Institute for System Programming of the Russian
Academy of Sciences

MicroTESK User Guide
(UNDER DEVELOPMENT)

Moscow 2016

Contents

1 Installation **3**
1.1 System Requirements 3
1.2 Running MicroTESK 3
1.3 Command-Line Options 4
1.4 Overview 6

2 Test Templates **7**
2.1 Introduction 7
2.2 Test Template Structure 7
2.3 Reusing Test Templates 8
2.4 Test Template Settings 9
2.5 Data Definitions 10

3 Appendixes **12**
3.1 References 12
Bibliography 12

Chapter 1

Installation

1.1 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 under the corresponding operating system.

1.2 Running MicroTESK

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:

1. model name

2. test template file

3. 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.

1.3 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	-dfp	The data file prefix	–generate

1.4 Overview

Chapter 2

Test Templates

2.1 Introduction

MicroTESK generates test programs on the basis of test templates that provide an abstract description of scenarios to be reproduced by the generated programs. Test templates are created using the test template description language. It is a Ruby-based domain-specific language that provides facilities to describe test cases using symbolic names (that refer to a set of data satisfying certain conditions) instead of concrete input data and to manage the structure of the generated test programs. The language is implemented as a library that includes functionality for describing test templates and for further processing these test templates to produce a test program. MicroTESK uses the JRuby interpreter to process Ruby files. This allows Ruby libraries to interact with other components of MicroTESK written in Java.

2.2 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. So, 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 the test program;
3. post (optional) - specifies the finalization code for the test program;
4. run - specifies the main code of the test program (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
```

2.3 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
```

2.4 Test Template Settings

Test templates use the following settings:

1. Starting characters for single-line comments in the test program;
2. Starting characters for multi-line comments in the test program;
3. Terminating characters for multi-line comments in the test program;
4. Indentation token;
5. 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
```

2.5 Data Definitions

Describing data requires the use of assembler-specific directives. Information on these directives in 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 template is allowed. Here is an example:

```
data_config(:text => '.data', :target => 'M', :addressableSize => 8) {
  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 (compulsory):

1. text - specifies the keyword that marks the beginning of the data section of the generated test program;

2. target - specifies the memory array defined in the nML specification to which data will be placed during simulation;

3. addressableSize - specifies the size (in bits) of addressable memory locations.

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 8 bits (or 1 byte).

After all data directives are configured, data can be defined using the data block:

```
data {
  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).

Chapter 3

Appendixes

3.1 References

Bibliography

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