Automatic Test Generation Tool For C Programs

User Manual

Version 3.1

Contact: pathcrawler@cea.fr

DIRECTION DE LA RECHERCHE TECHNOLOGIQUE
LABORATOIRE D'INTEGRATION DES SYSTEMES ET DES TECHNOLOGIES
DEPARTEMENT INGENIERIE LOGICIELS ET SYSTEMES

CEA SACLAY, 91191 GIF-SUR-YVETTE CEDEX FRANCE
TEL: 01 69 08 33 53 – FAX: 01 69 08 83 95

Table of content

CHAPTER 1 INTRODUCTION

This is PathCrawler's user manual. PathCrawler[1] is an automatic path test generation tool developed by the LSL laboratory of the CEA LIST.

This manual gives an overview of PathCrawler for newcomers, and serves as a reference for experienced users.

1 http://www.pathcrawler-online.com

Chapter 2 Getting Started

This chapter describes how to install PathCrawler and what this installation provides.

2.1 Installation

The PathCrawler tool is distributed as binaries available for Linux OS. We recommend the installation of PathCrawler on one of the recent versions of Ubuntu Linux (for example, Ubuntu 10.04) 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.

PathCrawler is a Frama-C plugin, wich is a platform of C program analysis.

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 byte code executables),
- gcc (necessary for compiling your C files),
- ECLiPSe (the ECLiPSe Constraint Programming System, not the well-known tool platform).
- Frama-C

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

2.1.1 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:

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

We tested the installation with the versions 4.4.5 for gcc, 5.10.1 for Perl, 3.11.2 for Caml runtime and 5.10 147 for ECLiPSe, but other recent versions may also work.

2.1.2 Installation of ECLiPSe, version 5.10

To install the ECLiPSe Constraint Programming System, you will need to visit http://www.eclipseclp.org/.

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):

The distribution to download correspond to version 5.10, PathCrawler has not yet been ported to 6.0 (http://eclipseclp.org/Distribution/Old/5.10_147/i386_linux/).

- 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 files **eclipse_basic.tgz** and **eclipse_misc.tgz** from the i386_linux subdirectory in the distribution directory (e.g. 5.10_147/i386_linux/) on the distribution website
- decompress them by typing:

```
tar -zxvf eclipse basic.tgz
tar -zxvf eclipse misc.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_147, but other distributions of the version 5.10 should also work. We have not tested the version 6.0.

2.1.3 Frama-C Installation

To install Frama-C, you need to visit http://frama-c.com

All the explanations are given, the user manual is also available.

2.1.4 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 your 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 ~/
```

2.1.5 PathCrawler Distribution Contents

The directory PathCrawler/ contains the following files:

- this manual manual.pdf
- the subdirectory bin, lib, xml and Examples

2.1.6 Environment Variables

You need to set and to export the environment variables **PATHCRAWLER_HOME**, **ECLIPSE_CLP_HOME** and **PATH** so that they can be visible and used by PathCrawler. It may be also convenient 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>
ECLIPSE_CLP_HOME=<ECLiPSe_dir>
PATH=$PATH:$PATHCRAWLER_HOME/bin:$ECLIPSE_CLP_HOME
export PATHCRAWLER_HOME ECLIPSE_CLP_HOME PATH
```

replacing <pathcrawler_dir> by the directory where you installed PathCrawler, and <ECLiPSe_dir> by the directory containing ECLiPSe executables (e.g <where-you-want>/ECLiPSe/bin/i386_linux).
After you have done this, open a new shell and type:

```
echo $PATHCRAWLER_HOME
echo $ECLIPSE_CLP_HOME
echo $PATH
```

in your shell to check if the variables are set correctly and visible.

2.1.7 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
frama-c -pc -pc-kpath 2 -main Merge merge.c
ls testcases_merge/Merge/html/TC*
```

If the installation is successful, you will see the generated test cases.

CHAPTER 3 PATHCRAWLER BY EXAMPLE

When PathCrawler constructs a set of tests to cover all feasible execution paths, it actually runs the function under test (in its instrumented version, e.g. slightly modified in order to output the trace of the execution path) on each test case.

The user may 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.

In this Chapter, 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.

For more general explanations, see chapter 4.

3.1 The Merge Function

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. Here is the source code of the function :

```
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++;
    }
}
```

The following steps must be performed 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 customization 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 customization of step 2,
4. Automatic generation of a set of test cases satisfying the test criterion.

We will detailed in the rest of this chapter, step by step, how to generate test cases for the Merge function.

Note that, throughout the chapter, we consider that the current directory is Merge/.

3.2 *Instrumentation*

In this step, the source code will be instrumented by adding some trace instructions. These trace instructions are necessary for the "concolic" execution of the program. This mean that the instrumented program will be really executed, but also it will be symbolically executed.

To perform this step, the user must type (in the Merge/ directory):

```
frama-c -pc -pc-analyzer -main Merge merge.c
```

Note that the instrumented program will also be transformed in order to normalize the source code, and this normalization produces an equivalent program.

After this step, the two important files (created in the generated directory pathcrawler_merge/) is :

– the file test_parameters.pl which contains the default ranges of the input variables, the default strategy used for generation of test cases, etc...

– the instrumented file used also for the generation of test cases (but the user must not modify or edit it)

3.3 Customization

In the content of the present distribution of PathCrawler, we have in the Merge/ directory a predefined test_parameters_merge.pl. But suppose we don't have this file. For the Merge example, the generated test parameters don't suffice. So we need to customize the test parameters, for example to set that l1 represents the length of the array t1.

We have to edit the file pathcrawler_merge/test_parameters.pl.

Let us examine a part of the test_parameters.pl automatically generated by the previous step.

```
% Range for element of arrays or pointers
% cont('t',_) is the equivalent of : t[i] for all i
dom('Merge',cont('t1',_),[],int([-2147483648..2147483647])).
dom('Merge',cont('t2',_),[],int([-2147483648..2147483647])).
dom('Merge',cont('t3',_),[],int([-2147483648..2147483647])).

% Range for variables of scalar type
% Note that dim('t') is the length of t if t is an array,
% or it is the dimension of t if t is a pointer
create_input_vals('Merge',Ins):-
  create_input_val(dim('t1'),int([0..1]),Ins),
  create_input_val(dim('t2'),int([0..1]),Ins),
  create_input_val(dim('t3'),int([0..1]),Ins),
  create_input_val('l1',int([-2147483648..2147483647]),Ins),
  create_input_val('l2',int([-2147483648..2147483647]),Ins),
  true.

% Preconditions on the input variables
% By default, there is no precondition
unquantif_preconds('Merge',[]).
quantif_preconds('Merge',[]).
% C preconditions
precondition_of(0,0).
```

We have to modify different things in order to perform the tests for the function Merge. For example, if we don't restrict the **length of arrays**, the numbers of paths for the function will be enormous. So we can set:

```
create_input_vals('Merge',Ins):-
  create_input_val(dim('t1'),int([0..10]),Ins),
  create_input_val(dim('t2'),int([0..10]),Ins),
  create_input_val(dim('t3'),int([0..20]),Ins),
  create_input_val('l1',int([0..10]),Ins),
  create_input_val('l2',int([0..10]),Ins),
  true.
```

We can also **restrict the range** of array elements:

```
dom('Merge',cont('t1',_),[],int([-10..10])).
dom('Merge',cont('t2',_),[],int([-10..10])).
```

As we said before, we have to say that the lengths of t1 and t2 are represented by l1 and l2 respectively, because we can't automatically deduce that:

```
unquantif_preconds('Merge',
  [cond(supegal,dim('t1'),'l1',pre),          % length of t1 >= l1
   cond(supegal,dim('t2'),'l2',pre),          % length of t2 >= l2
   cond(supegal,dim('t3'),                    % length of t3 >= l1+l2
             +('l1','l2'),pre)]).
```

We have seen that Merge take as input the two sorted arrays t1 and t2. We must set this as a precondition:

```
quantif_preconds('Merge',[
  uq_cond([I],[cond(supegal,I,1,pre)],        % For I such that I >= 1, we have
           supegal,
           cont('t1',I),                        % t1[I] >= t1[I]
           cont('t1',I - 1)),
  uq_cond([J],[cond(supegal,J,1,pre)],        % For J such that J >= 1, we have
            supegal,
            cont('t2',J),                       % t2[J] >= t1[J]
            cont('t2',J - 1))]).
```

We give a complete example of a correct test_parameters.pl for the function Merge in section 7.2.

3.4 Generating and examining test cases

After analyzing, the user can generate test cases for the function Merge, by typing:

```
frama-c -pc -pc-generator -pc-k-path 2 -main Merge merge.c
```

Note that, in general case, if the user doesn't have to customize the test parameters or if he/she has already defined a test parameters file, he/she can, instead of instrumenting, customizing, compiling and generating, directly type:

```
frama-c -pc -pc-k-path Merge merge.c
```

The result, for the Merge function, is at least 19 test cases corresponding to the 19 feasible paths that we can see in the control flow graph of the function.

In this case, the -pc-k-path parameter define a **limit for the loop iterations** (see section 4.2.1 and 5.4).

The test cases are written in several forms, a C form, a text form, and a HTML form:

- First, a test driver for each test case is created in a separate C file:
  testcases_merge/Merge/testdrivers/test_Merge<testcase_num>.c
  It executes the function on the corresponding test case (cf Example 8).

- Secondly, a HTML file is generated and the user can navigate to see the different test cases. This file is available in:
  testcases_merge/Merge/html/index.html

- Third, PathCrawler writes a detailed trace file trace/<function> (cf Example 7) 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 oracle's verdict.

These steps are described in detail for the general case in Chapter 4.

CHAPTER 4 TO RUN PATHCRAWLER

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

4.1 *First Step: Instrumentation*

In this step, we statically analyze the source code to prepare test generation.
In particular, this analysis extracts the set of input variables for the function under test.
We will describe more precisely this set of variables, and then we will describe the instrumentation step in more detail.

4.1.1 The Set of Input Variables

The set of input variables produced by the code analyzer is in fact an over approximation such that :

- All the formal parameters of the function under test are input variables
- All the global variables used in all the functions are considered as input variables
- All the global variables declared “const” are not input variables, and are supposed constant during the execution of the function (even though we known that the C language does allow these variables to be changed)
- All the global variables that are initialized in code outside the body of the function under test are not input variables if and only if the option –pca-keep-init-values is set. Otherwise, those variables are considered as input variables and the initialization values are forgotten.

The test cases generated in the final stage are composed of values for the set of input variables as described above.

4.1.2 Instrumentation

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

```
frama-c -pc -pc-analyzer -main <function> <file.c>
```

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

```
frama-c -pc -pc-analyzer -main Merge merge.c
```

This creates a subdirectory called pathcrawler_<file>/ of the directory containing <file.c>. (It will be Merge/pathcrawler_merge/ in our example.)

This new subdirectory contains among others the following files which may be modified 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.
- oracle_<function>.c: default oracle program, giving the verdict “unknown” for each function of the source file.

If the C function under test may read the values of global variables which are initialized in the instrumented file, the user must decide whether these variables may change value between initialization and any call to the function under test. If not, the user can ask PathCrawler to treat such variables as constants, and not create test-cases in which they have other values, by setting the option

```
frama-c -pc -pc-analyzer -pc-keep-init-values -main <function> <file.c>
```

General utilization :
Usage : **frama-c -pc -pc-analyzer -main <functionName> <fileName.c> [OPTIONS]**

```
*** PATHCRAWLER ANALYZER

-pc-graphs              construct call graph and CFG (default: no)
-pc-keep-init-values    consider initialized global variables as constants
                        (default: no)
-pc-ptr-0-1             domain of input pointers dimension is [0..1] (set by
                        default)
-pc-ptr-0-max           domain of input pointers dimension is [0..MaxInt]
-pc-ptr-1-1             domain of input pointers dimension is 1
-pc-ptr-1-max           domain of input pointers dimension is [1..MaxInt]
```

4.2 Second Step: Optional Customization of Test Parameters and Oracle

The second step is optional. The user may copy the file test_parameters_<function>.pl into the current directory and then edit it in order to modify the input variable ranges and preconditions.

The user may also provide the file oracle_<function>.c for the oracle.

4.2.1 Optional Customization of Test Parameters

An alternative way to customize test parameters using a C precondition is described in section 5.1.

Example 3
Correct test parameters Merge.pl and oracle_Merge.c files are given in the 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 apologize 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 section 4.5), 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_<file>, the user has to copy it into the current directory and has to rename it as test_parameters_<function>.pl.

In the Merge example, the first term is

```
dom('Merge',cont('t1',_),[],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 integer overflow problems as explained in sections 4.5.

A domain may be also defined as a union of intervals or values, for example,

```
dom('Merge', cont('t1',_),[], int([(-100)..(-50),-25,-15..(-5),0..10]))
```

A negative upper bound of an interval must be put into brackets.

The term cont('t1',_) refers to all the elements of t1 and the term cont('t1',i) refers to the ith+1 element of t1.

The second term is:

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

By default, PathCrawler sets all variable dimensions, such as that of t1 in the previous example, to the interval 0..1, containing the values 0 and 1.

The user could wish 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 Merge directory contains the version of the test_parameters.pl file obtained after all the above modifications have been made by the user.

This file is shown in the Appendix at section 7.2.

4.2.2 Optional Customization of Oracle

The user may define a file with an oracle function. It must have the same interface as the default oracle_<function>.c generated by Step 1, so the easiest way is to copy the default oracle file **pathcrawler_<file>/oracle_<function>.c**
to **./oracle_<function>.c**
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 the 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 has 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 modify.

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 The Third 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:

```
frama-c -pc -pc-generator -main <function> <file.c>
```

Example 6
For the function Merge, type:

```
frama-c -pc -pc-generator -pc-k-path -main Merge merge.c
```

A file called <function> 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/Merge). 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/Merge:

```
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
```

Besides, separate test drivers allowing to execute the function under test on one test case are generated in the subdirectory testcases_merge.c/test_Merge/.

Example 8 Example of a test driver

```
(file  testcases_merge.c/test_Merge/test_Merge10.c):

 /************ TEST CASE 10 ************/
#include "../../merge.c"
int main()
{
//declarations
int *t1__Merge;
int *t2__Merge;
int *t3__Merge;
int l1__Merge;
int l2__Merge;
//allocations
t1__Merge = (int*) malloc(2* sizeof(int));
t2__Merge = (int*) malloc(1* sizeof(int));
t3__Merge = (int*) malloc(3* sizeof(int));
//assignments
l1__Merge = 2;
l2__Merge = 1;
t1__Merge[0] = -10;
t1__Merge[1] = 7;
t2__Merge[0] = 9;
//function call
Merge(t1__Merge, t2__Merge, t3__Merge, l1__Merge, l2__Merge);
//desallocation
free(t3__Merge);
free(t2__Merge);
free(t1__Merge);
//end
}
```

The first three sections here show the dimensions and values of input variables and the test verdict.

In the 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 the 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.

4.4 PathCrawler Usage Outline

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

1. `frama-c -pc -pc-analyzer -main <function> <file.c> [OPTIONS]`
2. Then the user can create `test_parameters.pl` and `oracle_<function>.c` in the current directory as explained in section 4.2
3. `frama-c -pc -pc-generator -main <function> <file.c> [OPTIONS]`

These steps are given in comments in the readme.pl file in the subdirectory pathcrawler_<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>.c files in the current directory), you may use a shortcut to run Steps 1,3,4 directly:

```
frama-c -pc -main <function> <file.c>
```

Example 9 In our example, just run

```
frama-c -pc -pc-k-path -main Merge merge.c
```

4.5 Causes of Errors

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.

CHAPTER 5 GENERAL SERVICES

5.1 *Easy Customization: Use of a C Precondition*

You can generate test cases for your function without any knowledge of the syntax described in Section 4.2.1.

Indeed, PathCrawler accepts a C function to express the preconditions of the function under test. To increase C expressiveness, it provides a function named pathcrawler_dimension which gives the number of elements in input arrays.

Section 5.1.1 explains how to write such a function and how to use it. In the section 5.1.2, 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.

5.1.1 Writing and Using Preconditions

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>_precond if the function under test is called <function>
- 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 10

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. frama-c -pc -pc-analyzer -main <function> <file.c> [OPTIONS]
2. Optionally create a file  oracle_<function>.c.
3. If your function has array parameters, copy the default test parameters
   file pathcrawler_<file>/test_parameters.pl to
   ./test_parameters_<function>.pl and modify it.
4. frama-c -pc -pc-generator -main <function> <file.c> [OPTIONS]
```

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>.c files in the current directory), you may use a shortcut to run Steps 1,3,4 directly:

```
frama-c -pc -main <function> <file.c> [OPTIONS]
```

5.1.2 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.

Example 11
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):-
        create_input_val('I2__Merge',
                int([-2147483648..2147483647]),Ins),
        create_input_val('I1__Merge',
                int([-2147483648..2147483647]),Ins),
        create_input_val(dim('t3__Merge'),
                int([0..4294967295]),Ins),
        create_input_val(dim('t2__Merge'),
                int([0..4294967295]),Ins),
        create_input_val(dim('t1__Merge'),
                int([0..4294967295]),Ins),
        true.
        [...]
```

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

5.2 Restricting Program Paths to be Covered

Some users may be interested in covering only paths with a particular property in a particular location, or only some parts of the program. So, to discharge the user from waiting for all paths to be covered, PathCrawler provides a mechanism to select program paths which verify a given condition at certain location when this location belongs to the path.

There is no restriction for the paths which do not pass through this location.

In order to inject assumes into the function under test, put the desired pathcrawler_assume(condition).

Here is a quick example to illustrate this aspect:

Example 12
In the following program, we assume that the given condition should be verified
before evaluating the if statement. Therefore the generator will try to cover all paths respecting this condition.

```
        /* ... */
        pathcrawler_assume(x == y);
        if(x == y ) {
                /* ... */
                /* paths to be covered */
        }
        else {
                /* ... */
                /* paths to be ignored */
        }
        /* ... */
```

This feature may be seen as another way to write a precondition. However, unlike the usual precondition (described in sections 4.2.1 and 5.1), pathcrawler_assume may be used at any location in the program under test.

Sometimes, despite the presence of pathcrawler_assume at some location, while trying to generate a test for a shorter path prefix, PathCrawler may happen to generate a test case violating the assume condition. In this case the execution of the test is stopped by the pathcrawler_assume instruction and assume_violated(line_number) is returned as the oracle’s verdict.

The path prefix is then extended to contain the assume condition to ensure this condition at this location for the following tests with this path prefix.

5.3 Assertions

The macro pathcrawler_assert(condition) may be used to inject an assertion into the program under test. This feature may be seen as another way to write an oracle.

However, unlike for the usual oracle (described in section 4.2.2) pathcrawler_assert(condition) may be used at any location in the program under test, and will force the tool to generate test cases to cover both the case condition is true, and condition is false.

In the last case, the execution of the test is stopped by the pathcrawler_assert instruction and assert_violated(line_number) is returned as the oracle’s verdict.

In order to inject assertions into the function under test put the desired pathcrawler_assert(condition) at the required location.

5.4 k-path strategy

The user can choose to apply the criterion “all-paths”, that is the default, or the criterion “all-k-paths”.

With this last criterion, the number of loop iterations is restricted to k, for all the loops with a variable number of iterations in the function under test. As we see for the Merge example, the criterion is given by the **-pc-k-path <k>** option.

CHAPTER 6 RESTRICTIONS ON THE 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. If they contain the function main, it will be renamed by PathCrawler.

The current version cannot treat:

- reading data from console or files (scanf, fscanf, ...),
- the floating-point type float. Double is treated (but path conditions using results of trigonometric functions may considerably slow down constraint resolution)
- explicit or implicit casts other than those between integer and floating point types (e.g. pointer casts),
- assembly language code,
- pointers to functions as input of the function to be tested (but other pointers to functions are treated)
- formal parameters which are functions,
- recursive structures,
- pointers of type void* as input of the function to be tested,
- 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.
- some conversions of the type of integer constants: we can add the letter L (or l) at the end of the constant to ensure that the type long is used, and we can add U (or u) to specify the attribute unsigned. But we cannot write a positive constant greater than the capacity of unsigned long int type or a negative constant not representable in long int type because then the meaning is unclear and PathCrawler cannot guarantee good behavior.

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 generate tests to reproduce the results of calculations provoking overflows. 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 this reason, it is better to restrict the range of values for the input variables used in arithmetic 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. By default, the criterion is “all-paths”.

Remark – C test drivers

For the moment, some restrictions appear on the use of the test drivers:

- First, the user can't use the pathcrawler primitive pathcrawler_dimension in the source code.
- Second, when there are static local variables in the code under test, and if the user wants to consider these variables as input variables, we can't instantiate these input variables in the test drivers, because they are in the scope of the function under test.

CHAPTER 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) {     /* line 21 */
    if (t1[i] < t2[j]) {     /* line 22 */
      t3[k] = t1[i];
      i++;
      }
    else {
      t3[k] = t2[j];
      j++;
      }
    k++;
    }
  while (i < l1) {     /* line 32 */
    t3[k] = t1[i];
    i++;
    k++;
    }
  while (j < l2) {     /* line 37 */
    t3[k] = t2[j];
    j++;
    k++;
    }
}
```

7.2 A Correct `test_parameters_Merge.pl` File

```
:- module(test_parameters).

:- import create_input_val/3 from substitution.

:- export dom/4.
:- export create_input_vals/2.
:- export unquantif_preconds/2.
:- export quantif_preconds/2.
:- export strategy/2.
:- export precondition_of/2.
:- export output_options/1.

dom('Merge',cont('t1',_),[],int([-10..10])).
dom('Merge',cont('t2',_),[],int([-10..10])).

create_input_vals('Merge',Ins):-
  create_input_val(dim('t1'),int([0..10]),Ins),
  create_input_val(dim('t2'),int([0..10]),Ins),
  create_input_val(dim('t3'),int([0..20]),Ins),
  create_input_val('l1',int([0..10]),Ins),
  create_input_val('l2',int([0..10]),Ins),
  true.

unquantif_preconds('Merge',
                   [cond(supegal,dim('t1'),'l1',pre),
                    cond(supegal,dim('t2'),'l2',pre),
                    cond(supegal,dim('t3'),+('l1','l2'),pre)]).
quantif_preconds('Merge',[uq_cond([UQV3],
             [cond(supegal,UQV3,1,pre)],
             supegal,
             cont('t1',UQV3),
             cont('t1',UQV3 - 1)),
      uq_cond([UQV4],
              [cond(supegal,UQV4,1,pre)],
              supegal,
              cont('t2',UQV4),
              cont('t2',UQV4 - 1))]).

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 < l1; i++) {
    if (Pre_t1[i] != t1[i]) {
      pathcrawler_verdict_failure(); /* t1 modified */
      return; }}

  for (i = 0; i < l2; i++) {
    if (Pre_t2[i] != t2[i]) {
      pathcrawler_verdict_failure(); /* t2 modified */
      return; }}

  for (i = 0; i < l3moins1; i++) {
    if (t3[i] > t3[i+1]) {
      pathcrawler_verdict_failure(); /* t3 not ordered */
      return; }}
  i = 0;
  while (i < l3) {
    /* count occurrences of this element in t3 */
    n3 = 1;
    while (i < l3moins1 && t3[i + 1] == t3[i]) {
      i++;
      n3++; }
    /* count occurrences of this element in  t1 */
    n1 = 0;
    for (j = 0; j < l1; j++) {
      if (t1[j] == t3[i])
        n1++; }
    /* count 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)) {
       pathcrawler_verdict_failure(); /* t3 does not have the correct number of occurrences of all
elements */
      return; }
    i++; }
  pathcrawler_verdict_success();
}
```