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

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. The language is implemented as a library that includes provides facilities for describing test cases.

MicroTESK uses the JRuby 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

Test templates use the following settings:

1. Starting characters for single-line comments;
2. Starting characters for multi-line comments;
3. Terminating characters for multi-line comments;
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
```

3.5 Data Definitions

3.5.1 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.

3.5.2 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.

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

- `align`
- `org`
- `space`

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).

3.6 Preparators

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

Appendixes

5.1 References

Bibliography

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