

PathCrawler Manual

Version 2.1

CEA LIST,
Software Reliability Laboratory (LSL)
Contact: pathcrawler@saxifrage.saclay.cea.fr

April 1, 2008

This document explains how to install and use the current version of the PathCrawler¹ automatic path test generation tool developed by the LSL laboratory² of the CEA LIST.

1 Installation

We recommend the installation of PathCrawler on one of the recent versions of Ubuntu linux (for example, Ubuntu 7.10) and describe the installation process for this version in detail. Actually, you can install it on other versions of Ubuntu or other distributions of linux in a similar way, but we have not tested it for other linux distributions and do not detail the installation for them. You may need the superuser's password or rights during the installation.

To run PathCrawler, you will need to get these four programs installed on your computer:

- perl,
- ocaml (we need the Caml runtime system to run bytecode executables),
- gcc (necessary for compiling your C files),
- eclipse (the ECLiPSe Constraint Programming System, not the well-known tool platform for development in Java of the same name).

Note that PathCrawler may work faster if you install PathCrawler and ECLiPSe files in local directories rather than in network ones.

1.1 Installation of ocaml, perl, gcc

To install ocaml, perl and gcc on Ubuntu, start Synaptic package manager from the System/Administration menu, choose the packages ocaml, perl, gcc for installation (those of them which are not installed yet) and apply your changes. The programs will be installed.

After the installation, check the installed versions by typing in your shell:

¹<http://www-list.cea.fr/labos/gb/LSL/test/pathcrawler/index.html>

²<http://www-list.cea.fr/labos/gb/LSL/test/index.html>

```
gcc --version
perl --version
ocamlrun -version
```

We tested the installation with the versions 4.1.3 for gcc, v5.8.8 for perl and 3.09.2 for Caml runtime, but other recent versions may also work.

1.2 Installation of ECLiPSe

To install the ECLiPSe Constraint Programming System³, you will need to visit <http://www.eclipse-clp.org/Distribution> to download a recent distribution (choose the subdirectory of the recent version, then the subdirectory corresponding to your system platform, e.g. 5.10_118/i386_linux/). The fastest way is to install ECLiPSe from a binary package. An installation from source code may be necessary if the binary one does not work.

We describe a binary installation on a i386_linux platform (see the readme file for a different platform or a source code installation):

- create a new directory for ECLiPSe, say, ~/ECLiPSe, by typing in your shell: `mkdir ~/ECLiPSe`
- change to this directory by: `cd ~/ECLiPSe`
- download into this directory the compressed file `eclipse_basic.tgz` from the `i386_linux` subdirectory in the recent distribution directory (e.g. 5.10_118/i386_linux/) on the distribution website
- decompress it by typing: `tar -zxvf eclipse_basic.tgz`
- type: `./RUNME`
- you can now type: `~/ECLiPSe/bin/i386_linux/eclipse` to check the installation (ECLiPSe will start and print the version information, type `halt.` to exit ECLiPSe).

We tested PathCrawler with ECLiPSe version 5.10#118, but other recent versions should also work.

1.3 PathCrawler Installation

To install PathCrawler files, you may need to decompress the distribution archive by typing `tar -zxvf pathcrawler_distrib.tgz` (if you received it in compressed form) or just to find the directory `PathCrawler` on you distribution CD.

Copy this directory to the desired location. For example, if you have the directory `PathCrawler` on a CD, say, in `/media/cdrom0/PathCrawler`, and you wish to have it in `~/PathCrawler`, you can copy it by typing in your shell: `cp -r /media/cdrom0/PathCrawler ~/.`

³<http://eclipse.crosscoreop.com>

1.4 PathCrawler Distribution Contents

The directory `PathCrawler/` contains the following files:

- this manual `manual.pdf`
- the subdirectory `PathCrawler/bin/` containing (at least) the following files
 - `pathcrawler_instru`
 - `pathcrawler.eco`
 - `replace.perl`
 - `pathcrawler_test.sh`
 - `instru.sh`
 - `compil_harness.sh`
 - `generate_tests.sh`
 - `gen_fich_out_assert.sh`
- the subdirectory `PathCrawler/lib/` containing the following files
 - `lancMem.c`
 - `lancPthc.h`
 - `lancSock.c`
 - `lancStd.c`
- the subdirectory `PathCrawler/Examples/` containing some examples in subdirectories `Merge/`, `MergePrecond/`, `Sample/`, `BSort/`, `BSearch/` and `Tritype/`.

1.5 Environment Variables

You need to set and to export the environment variables `PATHCRAWLER_HOME` and `PATH` so that they can be visible and used by `PathCrawler`. It is necessary to indicate in your `PATH` variable the path to `ECLiPSe` and `PathCrawler` executables. It can be done in your shell profile which depends on your shell. After you have done this, you may need to login again, or re-source the profile before continuing. We explain this step in more detail for Ubuntu linux and `bash` shell.

You may type `bash` in your shell to see if `bash` is installed. If you use Ubuntu, but `bash` is not installed, you may install it using Synaptic package manager like you did for `ocaml`.

To set the environment variables on Ubuntu linux with `bash` shell, you can just add the following lines at the end of your `~/.bashrc` (create this profile if it does not exist):

```
PATHCRAWLER_HOME=<pathcrawler_dir>
PATH=<ECLiPSe_dir>:$PATH:$PATHCRAWLER_HOME/bin
export PATHCRAWLER_HOME PATH
```

replacing `<pathcrawler_dir>` by the directory where you installed PathCrawler (we have taken `~/PathCrawler` in Section 1.3), and `<ECLIPSe_dir>` by the directory containing ECLIPSe executables (in our installation example in Section 1.2, it was `~/ECLIPSe/bin/i386_linux`).

After you have done this, open a new shell and type `echo $PATHCRAWLER_HOME` and `echo $PATH` in your shell to check if the variables are set correctly and visible.

1.6 Installation Test

To check if the installation is finished correctly, try to generate and to print test cases for the example of the function Merge in the file `merge.c` by typing

```
cd $PATHCRAWLER_HOME
cd Examples/Merge
pathcrawler_test.sh Merge merge.c
more trace/Merge0
```

If the installation is successful, you will see the generated test cases (press `q` to exit).

2 Restrictions on C Program under Test

The user must provide the ANSI C source files of the top-level function under test and of all other functions called by the function under test. Only one source file is submitted as input to PathCrawler but it can include other files. They should not contain the function `main` (so rename it if necessary).

The current version **cannot treat**:

- dynamic memory allocation (`malloc, ...`),
- reading data from console or files (`scanf, fscanf, ...`),
- floating-point types (`float, double`),
- explicit or implicit casts other than those between integer types (e.g. pointer casts),
- assembly language code,
- pointers to functions,
- formal parameters which are functions,
- enumerated types,
- recursive structures,
- pointers of type “`void*`”,
- source files containing the function `main`,
- functions with a variable number of arguments,

- recursive functions,
- functions from a standard C library or any other function whose source code is not available may not be treated correctly.

The user must also decide on suitable ranges and other restrictions on the values of input variables to the top-level function under test: is it necessary to test the function on the full range of values allowed by the declared C type (e.g. `-2147483648,2147483647` for variables of type `int`) or is the range of values more restricted in the calling contexts in which the function will be used?

Let's note that the current version cannot treat the overflows. So, if a variable is used on the full range of its values and if this variable is used in a addition (or multiplication, ...), there would be a possible overflow. For these reasons, it's better to restrict the range of values for the input variables used in operations.

The values of different inputs can be related by some precondition, which must be respected either to prevent a run-time error (e.g. one input variable represents the size of an array given by another input variable) or for the function to calculate the right output (e.g. a function which only works if the array on input is ordered).

The user can also choose to apply the criterion "all-paths" or the criterion "all- k -paths" in which the number of loop iterations is restricted to k . The criterion is given in the `test_parameters.pl` file which contains the description of the test parameters. This file will be described below in Section 4.2. By default, the criterion is "all-paths".

3 PathCrawler: Overview

When PathCrawler constructs a set of tests to cover all feasible execution paths, it actually runs the function under test (in its *instrumented* version, i.e. slightly modified in order to output the trace of the execution path) on each test case. The user may also provide an oracle program to be run by PathCrawler after each test in order to decide the verdict (pass or fail) of the results of the test. By default, an oracle program is automatically generated with a default verdict "unknown".

Several directories containing examples of functions to test are provided with the current version of PathCrawler.

Example 1 *In order to illustrate the following explanation, we will suppose that the user is testing the Merge function defined in the `merge.c` source file of the `Examples/Merge` directory. The source code of this function is in Appendix. This function takes as its first two arguments, `t1` and `t2`, two sorted integer arrays, whose elements are merged into the array `t3` of the third argument. The fourth and fifth arguments, `l1` and `l2`, represent the number of elements in `t1` and `t2` respectively. For the examples, the current directory is `$PATHCRAWLER_HOME/Examples/Merge` containing the file `merge.c`*

Notations. We will write `<function-under-test>` for the function under test, `<source-file.c>` for the source program file under test (with extension), and `<source-file>` for the source program file without extension.

The following steps must be done in order to run PathCrawler:

1. instrumentation with automatic creation of the default test harness, default oracle and default definitions of input variable ranges and preconditions,
2. optional customisation by the user of
 - the definitions of input variable ranges and preconditions,
 - the test criterion,
 - the oracle,
3. compiling of test harness taking into account optional customisation of step 2,
4. automatic generation of a set of test-case inputs satisfying the test criterion in the file `trace/<function-under-test>0`, containing for each test case:
 - the input values,
 - the path covered,
 - the path predicate on the input variable values,
 - the symbolic values of the output variables,
 - the verdict.

These steps are described in detail in Section 4.

4 To Run PathCrawler

4.1 First Step: Instrumentation

In the first step, the user runs the automatic creation of the default test harness, the default oracle and default definitions of input variable ranges and preconditions by typing the following command:

```
instru.sh <source-file.c>
```

Example 2 *In our example, supposing the current directory contains `merge.c`:*

```
instru.sh merge.c
```

This creates a subdirectory called `pathcrawler_<source-file>/` of the directory containing `<source-file.c>`. (It will be `Merge/pathcrawler_merge` in our example.) This new subdirectory contains the following files, which may be used by the user:

- `test_parameters.pl`: an ECLiPSe source file which contains the default definitions of input variable ranges and preconditions for the each function of the source file.
- `readme.pl`: an ECLiPSe source file which loads other necessary ECLiPSe files (`pathcrawler.eco`, `blocks_cond.pl`, `blocks_assgnt.pl`, ...). This file `readme.pl` also contains the different steps to run PathCrawler in comments (cf. example 5). This file should not be edited by the user, but may be consulted to see the steps.

- `oracle_<function-under-test>.c`: default oracle program, giving the verdict “unknown” for each function of the source file.

This subdirectory also contains different internal files which should not be edited by the user.

4.2 Second Step: Optional Customisation of Test Parameters and Oracle

The second step is optional. In the current directory, the user may edit in the current directory the file `test_parameters_<function-under-test>.pl` to modify the input variable ranges and preconditions. The user may also provide the file `oracle_<function-under-test>.c` for the oracle.

4.2.1 Optional Customisation of Test Parameters

An alternative way to customize test parameters using a C precondition is described in Section 6.

Example 3 *Correct `test_parameters_Merge.pl` and `oracle_Merge.c` files are given in Appendix.*

Please note that for most examples, PathCrawler will NOT work well using the automatically generated `test_parameters.pl` file. You MUST review it and modify it if necessary before continuing to the third step. Defining variable ranges and preconditions which ensure that the function under test behaves as expected (and, in particular, does not provoke a runtime-error) cannot be automated and CAN BE VERY DIFFICULT. We apologise for the current format for the definition of preconditions, which is not user-friendly and will soon be improved. Note also that it is currently not possible to define preconditions containing a logical disjunction (or).

If you have problems defining the preconditions for your example (see the PROBLEMS section below), do contact us at the LSL laboratory.

In the current version, all variables referenced by the function under test are counted as possible input variables. These include the formal arguments of the function under test and any global variables.

To modify the `test_parameters.pl` default file in the subdirectory `pathcrawler_<source-file>`, the user has to copy it in the current directory and has to rename it as `test_parameters_<function-under-test>.pl`.

In the Merge example, the first term is `dom('Merge', cont('t1', UQ1), [], int(-2147483648, 2147483647))`, which defines the default range of values for the elements of the array `t1` which is the first formal argument of the `Merge` function. As `t1` is declared in C as an array of type `int`, the default range of elements is set to `-2147483648, 2147483647`. The user can leave this range but the generated tests are easier to read and understand if a smaller range (such as `-10, 10`) is used. It also avoids the overflow problems as explained in Section 2. Note that the term `cont('t1', UQ1)` refers to all the elements of `t1` and the term `cont('t1', i)` refers to the $i_{th} + 1$ element of `t1`.

The second term is:

```

create_input_vals('Merge',Ins):-
  var_map_new(Ins,dim('t1__Merge'),int(1,1)),
  var_map_new(Ins,dim('t2__Merge'),int(1,1)),
  var_map_new(Ins,dim('t3__Merge'),int(1,1)),
  var_map_new(Ins,'l1__Merge',int(-2147483648,2147483647)),
  var_map_new(Ins,'l2__Merge',int(-2147483648,2147483647)),
  true.

```

By default, PathCrawler sets all variable dimensions, such as that of `t1` in the previous example, to the interval `[1, 1]`, containing the single value 1. The user could want to change this range to, for example, `[0, 10]`.

The array `t3` in this example is in fact an output array. The current version of PathCrawler cannot determine that `t3` is an output array and will provide test input values for this array which will be overwritten by `Merge`. The range of values of the elements of `t3` is therefore irrelevant, but it is important that the maximum dimension of `t3` is set to be greater than or equal to the sum of the maximum dimensions of `t1` and `t2`.

Then, we have two lists of preconditions, the first of which is the list of “unquantified” preconditions. In the case of the `Merge` function, this list must contain three preconditions to define the relation between the dimensions of arrays `t1`, `t2` and `t3` and the arguments `l1` and `l2`. In the syntax used in the current version, these preconditions are as follows:

```

unquantif_preconds('Merge',
  [cond(supegal,dim('t1__Merge'),'l1',pre),
   cond(supegal,dim('t2__Merge'),'l2',pre),
   cond(supegal,dim('t3__Merge'),+( 'l1', 'l2' ),pre)]).

```

The second list contains preconditions in which an array index is replaced by a variable so that the precondition applies to all elements of the relevant array whose index satisfies a particular condition. In the case of `Merge`, two preconditions of this sort are needed to specify that the arrays `t1` and `t2` must be ordered, i.e. that each element (other than the first element of the array) is greater or equal to the preceding element. In the syntax used in the current version, these preconditions are as follows:

```

quantif_preconds('Merge',[uq_cond([UQV3],
  [cond(supegal,UQV3,1,pre)],
  supegal,
  cont('t1__Merge',UQV3),
  cont('t1__Merge',UQV3 - 1)),
  uq_cond([UQV4],
  [cond(supegal,UQV4,1,pre)],
  supegal,
  cont('t2__Merge',UQV4),
  cont('t2__Merge',UQV4 - 1)]]).

```

The first argument of a “quantified” precondition is the list of the variables representing array index values. The next argument is a list of conditions to be satisfied by these indices in order for the precondition to be applied. The following arguments contain the precondition on these array elements and/or other input variables.

Finally, the test criterion is defined. The current version of PathCrawler generates tests satisfying two alternative criteria:

- 100% feasible execution paths (the default value):
`strategy('<function-under-test>',off)`.
- 100% feasible k -paths (i.e. paths containing no more than k iterations of each loop with a variable number of iterations):
`strategy('<function-under-test>',on(k))`.

The Merge directory contains the version of the `test_parameters.pl` file obtained after all the above modifications have been made by the user and the test criterion set to `on(2)`. This file is shown in Appendix at Section 7.2.

4.2.2 Optional Customisation of Oracle

The user may define a file with an oracle function. It must have the same interface as the default `oracle_<function-under-test>.c` generated by Step 1, so the easiest way is to copy the default oracle file `pathcrawler_<source-file>/oracle_<function-under-test>.c` to `./test_parameters_<function-under-test>.pl` and to modify it. The Merge directory contains a version of `oracle_Merge.c` file with a definition of the oracle function which evaluates the results of the Merge function and provides a success or fail verdict. This file illustrates the syntax to be respected by user-defined oracles (cf. Section 7.3 in Appendix).

The output of the function under test may overwrite the input in some cases (for example, array sorting algorithms will usually overwrite the initial array). The oracle function is called after the function under test, when the input may be already overwritten. Since the oracle have to examine both input and output data of the function under test, additional parameters are added to the oracle function interface. The parameters with `Pre_` prefix will contain (recursive) copies of the input data provided in the function call. They are not accessible from the function under test, so do not modified. The parameters without prefix are exactly the variables which were provided while calling the function under test, so they may contain the output values at the moment when the oracle function is called. Although this duplication of parameters is systematic, this difference is only important for pointers and arrays (or structures containing them). The importance of the `Pre_` parameters is clear from the `Bsort` example (file `oracle_bsort.c`).

4.3 Third Step: Compiling Test Harness

During the third step, the user defines the function under test and runs the automatic compilation of the test harness by typing the following command:

```
compil_harness.sh <function-under-test> <source-file.c>
```

Example 4 *In our example, supposing the current directory contains `merge.c`:*

```
compil_harness.sh Merge merge.c
```

Note that if the user has defined a function `oracle_<function-under-test>.c` and a file `test_parameters_<function-under-test>.pl` (or `test_parameters.pl`) in the current directory, this step copies these files into `pathcrawler_<source-file>` subdirectory and overwrites the default files generated during the first step.

Example 5 *The `readme.pl` file (generated during step 1):*

```
% To run PathCrawler, please do the following four steps:
% 1) instru.sh merge.c
% 2) Copy the file pathcrawler_merge/test_parameters.pl to the
% current directory and edit the preconditions and variable domains
% if necessary. If different functions in the file, rename the file to
% test_parameters_<function_under_test>.pl in the current directory.
% 3) compil_harness.sh <function_under_test> merge.c
% 4) generate_tests.sh <function_under_test> merge.c

:- use_module('$PATHCRAWLER_HOME/bin/pathcrawler').
:- [blocks_assgnt].
:- [blocks_cond].
:- [fixed_domain].
:- [dico].
:- [simul].
:- [test_parameters].

:- heading:setval_file_name('merge.c').
:- heading:setval_pathcrawler_directory('pathcrawler_merge').
```

4.4 The Fourth Step: Generation of Test-Case Inputs

This last step consists of the generation of test-case inputs satisfying the test criterion. The simplest way is to type in a shell this command:

```
generate-tests.sh <function-under-test> <source-file.c>
```

Example 6 *For the function `Merge`, type:*

```
generate-tests.sh Merge merge.c
```

A file called `<function-under-test>0` which contains all the information on the test set is created in a new subdirectory of the current directory called `trace/` (in our example, it is the file `Merge/trace/Merge0`). Note that if you run PathCrawler several times, it may create different test cases for the same path, and the order of paths may also be different.

Example 7 *Extract from the file `trace/Merge0`:*

```
TEST CASE 5
```

```
Dimensions:
t1 = 2
t2 = 0
t3 = 2
```

Other input values:

```
t1[1] = -16
l2 = 0
t1[0] = -19
l1 = 2
```

Result:

success

Path Covered:

```
merge.c
+7 -7b +18x2 -18 -23
```

Path Predicate:

```
merge.c: +7 0<l1
merge.c: -7b 0>=l2 (exit loop line 7)
merge.c: +18 1<l1 (iteration of loop line 18)
merge.c: -18 2>=l1 (exit loop line 18 after 2 iterations)
merge.c: -23 0>=0 (exit loop line 23)
```

Out Values:

```
t3[0] = t1[0]
t3[1] = t1[1]
```

negation deepest unexplored condition violates the iteration limit,
Path Predicate Prefix to solve:

```
merge.c: +7 0<l1
merge.c: +7b 0<l2
```

The first three sections here show the dimensions and values of input variables and the test verdict. In `Path Covered` and `Path Predicate` sections, +N (resp. -N) means that the (first) condition at line N is verified (resp. not verified) in the current path. So, +7 means that the condition at line 7 of file `merge.c` is verified and -23 means that the condition at line 23 is not verified. Since line 7 of the source code of `merge.c` (see Appendix) is `while (i < l1 && j < l2)`, this line contains actually two sub-conditions. Here +7 exactly means that the first condition `i<l1` is verified and -7b means that the second condition `j<l2` is not verified. In the same way, +Nc or -Nc would represent the third condition of line N and so on. We also use abbreviated notation for loops: +18x2 replaces +18 +18 and means that the condition at line 18 is verified twice, so two iterations of the loop are executed. In `Out Values` section, there are two output variables `t3[0]` and `t3[1]` which respectively take the values of the input variables `t1[0]` and `t1[1]`. `Path Predicate Prefix to solve` shows the path predicate prefix which PathCrawler will try to execute during the next test case generation.

4.5 PathCrawler Usage Outline

Suppose that the current directory contains the function under test. The different steps to run PathCrawler are:

1. `instru.sh <source-file.c>`
2. Then the user can create in the current directory `test_parameters.pl` and `oracle_<function-under-test>.c` as explained in Section 4.2
3. `compil_harness.sh <function-under-test> <source-file.c>`
4. `generate-tests.sh <function-under-test> <source-file.c>`

These steps are given in comments in the `readme.pl` file in the subdirectory `pathcrawler_<source-file>` (cf. example 5) generated by the first step.

If you do not need to do Step 2 (for example, if you do not need, or have already `test_parameters.pl` and `oracle_<function-under-test>.c` files in the current directory), you may use a shortcut to run Steps 1,3,4 directly:

```
pathcrawler-test.sh <function-under-test> <source-file.c>
```

Example 8 *In our example, just run*

```
pathcrawler-test.sh Merge merge.c
```

5 Problems

Let us describe the most likely causes of errors during the test generation.

- An error is a `test_parameters.pl` file containing input ranges or preconditions which are unsuited to the program. You can check for this by looking at the trace file which was generated before the error. Do the test inputs seem to be reasonable values? For example, for the Merge function, do the last two parameters contain values smaller or equal to the dimensions of the first two parameters. Are the elements of the first two parameters ordered?
- Another cause of error is the use of the full range of integer values for integers which are added or multiplied by the function under test. When the function is executed, an integer overflow may occur, but the PathCrawler constraint solving does not yet take overflows into account. The solution is to define smaller input ranges.
- If the program “hangs”, i.e. does not crash but appears to stop writing to the trace file then it is probably trying to solve a particularly intractable set of constraints. This is usually the result of non-linear constraints arising from e.g. bit operations in the source code or multiplications of two variables in the source code or in a precondition.
- Other problems: please report the bug.

6 Easy Customization: Use of a C Precondition

You can generate test cases for your function without any knowledge of the underlying Prolog configuration. Indeed, PathCrawler accepts a C function to express the preconditions of the function under test. To increase C expressivity, it provides a function named `pathcrawler_dimension` which gives the number of elements in input arrays.

The first subsection explains how to write such a function and how to use it. In the second subsection, we describe a modification of `test_parameters.pl` needed if and only if the function under test takes an array as a parameter.

The subdirectory `Examples/MergePrecond` contains another version of the Merge example (subdirectory `Examples/Merge`) which uses a C precondition.

6.1 Writing and Using Precondions

To use this feature, you must first write a function which could state whether a given set of inputs for the function under test is valid or invalid. It will satisfy the following rules:

- it is called `<function-under-test>_precond` if the function under test is called `<function-under-test>`;
- is is written (included) directly in the source file of the function under test;
- it takes the same parameters as the function under test (both name and type are important);
- it returns a nonzero value if the input respects the preconditions, and zero otherwise;
- it has no side effects whatsoever.

Example 9 *In the Merge example, we could write the following function (see the complete file `Examples/MergePrecond/merge.c`):*

```
int Merge_precond(int t1[], int t2[], int t3[], int l1, int l2) {
    if (l1 > pathcrawler_dimension(t1)
        || l2 > pathcrawler_dimension(t2)
        || l1+l2 > pathcrawler_dimension(t3)) {
        return 0;
    }
    int i;
    for (i=1; i < l1; i++) {
        if (t1[i-1] > t1[i]) {
            return 0;
        }
    }
    for (i=1; i < l2; i++) {
        if (t2[i-1] > t2[i]) {
            return 0;
        }
    }
}
```

```

}
return 1;
}

```

After you have written the precondition function, it will be automatically detected and used during the normal process:

1. `instru.sh <source-file.c>`
2. Optionally create a file `oracle_<function-under-test>.c`. If your function have array parameters, copy the default test parameters file `pathcrawler_<source-file>/test_parameters.pl` to `./test_parameters_<function-under-test>.pl` and modify it.
3. `compil_harness.sh <function-under-test> <source-file.c>`
4. `generate-tests.sh <function-under-test> <source-file.c>`

If you do not need to do Step 2 (for example, if you do not need, or have already `test_parameters.pl` and `oracle_<function-under-test>.c` files in the current directory), you may use a shortcut to run Steps 1,3,4 directly:

```
pathcrawler-test.sh <function-under-test> <source-file.c>
```

6.2 Array Parameters

The array parameters need a special configuration in `test_parameters.pl`. Indeed, in C (ANSI-C, C90 and even C99), the three following prototypes are equivalent:

```

int f(int* a);
int f(int[] a);
int f(int[10] a);

```

That is why our static analysis can not determine if this kind of parameter points to one value (a reference) or several values (an array). The former interpretation being the default choice, you must explicitly set the bounds of the array size in `test_parameters.pl` to obtain the latter behavior. In this case, we recommend to use the bounds of `unsigned int`, i.e. 0,4294967295.

Example 10 *For `merge.c`, we need to modify only three lines of the default `test_parameters.pl` in the following way:*

```

[...]
create_input_vals('Merge',Ins):-
  var_map_new(Ins,'l2__Merge',int(-2147483648,2147483647)),
  var_map_new(Ins,'l1__Merge',int(-2147483648,2147483647)),
  var_map_new(Ins,dim('t3__Merge'),int(0,4294967295)),%was int(1,1)
  var_map_new(Ins,dim('t2__Merge'),int(0,4294967295)),%was int(1,1)
  var_map_new(Ins,dim('t1__Merge'),int(0,4294967295)),%was int(1,1)
  true.
[...]

```

See the complete file in `Examples/MergePrecond/test_parameters_Merge.pl`.

7 Appendix

7.1 Source Code of merge.c

```
void Merge (int t1[], int t2[], int t3[], int l1, int l2) {

    int i = 0;
    int j = 0;
    int k = 0;

    while (i < l1 && j < l2) {
        if (t1[i] < t2[j]) {
            t3[k] = t1[i];
            i++;
        }
        else {
            t3[k] = t2[j];
            j++;
        }
        k++;
    }
    while (i < l1) {
        t3[k] = t1[i];
        i++;
        k++;
    }
    while (j < l2) {
        t3[k] = t2[j];
        j++;
        k++;
    }
}
```

7.2 A Correct test_parameters_Merge.pl File

```
:- module(test_parameters).

:- import var_map_new/3 from var_map.

:- export dom/4.
:- export create_input_vals/2.
:- export unquantif_preconds/2.
:- export quantif_preconds/2.
:- export strategy/2.
:- export precondition_of/2.

%domains of elements (or fields) of structural input variables
dom('Merge',cont('t1__Merge',UQ1),[],int(-10,10)).
dom('Merge',cont('t2__Merge',UQ1),[],int(-10,10)).

%dimensions of structural input variables
%and domains of scalar input variables
create_input_vals('Merge',Ins):-
    var_map_new(Ins,dim('t1__Merge'),int(0,10)),
    var_map_new(Ins,dim('t2__Merge'),int(0,10)),
    var_map_new(Ins,dim('t3__Merge'),int(0,20)),
    var_map_new(Ins,'l1__Merge',int(0,10)),
    var_map_new(Ins,'l2__Merge',int(0,10)),
    true.

%unquantifiable preconditions
unquantif_preconds('Merge',[cond(supegal,dim('t1__Merge'),'l1__Merge',pre),
    cond(supegal,dim('t2__Merge'),'l2__Merge',pre),
    cond(supegal,dim('t3__Merge'),+( 'l1__Merge','l2__Merge'),pre)]).

%quantifiable preconditions
quantif_preconds('Merge',[uq_cond([UQV3],
    [cond(supegal,UQV3,1,pre)],
    supegal,
    cont('t1__Merge',UQV3),
    cont('t1__Merge',UQV3 - 1)),
    uq_cond([UQV4],
    [cond(supegal,UQV4,1,pre)],
    supegal,
    cont('t2__Merge',UQV4),
    cont('t2__Merge',UQV4 - 1)])].

%test criterion
strategy('Merge',on(2)).

precondition_of(0,0).
```

7.3 Source Code of oracle_Merge.c

```
void oracle_Merge(
    int Pre_t1[], int t1[],
    int Pre_t2[], int t2[],
    int Pre_t3[], int t3[],
    int Pre_l1, int l1,
    int Pre_l2, int l2)
{
    int i, j, n1, n2, n3;
    int l3 = l1 + l2;
    int l3moins1 = l3 - 1;

    for (i = 0; i < l3moins1; i++) {
        if (t3[i] > t3[i+1]) {
            fprintf(pathcrawler_out_stream,
                "failure(\`t3[%d] > t3[%d]\`). ", i, i+1);
            return;
        }
    }
    i = 0;
    while (i < l3) {
        /* number of occurrences of this element in t3 */
        n3 = 1;
        while (i < l3moins1 && t3[i + 1] == t3[i]) {
            i++;
            n3++;
        }
        /* number of occurrences of this element in t1 */
        n1 = 0;
        for (j = 0; j < l1; j++) {
            if (t1[j] == t3[i])
                n1++;
        }
        /* number of occurrences of this element in t2 */
        n2 = 0;
        for (j = 0; j < l2; j++) {
            if (t2[j] == t3[i])
                n2++;
        }
        /* compare */
        if (n3 != (n1 + n2)) {
            fprintf(pathcrawler_out_stream,
                "failure(\`%d occurrences %d in t3\`). ", n3, t3[i]);
            return;
        }
        i++;
    }
    fprintf(pathcrawler_out_stream, "success. ");
}
```