Maniac

BY NICOLAS BENOIT

nbenoit@tuxfamily.org

Version 1.0.1 - February 20, 2011

Abstract

This manual describes Maniac, a tool designed to automate the validation and the comparison of multiple variants of a same program. This goal is achieved by compiling each variant in a distinct shared object and generating a program that will successively load and execute them.

Table of contents

1 Introduction

Maniac helps to automate the validation and the comparison of multiple variants of a program.

1.1 Motivation

Writing a testsuite or benchmarks for code generation tools often requires a lot of code that is not stricly part of the routine studied, for example trace checking, data verification or timing. Many testsuites embed those utilities in a library to avoid code duplication. However the code gluing the library and the testcase is left to the developer/tester.

Maniac was written from the point of view that the developer/tester should focus only on the testcase design. To this end, it automatically generates programs that will initialize data, call the testcase and check data against the ones generated by a reference implementation.

1.2 Concepts

This subsection explains the names given to the concepts Maniac deals with.

1.2.1 Variant

A variant is an implementation of a given algorithm.

1.2.2 Generator

A generator is a program that will generate a variant of an input source file.

1.2.3 Plan

A plan is a list of variants to process by Maniac. For each variant, a plan specifies if it has to be generated, checked, timed or whatever.

1.2.4 Loader

A loader is a program generated by Maniac that will call the variants of the studied routine.

2 Quickstart

This section explains how to install and run Maniac.

2.1 Installation

Maniac is a Python script. In order to run it from the command-line, you can either copy it in one of the directories of your PATH, or add Maniac directory to it. One of the following command should suffice :

```
$ cp maniac.py $HOME/.local/bin/maniac

$ sudo cp maniac.py /usr/bin/maniac
```

2.2 Usage & Behavior

Through the command-line, the user specifies which directories Maniac should inspect and which plan(s) it should follow, each directory is assumed to contain a single program. According to the selected plan, Maniac first generates the variants. Then, it compiles each variant in a distinct shared object, and finally generates a source file which will sequentially load, initialize, run, time and/or check the variants' shared objects. After each plan, Maniac outputs a log of the whole process. It can also output this log to an HTML file.

Maniac comes with a few demos :

```
$ cd demos/compiler options
```

Try them out and read their Mania files to understand how it works.

2.3 Command-line Options

The command-line options related to Maniac are listed in table 1

Option	Long Option	Purpose
-G	–nogen	Disable generation of variants
-m M	–mania=M	Look in the provided file for mania description
-n N	–numrpt=N	Repeat each plan N times
-o O	–output=O	Output HTML report to file named O

Table 1. Command-line options

3 Mania Description Files

In order to proceed, Maniac requires the definition of an environment and a program in so-called *mania* files.

Mania files are named 'mania.xml' by convention. A mania file can contain both environment and program descriptions. When Maniac is invoked, it first searches for environment description. The search order is the following : current directory, directories passed as arguments, parent directories. Once the environment is loaded, Maniac looks for program description as follow : current directory and directories passed as arguments.

3.1 Environment Description

The environment describes the variant generators, the variants and the plans.

3.1.1 Generator Description

The section describing the available generators is named *generators*.

For each generator, the attributes are the following :

Attribute	Purpose
name	Name under which the generator will be referenced (reserved: none, copy)
bin	The absolute or relative path to the executable of the generator
flags	The flags to use when invoking the generator

Table 2. Generator Attributes

Here is an example of a generator description for Pocc (more info at `http://pocc.sf.net`) :

```
<generators>
  <generator name=''pocc-tile'' bin=''pocc'' flags=''-pluto -pluto-tile -o %prog%-tile.c'' />
</generators>
```

Cleaning

For each generator, it is possible to set a list of files that should be rmeoved when the cleaning plan is requested. The syntax is the following :

```
<cleans>
  <clean files=''file.tmp intermed.txt'' />
</cleans>
```

Checking

After a generation step, a diff command can be invoked to check the output of some generation trace against a reference file. The syntax is the following :

```
<checks>
  <check name=''debug'' file=''%prog%-gen-debug'' />
  <check name=''log'' file=''generation.log'' />
</checks>
```

Builtin Generators

Two generators are currently defined by default.

The *none* generator does nothing and should be used when the variant source file is not generated by an external program.

The *copy* generator creates a copy of the reference program (actually a symlink).

3.1.2 Variant Description

Variants are defined within the *variants* section of the environment description. Usually, only two attributes are required : a name and the name of the generator that should be invoked to produce it.

Note: the variant name *ref* is reserved for the original version of the program.

3.1.3 Plan Description

Plans are defined within the *plans* section of the environment description. A plan generally consist of a list of variants. For each variant, it possible to specify:

- compilation flags with the *compflags* attribute ;
- wether the variant should be timed or not with the *time* attribute ;
- wether the variant's data should be checked or not with the *chk* attribute.

Builtin Plans

A plan named *clean* is defined by default. It will remove all generated and intermediate files for all variants defined in the environment.

A plan named *ref* is also defined by default. It will invoke the generators and rename the *checks* files so they are used as a comparison basis for future generations.

External Commands

Within a plan, it possible to replace or wrap some Maniac actions with external scripts. This is useful when the generated variants should be compiler and/or executed on an another computer.

External commands shall be defined in a *commands* node within a plan.

Name	Purpose
compile_variant	Invokes a compiler for a variant source file provided as an argument.
compile_loader	Invokes a compiler for a loader source file provided as an argument.
run_loader	Invokes the loader binary provided as an argument.

Table 3. External command hooks

Example :

```
<plan name=''extern_test''>
  <variants>
    <variant name=''ref'' />
    <variant name=''tiled'' gen=''pocc-tile'' />
  </variants>
  <commands>
    <command id=''run_loader'' cmd=''send-and-run-loader.sh'' />
  </commands>
</plan>
```

3.2 Program Description

The program description contains the program file name, the entry function name and its arguments, and the global variables the program accesses.

The *program* node has two attributes : a *name* and a filename extension ("c" is the default).

The *entry* node has one attribute : the *name* of the entry function for the input program.

Example :

```
<program name=''program'' ext=''c''>
  <entry name=''main'' />
</program>
```

3.2.1 Extra Source File

It is possible to add extra source files to a program using the *extrafile* node. Those are not processed by the generators, they are only used during the compilation of each variant.

There can be as many extrafile nodes as you wish.

Example :

```
<program name=''program'' ext=''c''>
  <extrafile name=''program_extra.c'' />
  </entry name=''main'' />
</program>
```

3.2.2 Data List

An entry node and a program node can contain a data node which is a list of data to initialize and/or check. Note that the data list attached to an entry node corresponds to the entry function parameters.

Name	Purpose
type	The type of the data, for example int, double, ...
name	The name of the data
ini	Indicates wether or not the data should be initialized
value	Describes the initial value of the data
chk	Indicates wether or not the data should be checked after the execution

Table 4. Data attributes

In order to describe the initial value of a data, the user can use arithmetic expressions. It is possible to access special values into those expressions :

Name	Purpose
index	A value corresponding to the index of an element in array data
rand	A random value

Table 5. Special values

Scalar

Scalar data are simple variables.

Example :

```
<scalar type=''double'' name=''var'' ini=''true'' value=''(rand+1)%76'' />
```

Array

Array data have the same attribute as other data, but require also the definition of a *dims* attribute that hold the dimensions of the array.

Example for an array A declared as int A[16][16] :

```
<array type=''int'' name=''A'' dims=''[16,16]'' />
```

Structure and Union

Structured and united data have the same attributes as other data but embed a data list which described the content of the structure.

Beware, the type attribute corresponds to the name of the structure while the name attribute refers to the name a variable which type is the structure itsef.

Example for a structure containing two integers :

```
<struct type=''point'' name=''p1''>
  <data>
    <scalar type=''int'' name=''x'' />
    <scalar type=''int'' name=''y'' />
  </data>
</struct>
```