name;
- test template file;
- target test program source code file.

For example, the command below runs the `euclid.rb` test template for the miniMIPS model generated by the command from the previous example and saves the generated test program to an assembler file. The file name is based on values of the `-code-file-prefix` and `-code-file-extension` options.

```
sh bin/generate.sh minimips arch/minimips/templates/euclid.rb
```

To specify whether Z3 or CVC4 should be used to solve constraints, a user needs to specify the `-s` or `-solver` command-line option as `z3` or `cvc4` respectively. By default, Z3 will be used. Here is an example:

```
sh bin/generate.sh -s cvc4 minimips arch/minimips/templates/constraint.rb
```

More information on command-line options can be found on the Command-Line Options section.

2.5 Command-Line Options

MicroTESK works in two modes: *specification translation* and *test generation*, which are enabled with the `-translate` (used by default) and `-generate` keys correspondingly. In addition, the `-help` key prints information on the command-line format.

The `-translate` and `-generate` keys are inserted into the command-line by `compile.sh`/`compile.bat` and `generate.sh`/`generate.bat` scripts correspondingly. Other options should be specified explicitly to customize the behavior of MicroTESK. Here is the list of options:

Full name	**Short name**	**Description**	**Requires**
–help	-h	Shows help message	
–verbose	-v	Enables printing diagnostic messages	
–translate	-t	Translates formal specifications	
–generate	-g	Generates test programs	
–output-dir <arg>	-od	Sets where to place generated files	
–include <arg>	-i	Sets include files directories	–translate
–extension-dir <arg>	-ed	Sets directory that stores user-defined Java code	–translate
–random-seed <arg>	-rs	Sets seed for randomizer	–generate
–solver <arg>	-s	Sets constraint solver engine to be used	–generate
–branch-exec-limit <arg>	-bel	Sets the limit on control transfers to detect endless loops	–generate
–solver-debug	-sd	Enables debug mode for SMT solvers	–generate
–tarmac-log	-tl	Saves simulator log in Tarmac format	–generate
–self-checks	-sc	Inserts self-checking code into test programs	–generate
–arch-dirs <arg>	-ad	Home directories for tested architectures	–generate
–rate-limit <arg>	-rl	Generation rate limit, causes error when broken	–generate
–code-file-extension <arg>	-cfe	The output file extension	–generate

–code-file-prefix <arg>	-cfp	The output file prefix (file names are as follows prefix_xxxx.ext, where xxxx is a 4-digit decimal number)	–generate
–data-file-extension <arg>	-dfe	The data file extension	–generate
–data-file-prefix <arg>	-dfp	The data file prefix	–generate
–exception-file-prefix <arg>	-efp	The exception handler file prefix	–generate
–program-length-limit <arg>	-pll	The maximum number of instructions in output programs	–generate
–trace-length-limit <arg>	-tll	The maximum length of execution traces of output programs	–generate
–comments-enabled	-ce	Enables printing comments; if not specified no comments are printed	–generate
–comments-debug	-cd	Enables printing detailed comments; must be used together with –comments-enabled	–generate
–no-simulation	-ns	Disables simulation of generated test programs on the model	–generate

2.6 Settings File

Default values of options are stored in the `<MICROTESK_HOME>/etc/settings.xml` configururation file that has the following format:

```
<?xml version="1.0" encoding="utf-8"?>
<settings>
  <setting name="random-seed" value="0"/>
  <setting name="branch-exec-limit" value="1000"/>
  <setting name="code-file-extension" value="asm"/>
  <setting name="code-file-prefix" value="test"/>
  <setting name="data-file-extension" value="dat"/>
  <setting name="data-file-prefix" value="test"/>
  <setting name="exception-file-prefix" value="test_except"/>
  <setting name="program-length-limit" value="1000"/>
  <setting name="trace-length-limit" value="1000"/>
  <setting name="comments-enabled" value=""/>
  <setting name="comments-debug" value=""/>
  <setting
    name="arch-dirs"
    value="cpu=arch/demo/cpu/settings.xml:minimips=arch/minimips/settings.xml"
  />
```

```
</settings>
```

Chapter 3

Test Templates

3.1 Introduction

MicroTESK generates test programs on the basis of *test templates* that describe test programs to be generated in an abstract way. Test templates are created using special Ruby-based test template description language that derives all Ruby features and provides special facilities. The language is implemented as a library that implements facilities for describing test cases. Detailed information on Ruby features can be found in official documentation [2, 3].

MicroTESK uses the JRuby [4] interpreter to process test templates. This allows Ruby libraries to interact with other components of MicroTESK written in Java.

Test templates are processed in two stages:

1. Ruby code is executed to build the internal representation (a hierarchy of Java objects) of the test template.

2. The internal representation is processed with various engines to generate test cases which are then simulated on the reference model and printed to files.

This chapter describes facilities of the test template description language and supported test generation engines.

3.2 Test Template Structure

A test template is implemented as a class inherited from the `Template` library class that provides access to all features of the library. Information on the location of the `Template` class is stored in the `TEMPLATE` environment variable. Thus, the definition of a test template class looks like this:

```
require ENV['TEMPLATE']

class MyTemplate < Template
```

Test template classes should contain implementations of the following methods:

1. `initialize` (optional) - specifies settings for the given test template;

2. `pre` (optional) - specifies the initialization code for test programs;
3. `post` (optional) - specifies the finalization code for test programs;
4. `run` - specifies the main code of test programs (test cases).

The definitions of optional methods can be skipped. In this case, the default implementations provided by the parent class will be used. The default implementation of the `initialize` method initializes the settings with default values. The default implementations of the `pre` and `post` methods do nothing.

The full interface of a test template looks as follows:

```
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 test problem description here
  end

end
```

3.3 Reusing Test Templates

It is possible to reuse code of existing test templates in other test templates. To do this, you need to subclass the template you want to reuse instead of the `Template` class. For example, the `MyTemplate` class below reuses code from the `MyPrepost` class that provides initialization and finalization code for similar test templates.

```
require ENV['TEMPLATE']
require_relative 'MyPrepost'

class MyTemplate < MyPrepost

  def run
    ...
  end

end
```

Another way to reuse code is creating code libraries with methods that can be called by test templates. A code library is defined as a Ruby module file and has the following structure:

```
module MyLibrary

  def method1
    ...
  end

  def method2(arg1, arg2)
    ...
  end

  def method3(arg1, arg2, arg3)
    ...
  end

end
```

To be able to use utility methods `method1`, `method2` and `method3` in a test template, the `MyLibrary` module must be included in that test template as a mixin. Once this is done, all methods of the library are available in the test template. Here is an example:

```
require ENV['TEMPLATE']
require_relative 'my_library'

class MyTemplate < Template
  include MyLibrary

  def run
    method1
    method2 arg1, arg2
    method3 arg1, arg2, arg3
  end

end
```

3.4 Test Template Settings

3.4.1 Managing Text Format

Test templates use the following settings that set up the format of generated test programs:

- `sl_comment_starts_with` - starting characters for single-line comments;
- `ml_comment_starts_with` - starting characters for multi-line comments;
- `ml_comment_ends_with` - terminating characters for multi-line comments;
- `indent_token` - indentation token;

- separator_token - token used in separator lines.

Here is how these settings are initialized with default values in the Template class:

```
@sl_comment_starts_with = "//"
@ml_comment_starts_with = "/*"
@ml_comment_ends_with   = "*/"

@indent_token     = "\t"
@separator_token = "="
```

The settings can be overridden in the initialize method of a test template. For example:

```
class MyTemplate < Template

  def initialize
    super

    @sl_comment_starts_with = ";"
    @ml_comment_starts_with = "/="
    @ml_comment_ends_with = "=/"

    @indent_token = "  "
    @separator_token = "*"
  end
  ...
end
```

3.4.2 Managing Address Alignment

The .align n directive may have different interpretation for different assemblers. By default, MicroTESK assumes that it aligns an address to the next 2^n byte boundary. If this is not the case, to make MicroTESK correctly interpret it, the alignment_in_bytes function must be overridden in a test template. This function returns the number of bytes that correponds to n. The default implementation of the function looks like this:

```
#
# By default, align n is interpreted as alignment on 2**n byte border.
# This behavior can be overridden.
#
def alignment_in_bytes(n)
  2 ** n
end
```

3.5 Text Printing

The test template description language provides facilities for printing text messages. Text messages are printed either into the generated source code or into the simulator log. Here is the list of functions that print text:

- `newline` - adds the new line character into the test program;
- `text(format, *args)` - adds text into the test program;
- `trace(format, *args)` - prints text into the simulator execution log;
- `comment(format, *args)` - adds a comment into the test program;
- `start_comment` - starts a multi-line comment;
- `end_comment` - ends a multi-line comment.

Formatted Printing

Functions `text`, `trace` and `comment` print formatted text. They take a format string and a variable list of arguments that provide data to be printed.

Supported argument types:

- constants;
- locations.

To specify locations to be printed (registers, memory), the `location(name, index)` function should be used. It takes the name of the memory array and the index of the selected element.

Supported format characters:

- `d` - decimal format;
- `x` or `X` - hexadecimal format (lowercase or uppercase letters);
- `s` - decimal format for constants and binary format for locations.

For example, the code below prints the `0xDEADBEEF` value as a constant and as a value stored in a register using different format characters:

```
prepare reg(1), 0xDEADBEEF
reg1 = location('GPR', 1)
text 'Constants: dec=%d, hex=0x%X, str=%s', 0xDEADBEEF, 0xDEADBEEF, 0xDEADBEEF
text 'Locations: dec=%d, hex=0x%X, str=%s', reg1, reg1, reg1
```

Here is how it will be printed:

```
Constants: dec=3735928559, hex=0xDEADBEEF, str=3735928559
Locations: dec=3735928559, hex=0xDEADBEEF, str=11011110101011011011111011101111
```

3.6 Random Distributions

Many tasks involve selection based on *random distribution*. The test template language includes constructs to describe ranges of possible values and their weights. To accomplish this task, the following functions are provided:

- `range(attrs)` - creates a range of values and its weight, which are described by the `:value` and `:bias` attribures. 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 weight is used to describe an even distribution based on ranges with equal weights.

- `dist(*ranges)` - creates 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
  )
```

Distributions are used in a number of test template features that will be described further in this chapter.

3.7 Instruction Calls

The `pre`, `post` and `run` methods of a test template contain descriptions of instruction call sequences. Instructions are operations defined in ISA specifications which represent target assembler instructions. Operations can have arguments of three kinds:

- immediate value;
- addressing mode;
- operation.

Addressing modes encapsulate logic of reading or writing values to memory resources. For example, an addressing mode can refer to a register, a memory location or hold an immediate value. Operations are used to describe complex instructions that are composed of several operations (e.g. VLIW instructions). What arguments are suitable for specific instructions is specified in ISA specifications.

Arguments are passed to instructions and addressing modes in two ways:

- As *arrays*. This format is based on methods with a variable number of arguments. Values are expected to come in the same order as corresponding parameter definitions in specifications.
- As *hash maps*. This format implies that operations and addressing modes are parameterized with hash tables where the key is in the name of the parameter and the value is the value to be assigned to this parameter.

The first way is more preferable as it is simpler and closer to the assembly code syntax. The code below demonstrates both ways (miniMIPS):

```
# Arrays
add reg(11), reg(9), reg(0)
# Hash maps
add :rd=>reg(:i=>11), :rs=>reg(:i=>9), :rt=>reg(:i=>0)
```

3.7.1 Aliases

Sometimes it is required to define *aliases* for addressing modes or operations invoked with certain arguments. This is needed to make a test template more human-readable. This can be done by defining in a test template Ruby functions that create instances with specific arguments. For example, the following code makes it possible to address registers `reg(0)` and `reg(1)` as `zero` and `at`:

```
def zero
  reg(0)
end

def at
  reg(1)
end
```

3.7.2 Pseudo Instructions

It is possible to specifify *pseudo instructions* that do not have correspondent operation in specifications. Such instructions print user-specified text and do not change the state of the reference model. The can be described using the following function: `pseudo(text)`. For example:

```
pseudo 'syscall'
```

3.7.3 Groups

Addressing modes and operations can be organized into *groups*. Groups are used when it is required to randomly select an addressing mode or an operation from the specified set.

Groups can be defined in specifications or in test templates. To define them in test templates, the following functions are used:

- `define_mode_group(name, distribution)` - defines an addressing mode group;
- `define_op_group(name, distribution)` - defined an operation group.

Both function take the `name` and `distribution` arguments that specify the group name and the distribution used to select its items. More information on distributions is in the Random Distribution section. *Notes*: (1) distribution items can be names of addressing modes and operations, by not names of groups; (2) it is not allowed to redefine existing groups.

For example, the code below creates 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
```

3.7.4 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 va-lues 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
```

3.7.5 Registers Selection

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, it is possible to exclude some elements from the range by using the `exclude` attribute. For example:

```
add reg(_ :exclude=>[1, 5, 7]), 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.

3.8 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`}

3.9 Data

3.9.1 Configuration

Defining data requires the use of assembler-specific directives. Information on these directives is not included in ISA specifications and should be provided in test templates. It includes textual format of data directives and mappings between nML and assembler data types used by these directives. Configuration information on data directives is specified in the `data_config` block, which is usually placed in the `pre` method. Only one such block per a test template is allowed. Here is an example:

```
data_config(:text => '.data', :target => 'M') {
  define_type :id => :byte, :text => '.byte', :type => type('card', 8)
  define_type :id => :half, :text => '.half', :type => type('card', 16)
  define_type :id => :word, :text => '.word', :type => type('card', 32)

  define_space :id => :space, :text => '.space', :fillWith => 0
  define_ascii_string :id => :ascii, :text => '.ascii', :zeroTerm => false
  define_ascii_string :id => :asciiz, :text => '.asciiz', :zeroTerm => true
}
```

The block takes the following parameters:

- `text` (compulsory) - specifies the keyword that marks the beginning of the data section in the generated test program;
- `target` (compulsory) - specifies the memory array defined in the nML specification to which data will be placed during simulation;

- `base_virtual_address` (optional) - specifies the base virtual address where data allocation starts. Default value is 0;
- `item_size` (optional) - specifies the size of a memory location unit pointed by address. Default value is 8 bits (or 1 byte).

To set up particular directives, the language provides special methods that must be called inside the block. All the methods share two common parameters: `id` and `text`. The first specifies the keyword to be used in a test template to address the directive and the second specifies how it will be printed in the test program. The current version of MicroTESK provides the following methods:

1. `define_type` - defines a directive to allocate memory for a data element of an nML data type specified by the type parameter;
2. `define_space` - defines a directive to allocate memory (one or more addressable locations) filled with a default value specified by the `fillWith` parameter;
3. `define_ascii_string` - defines a directive to allocate memory for an ASCII string terminated or not terminated with zero depending on the `zeroTerm` parameter.

The above example defines the directives `byte`, `half`, `word`, `ascii` (non-zero terminated string) and `asciiz` (zero terminated string) that place data in the memory array `M` (specified in nML using the `mem` keyword). The size of an addressable memory location is 1 byte.

3.9.2 Definitions

Data are defined using the `data` construct. Data definitions can be added to the test program source code file or placed into a separate source code file. There are two types of data definitions:

- **Global** - defined in the beginning of a test template and can be used by all test cases generated by the test template. Global data definitions can be placed in the root of the `pre` or `run` methods or methods called from these methods. Memory allocation is performed during inital processing of a test template (see stage 1 of template processing).
- **Test case level** - defined and used by specific test cases. Such definitions can be applied multiple times (e.g. when defined in preparators). Memory allocation is performed when a test case is generated (see stage 2 of template processing).

The `data` construct has two optional parameters:

- `global` - a boolean value that states that the data definition should be treated as global regardless of where it is defined.
- `separate_file` - a boolean value that states that the generated data definitions should be placed into a separate source code file.

Predefined methods

Here is the list of methods that can be used in `data` sections:

- `align` - aligns data by the amount n passed as an argument. By default, n means 2^n bytes. How to change this behaviour see here.
- `org` - sets data allocation origin. Can be used to increase the allocation address, but not to descrease it. Its parameter specifies the origin and can be used in two ways:
  1. As **obsolute** origin. In this case, it is specified as a constant value (`org 0x00001000`) and means an offset from the base virtual address.
  2. As **relative** origin. In this case, it is specified using a hash map (`org :delta => 0x10`) and means an offset from the latest data allocation.
- `label` - associates the specified label with the current address.

Configurable methods

Also, here is the list of runtime methods what has been configured in the `data_config` section in the previous example:

- `space` - increases the allocation address by the number of bytes specified by its argument. The allocated space is filled with the value which has been set up by the `define_space` method.
- `byte`, `half`, `word`
- `ascii`, `asciiz`

Here is an example:

```
data {
  org 0x00001000

  label :data1
  byte 1, 2, 3, 4

  label :data2
  half 0xDEAD, 0xBEEF

  label :data3
  word 0xDEADBEEF

  label :hello
  ascii  'Hello'

  label :world
  asciiz 'World'

  space 6
}
```

In this example, data is placed into memory. Data items are aligned by their size (1 byte, 2 bytes, 4 bytes). Strings are allocated at the byte border (addressable unit). For simplicity, in the current version of MicroTESK, memory is allocated starting from the address 0 (in the memory array of the executable model).

3.10 Preparators

Preparators describe instruction sequences that place data into registers or memory accessed via the specified addressing mode. These sequences are inserted into test programs 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.

Also, the `prepare` function can be used to explicitly insert preparators into test programs. 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
```

3.11 Comparators

Test programs 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
}
```

3.12 Exception Handlers

Test programs can provide handlers of exceptions that occur during their execution. Exception handlers are descibed 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
  }
}
```

Chapter 4

Test Engine Branch

4.1 Parameters

- *branch_exec_limit* is an upper bound for the number of executions of a single branch instruction;
- *trace_count_limit* is an upper bound for the number of execution traces to be returned.

More information on the parameters is given in the "Execution Traces Enumeration" section.

4.2 Description

Functioning of the *branch* test engine includes the following steps:

1. construction of a *branch structure* of an abstract test sequence;
2. enumeration of *execution traces* of the branch structure;
3. concretization of the test sequence for each execution trace:
   (a) construction of a *control* code;
   (b) construction of an *initialization* code.

Let D be the size of the delay slot for an architecture under scrutiny (e.g., D=1 for MIPS, and D=0 for ARM).

Chapter 5

Customization

5.1 Data Generators

Chapter 6

Appendixes

6.1 References

Bibliography

[1] M. Freericks. *The nML Machine Description Formalism*. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.

[2] *Ruby site* – `http://www.ruby-lang.org`

[3] Flanagan D., Matsumoto Y. *The Ruby Programming Language*. O'Reilly Media, Sebastopol, 2008.

[4] *JRuby site* – `http://jruby.org`