

Twins Project Experience Report: Using PathCrawler for the Generation of Unit Tests for Embedded Software

Author : Nicky Williams, CEA List

Date : 18th September 2009

1. INTRODUCTION

The PathCrawler tool developed at CEA List was used to generate test inputs for the *blocs* module of the software embedded in Schneider Electric's VIPAM (medium voltage relay) product. After this introduction to PathCrawler and the *blocs* module, Section 2 describes what the user had to do in order to apply PathCrawler to the module, Section 3 summarises the results and Section 4 gives a conclusion. In Section 5, the application of PathCrawler and the generated tests are illustrated on two examples of the *blocs* functions and commented in detail. The source code of the *blocs* module is given in Annex 1. Annex 2 gives the test context (defined semi-manually) and the automatically generated test-cases for all 16 *blocs* functions.

1.1. PathCrawler

PathCrawler (see <http://www-list.cea.fr/labos/gb/LSL/test/pathcrawler/index.html>) is a prototype tool for the automatic generation of test vectors for the unit test of C functions. Given the source code of the function (including the source code of all included or linked files) and a test context defined by the user, PathCrawler automatically generates test vectors to satisfy some criterion of source code coverage, such as coverage of all feasible execution paths or all reachable branches. PathCrawler also identifies all infeasible execution path prefixes or unreachable branches. However, PathCrawler is based on constraint resolution techniques whose worst-case complexity is NP-complete and the search for a vector to cover a particular branch or execution path may occasionally take a very long time. In this case, the search for a test vector to cover the branch or path is abandoned after a timeout. Pathcrawler uses specialised heuristics which mean that in practice such timeouts are rare and 100% coverage of feasible execution paths or reachable branches is usually ensured.

For each test-case generated, PathCrawler provides the test vector (i.e. the concrete value of each input variable), the theoretical concrete values of the outputs (obtained by evaluating the symbolic terms over the concrete values of the test vector); the covered item (branch or execution path); the path predicate (i.e. the satisfied branch conditions expressed as arithmetic expressions over the input variables) and symbolic values of the outputs (i.e. the value of each output variable as an arithmetic expression over the input variables); and a C program which runs the test case. Note that, except for variable data-structure dimensions, which are always chosen to be the smallest possible, PathCrawler chooses the particular input variable values in each test vector non-deterministically. This means that PathCrawler will generate different test vectors in two successive test generation runs. In both runs the same paths (or branches) will be covered but they will be covered by different test vectors and maybe in a different order.

The test process using PathCrawler is illustrated in Figure 1. The user intervention to modify the source code or the test context for the *blocs* module is described in detail in Sections 2 and 5. However, since this report just covers the generation of test-cases for the *blocs* module, and not the determination of a verdict, that part of the test process will now be briefly described.

1.2. Test verdicts

PathCrawler does not provide the *correct* output values for each test-case but those which are *effectively*

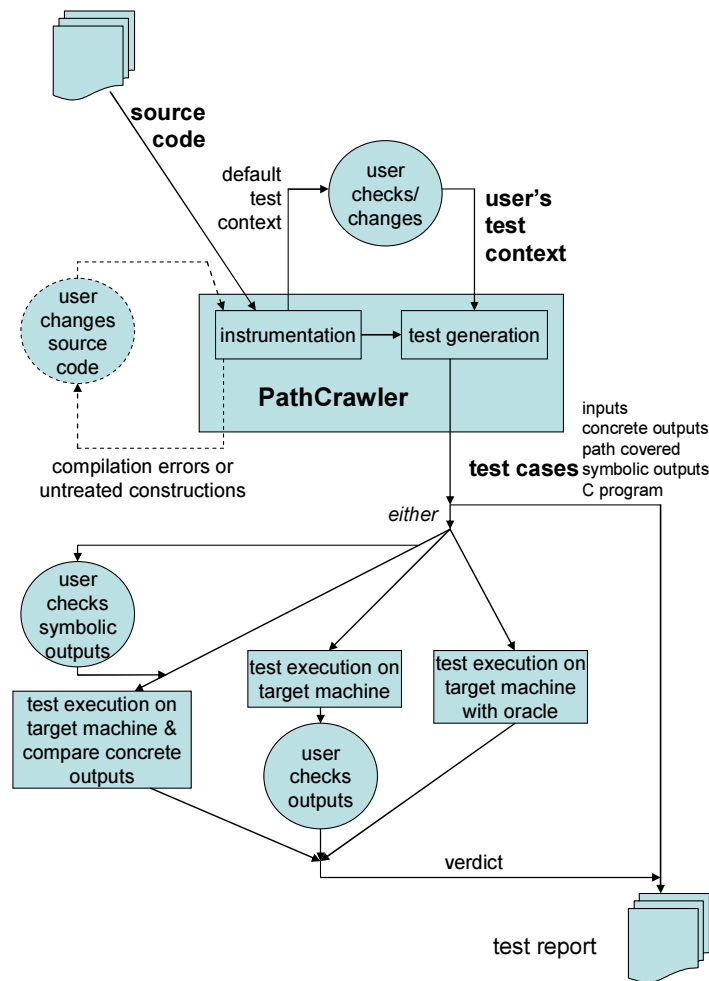


Figure 1: Test process using PathCrawler

calculated by the tested function according to the semantics of ANSI C used by PathCrawler. Indeed, PathCrawler has no knowledge of the *intended* behaviour of the tested function, it is the *user* who must decide whether the outputs of each test are correct, either by manual inspection or by some automatic oracle, such as running the same test on a reference version of the software.

If the user has such an automatic oracle (which was not the case for the *blocs* module used in this case study), then PathCrawler can also be used to output the oracle's verdict for the effective concrete values of each test case when run on the code instrumented by PathCrawler on the host machine used for test-case generation.

Note that there is a risk that the execution path and the theoretical (according to the C language semantics used by PathCrawler) outputs obtained by running a test-case on the instrumented source code and the host machine which are used for test-case generation are not the same as those effectively obtained by running the initial (un-instrumented) source code on the target machine. This is why, as illustrated in Figure 1, the user must choose how to obtain the final test verdict:

- If an automatic oracle is not available but the tested function is relatively simple (such as most of the *blocs* functions used in this case study), so that the path predicates and symbolic outputs calculated by PathCrawler can be easily understood by an engineer, then the user can decide

manually whether these path predicates and symbolic outputs are correct and then, if so, use the C programs provided by PathCrawler to run the tests on the un-instrumented source code on the target machine and compare the theoretical concrete output values calculated by PathCrawler with those effectively obtained;

- Or else, the user can ignore the symbolic and theoretical concrete output values calculated by PathCrawler and use the C programs provided by PathCrawler to run the tests on the un-instrumented source code on the target machine and decide, either manually or using an automatic oracle whether the effective (concrete) outputs of each test are correct.
- The user can also decide to ignore the potential difference with the un-instrumented code and the target machine (this option is not illustrated in Figure 1) and just accept the verdicts given by PathCrawler using the automatic oracle, if there is one, or, if not, decide the verdict of each test case by manually checking either the path predicate and symbolic output values provided by PathCrawler or (if the symbolic expressions are too complicated) the theoretical concrete output values calculated by PathCrawler;

1.1.3. The class of functions treated by PathCrawler

PathCrawler currently treats all functions written in a large subset of ANSI C which only excludes certain constructions such as (implicit or explicit) pointer casts, pointers to functions, functions with a variable number of arguments, trigonometric functions,... In fact, even these constructions can be treated if they do not impact on the branch conditions but only occur in the calculation of output values. Floating-point numbers declared as `double` can be treated but the `float` type is not yet treated.

Note that full execution path coverage of some functions may require a very large number of test cases, which means that PathCrawler effectively never finishes test-case generation. For such functions, the user must use a weaker coverage criterion.

1.1.4. PathCrawler's user interface

PathCrawler's textual, linux command line, user interface was used and the outputs are also in the form of text files. A graphical user interface is currently being developed for PathCrawler, and will be finished by the end of 2009. The new user interface will make it much easier for the user to define the test context and understand PathCrawler's output. In this report, the inputs and outputs are described in the format in which they will be presented in the new user interface.

1.1.5. The blocs software module

The *blocs* software module is written in C and forms a self-contained module composed just of two header files (*types.h* and *blocs.h*) and one source code file (*blocs.c*). The latter contains definitions of 16 functions, the longest of which is 92 lines of code (including intermediate comments and blank lines). There are two main types of function: those with a name ending in "initialisation", which allocate and initialise a data-structure pointed to by the `Contexte` formal parameter and those with a name ending in "mesurage", which read and update this data structure. The comments in the *blocs.c* source code specify that each "mesurage" function cannot be called before the corresponding "initialisation" function.

2. APPLYING PathCrawler TO THE *blocs* MODULE

2.1. Removal of pointer casts

The first step in the application of PathCrawler to this module was a modification of the *blocs.c* source file. This is because PathCrawler cannot treat (implicit or explicit) pointer casts which influence branch

conditions and some of the blocs functions contain such a cast of a formal parameter named `Contexte` whose declared type is `BLOC_CONTEXTE_T`, defined as `void*`.

In each of these functions, the input value of `Contexte` is compared to `NULL` and then, if non-`NULL`, `Contexte` is cast to a pointer to a `struct` datatype in order to read the values stored in the data structure. It is clear to an engineer reading the source code of these functions that the effective type of `Contexte` is the type used in the cast (i.e. a pointer to the `struct` datatype). However, for PathCrawler to deduce this automatically a sophisticated static analysis would be necessary and PathCrawler does not yet include this functionality.

This is why we had to transform the source code by hand in order to replace, in the type declaration of the `Contexte` parameter for these functions, `void*` by the type used in the cast. The pointer cast then becomes redundant (i.e. it “casts” `Contexte` to its declared type) and is ignored by PathCrawler.

2.2. Definition of the test context

The next step was the definition of the appropriate context in which to test the *blocs* functions.

Indeed, PathCrawler automatically defines a default test context but this may be inappropriate for the tested function. The user must decide whether the default test context represents the conditions in which the tested function will effectively be called. If not, PathCrawler may construct test-cases which use values, or combinations of values, which would never really occur in use of the tested function and which are inappropriate tests because they may reveal apparent bugs which would never actually arise in real use of the function. Moreover, the number of tests required for full coverage is often reduced when the test context is restricted to the conditions in which the tested function will effectively be called.

In this case study, the user did not really know the *blocs* module so information on the context of use of the functions was deduced from the comments in the source file and from calls to the functions under test in the application software which was made available by Schneider Electric. The modifications to the test context are explained below but they may not really correspond to those that would have been made by someone with a deeper knowledge of the functions under test.

First PathCrawler's *instrumentation* function was applied to the *blocs.c* file. One of the outputs of instrumentation is the default test context for each function in *blocs.c*. This includes a list of all possible input variables of the function, i.e. input variables of basic types such as `int`, input elements or fields of data-structures and variable dimensions of data-structures containing input elements or fields. An interval of input values is associated with each input variable and PathCrawler selects a value in this interval to construct each test-case. For input variables of basic types, the default interval of input values is given by the declared type in C (e.g. the interval `-2147483648..2147483647` for an input parameter declared as `int`). For each dimension of variable size of each data-structure containing possible input variables, the default dimension interval is `0..1`, meaning that the data-structure may have at most one element (dimension 1) or, in fact, be a `NULL` pointer (represented as dimension 0). There are 4 ways in which the user can modify the default test context:

- By removing variables from the list, if they are not, in fact, input variables (i.e. if their value on input is never read by the tested function) or if they are dimensions of data-structures which do not, in fact, contain input variables;
- By modifying the interval from which values will be selected to construct test-cases ;
- By defining (in the precondition) relations on the values to be used when constructing test-cases;
- By modifying the coverage criterion (set by default to all feasible execution paths).

In this case study, the test context of the *blocs* functions was modified in the 2nd, 3rd and 4th ways above.

2.2.1. Modification of input variable value intervals

Firstly, the intervals of values of some variables were reduced so as to avoid *arithmetic overflow*. Indeed, functions which may be called with the minimum or maximum value allowed by the declared type of a particular parameter and which then use this value in arithmetic calculations risk overflow and must be programmed accordingly. However, the *blocs* functions are clearly not expected to have to treat values at the limit of the declared types. The interval of some input variables declared as `int` was therefore reduced from

-2147483648..2147483647

to

-45000..45000

and the interval of some input variables declared as `double` from

-1.7976931348623157e+308..1.7976931348623157e+308

to

-1.7e+150..1.7e+150,

etc.

Secondly, the “initialisation” functions have an input parameter called `Fenetre` with default interval 0..65535 representing the size of a “window” of sampled values. This window is represented as an array allocated by the “initialisation” functions and attached, as the `Historique` field, to the data-structure pointed to by `Contexte`, so `Fenetre` in the calls to the “initialisation” function is usually the same as the dimension of the `Historique` field in the calls to the corresponding “mesurage” function. However, the “initialisation” functions test for, and reject, `Fenetre` values of zero, and the “mesurage” functions assume a non-zero dimension of the `Historique` field because they may reference its elements. The upper limit of the interval of values for `Fenetre` was changed to 10 times the maximum effective parameter in calls to each function in the application software and the upper limit of the interval of values for the dimension of the corresponding `Historique` field was set to the same value. The lower limit of the interval of values for the the dimensions of `Historique` fields was changed from 0 to 1 in the “mesurage” functions. The interval of values for the input parameter called `Nombre` representing the number of historical values in some functions was also changed in a similar way.

Finally, some *blocs* functions use the value of one input parameter, such as `Curseur`, as an index when reading the elements of a data-structure, such as the `Historique` field. Test cases in which `Curseur` is greater than the dimension of `Historique` would cause a buffer overflow. The user may either want to include such test cases in order to test the robustness of the tested function or else may know that it will never be called with input values that could provoke a buffer overflow. We assumed the latter and set the upper limit of the interval of values for `Curseur` to less than the upper limit of the interval of values for the dimension of `Historique`. However, modifying the interval limits is not sufficient to ensure that in each test case the value of `Curseur` is less than the dimension of the `Historique` field. For this, a *relation* must be defined between the two input values.

2.2.2. Precondition

To define relations (other than the interval in which they belong) on one or more input values, the user can code a precondition in C which returns 1 if the relation is satisfied and 0 if not. `PathCrawler` proposes a default precondition (with the same formal parameters as the tested function) which always returns 1 and the user can modify this precondition to impose relations on the input variables which will be respected by the test vectors generated by `PathCrawler`.

For instance, the precondition can be modified to compare the values of `Curseur` and of the dimension of the `Historique` field and return 1 if `Curseur` is less than the dimension of the `Historique` field and 0 if not.

Also, some *blocs* functions apply the square root operation to the result of a subtraction. In order to prevent the generation of test-cases in which the square root operation is applied to a negative number, a relation can be added to the precondition to ensure that the result of the subtraction is always greater than or equal to zero. As preconditions can only be expressed on input values, this may necessitate the user re-expressing the terms of the subtraction in terms of input values, by taking into account the assignments preceding the subtraction. Fortunately, this is easy to do for the expressions concerned in this case.

Another example where it is necessary to use a precondition is when a floating-point input value must be strictly greater or less than a particular limit. The `SeuilHaut` and `SeuilBas` data-structure fields used in some functions must have values which are strictly greater than 0. This cannot be ensured by an interval over floating-point values so the precondition must be used. It is also used to ensure that `SeuilHaut` is strictly greater than `SeuilBas`.

2.2.3. Coverage criterion

Some of the *blocs* functions contain a loop which successively treats each element of an array whose dimension is variable. This loop has a number of iterations which depends on the array dimension. To cover all feasible execution paths of such a function, each time PathCrawler generates a test to cover a particular path, p , with a certain number of iterations, n , of the loop, it must also generate one test for every other feasible path which is identical to p except that the number of iterations of the loop is different to n . If n is determined by the dimension of an array which may be quite large then the number of execution paths to be covered can be huge. However, if the tests of

- all feasible paths with no iterations of the loop,
- all feasible paths with 1 iteration of the loop,
- all feasible paths with 2 iterations of the loop and
- all feasible paths with 3 iterations of the loop

are successful, then we may be able to suppose that the tests of all other paths identical to these except in the number of loop iterations will also be successful. This is why PathCrawler proposes an alternative to the coverage of all feasible execution paths, called k -path coverage (where k is a parameter which is set by the user to a small number such as 3). This criterion is the coverage of all feasible execution paths with up to k iterations of any loop with a variable number of iterations. In fact, before the start of test vector generation, PathCrawler does not know whether the tested function contains any loops and which of these have a variable number of iterations. This means that the PathCrawler cannot always avoid generating some “redundant” test vectors which cover paths with more than k iterations. These are indicated to the user. Note that the danger of using k -path coverage is that certain paths may be infeasible for k iterations or less but feasible for some number of iterations greater than k . This is clearly not the case in the *blocs* functions with loops with a variable number of iterations, so the test strategy for these functions was set to 3-path. Note that in the case of functions with loops whose number of iterations depends on an array dimension, another way to limit the number of iterations, which avoids the possibility of redundant tests, is to limit the array dimension.

2.3. Test-case generation

Once the test context had been modified, PathCrawler’s *test-case generation* function was applied.

3. RESULTS

The number of (non-redundant) test-cases generated by PathCrawler for each function varied between 3 and 25. They were generated almost instantaneously (on a DELL Latitude D 630 portable computer) for all the functions except `bloc_sincosmoy_1_mesurage` for which test-case generation took about 20 minutes (for 25 non-redundant cases), presumably because it contains far more floating-point calculations than the other functions. There were no “timeouts” during test-case generation so 100% coverage of the criterion was achieved.

Note, however, that in the case of the “initialisation” functions which contain `malloc` instructions and branch conditions testing whether the result of the `malloc` is `NULL`, no test-cases were generated which satisfied this branch condition. This is because the result of the `malloc` instruction depends on the state of the machine when the test is run (and in modern linux kernels, the relation between the memory state and the result of `malloc` is in fact very complicated). This means that PathCrawler cannot find a test vector which will cover a path in which `malloc` returns `NULL`.

4. CONCLUSIONS

PathCrawler seems well-adapted to the automatic generation of structural unit test vectors for the *blocs* module. The only construction in the source code which could not be treated by PathCrawler was the pointer casts but these did not seem to be essential to the implementation: they were easily removed by a minor modification of the source code without changing its semantics. The modification of the test context did not demand much knowledge of the source code: it essentially consisted of avoiding the generation of tests which would cause arithmetic or buffer overflows, applying the square root operation to a negative number or have more than a certain number of iterations of any loops with a variable number of iterations. An analysis of the application software resulted in other proposed modifications of the limits of the intervals of values of the `Fenetre` argument but these may not have been entirely necessary because of the limit on the number of loop iterations. Each function has a reasonable number of feasible 3-paths and PathCrawler generated the test vectors quite fast. The only behaviour for which PathCrawler cannot generate tests is the failure of the `malloc` operations. The path predicates and symbolic outputs are relatively easy to read for an engineer so they can be checked to decide the test verdict if an automatic oracle is not available.

5. TWO EXAMPLES

The test contexts and generated test-cases for all 16 *blocs* functions can be found in Annex 2 but here we show and explain them in more detail for two example functions.

4.1. `bloc_threshold_initialisation`

Our first example is the relatively simple `bloc_threshold_initialisation` function which tests the validity of certain input values, sets the return value accordingly and, if the inputs are valid, allocates memory for the data-structure to be pointed to by the `Contexte` parameter and initialises this data-structure.

4.1.1. Test context for `bloc_threshold_initialisation`

For this function the input value ranges were modified but no input value relations were coded in the precondition. Note that limiting the “dimension” of the `Contexte` pointer in the test cases to 0 or 1

and the “dimension” of the “elements” of `Contexte` to 0 or 1 means that `Contexte` can either be the NULL pointer or be set to the address of a “data structure” of just one element, which is either the NULL pointer or is set to the address of a “data structure” of just one element. The interval of possible values of `SeuilHaut` and `SeuilBas` are limited to avoid arithmetic overflow. The maximum value of `Fenetre` is limited to 50, which is 10 times the value of the effective parameter in the call to `bloc_threshold_initialisation` in the application code. Here are the modified input value ranges:

- dimension of `Contexte` : 0..1
- dimension of elements of `Contexte` : 0..1
- `SeuilHaut` : -1.7e+150..1.7e+150
- `SeuilBas` : -1.7e+150..1.7e+150
- `Fenetre` : 0..50

The strategy was set to 3-path, i.e. coverage of all feasible execution paths with 3 or fewer iterations of any loops with a variable number of iterations.

4.1.2. Path Coverage for `bloc_threshold_initialisation`

For this function, PathCrawler generated 9 test-cases to cover 8 feasible execution paths with 3 or fewer iterations. The 4th test-case is redundant because it covers a path with 16 iterations of the loop at line 300 of the source code. The construction of such redundant test cases cannot always be avoided, as explained above. Six execution path prefixes were found to be infeasible. There were no timeouts so coverage is 100%, as can be seen in the list below of covered paths and infeasible path prefixes.

In this list, each path or prefix is annotated by the letter “C” and the number of the test-case if it is covered by a test-case, “R” and the number of the test-case if it violates the iteration limit, or “I” if it is infeasible. The path is shown as a sequence of branch conditions, each condition is denoted by the source code line number, preceded by “+” if the condition is satisfied and “-” if not. In the case of several conditional instructions in the same line of source code, the line number is followed by “_1” for the first, “_2” for the second and so on. In the case of a multiple condition, the line number is followed by the letter “b” for the second sub-condition, “c” for the third, etc. In the case of `bloc_threshold_initialisation`, the sub-conditions denoted 271b, 271c, 271d, 271e, 271f and 271g actually appear on lines 272, 272, 273, 273, 274 and 275 respectively but are not given these line numbers by PathCrawler because they are part of the overall condition which starts on line 271. When loops contain a single conditional instruction, the denotation of the condition is followed by xn , where n is the number of iterations of the loop, as for the loop at line 300 below.

Note that the conditions denoted 271c and 271e below result from the comparison with “false” of the result of the sub-conditions 271b and 271d and are redundant (the truth value of 271c is always the opposite of that of 271b and the truth value of 271e the opposite of that of 271d). This explains the infeasibility of the prefixes ending in -271b -271c, +271b +271c, -271d -271e or +271d +271e. The prefixes ending in +271f or +271g are declared as infeasible because they correspond to `malloc` returning NULL, for which PathCrawler cannot construct a test, as explained above.

```
+256 C1
-256 +271 C9
-256 -271 -271b +271c C2
-256 -271 -271b -271c I
-256 -271 +271b -271c -271d +271e C3
-256 -271 +271b -271c -271d -271e I
```

-256 -271 +271b -271c +271d -271e -271f -271g +300x16 -300 R4
 -256 -271 +271b -271c +271d -271e -271f -271g +300x3 -300 C5
 -256 -271 +271b -271c +271d -271e -271f -271g +300x2 -300 C6
 -256 -271 +271b -271c +271d -271e -271f -271g +300 -300 C7
 -256 -271 +271b -271c +271d -271e -271f -271g -300 C8
 -256 -271 +271b -271c +271d -271e -271f +271g I
 -256 -271 +271b -271c +271d -271e +271f I
 -256 -271 +271b -271c +271d +271e I
 -256 -271 +271b +271c I

4.1.3. Generated test cases for `bloc_threshold_initialisation`

Here are the details of the generated test cases. In the first test case, the meaning of the path predicate and symbolic outputs is “if `Contexte` is NULL then return 1”. For the second test case, the meaning is “if `Contexte` is non-NULL and `Fenetre` is not equal to 0 and `SeuilBas` \leq 0 then set *`Contexte` to NULL and return 1”

TEST CASE 1

Inputs:

`Contexte` = NULL;

`Fenetre` = 17;

`SeuilBas` = 192;

`SeuilHaut` = -204;

Concrete Outputs:

returned value = 1

Path:

+256

Path Predicate:

`Contexte` = NULL

Symbolic Outputs:

returned value = 1

TEST CASE 2

Inputs:

`Contexte`[0] = NULL;

`Fenetre` = 35;

`SeuilBas` = -327;

`SeuilHaut` = 198;

Concrete Outputs:

returned value = 1

`Contexte`[0] = 0

Path:

-256 -271 -271b +271c

Path Predicate:

`Contexte` \neq NULL AND

0 \neq ((int)`Fenetre`) AND

`SeuilBas` \leq 0

Symbolic Outputs:

returned value = 1
Contexte[0] = 0

TEST CASE 3

Inputs:

Contexte[0] = NULL;
Fenetre = 39;
SeuilBas = 832;
SeuilHaut = 325;

Concrete Outputs:

returned value = 1
Contexte[0] = 0

Path:

-256 -271 +271b -271c -271d +271e

Path Predicate:

Contexte <> NULL AND
0 <> ((int)Fenetre) AND
SeuilBas > 0 AND
SeuilHaut =< SeuilBas

Symbolic Outputs:

returned value = 1
Contexte[0] = 0

TEST CASE 4

violates the iteration limit

TEST CASE 5

Inputs:

Contexte[0] = NULL;
Fenetre = 4;
SeuilBas = 508;
SeuilHaut = 1306;

Concrete Outputs:

returned value = 0
Contexte[0][0][0] = 4
Contexte[0][0][1] = 0
Contexte[0][0][2] = 508
Contexte[0][0][3] = 1306
Contexte[0][0][4] = 0
Contexte[0][0][5] = -4
Contexte[0][0][6] = 4
Contexte[0][0][7][0] = 0
Contexte[0][0][7][1] = 0
Contexte[0][0][7][2] = 0
Contexte[0][0][7][3] = 0

Path:

-256 -271 +271b -271c +271d -271e -271f -271g +300x3 -300

Path Predicate:

Contexte <> NULL AND
 0 <> ((int)Fenetre) AND
 SeuilBas > 0 AND
 SeuilHaut > SeuilBas AND
 0 <> ((int)(Fenetre-1)) AND
 0 <> ((int)((Fenetre-1)-1)) AND
 0 <> ((int)(((Fenetre-1)-1)-1)) AND
 0 = ((int)((((Fenetre-1)-1)-1)-1))

Symbolic Outputs:

returned value = 0
 Contexte[0][0][0] = 4
 Contexte[0][0][1] = 0
 Contexte[0][0][2] = SeuilBas
 Contexte[0][0][3] = SeuilHaut
 Contexte[0][0][4] = 0
 Contexte[0][0][5] = -4
 Contexte[0][0][6] = 4
 Contexte[0][0][7][0] = 0
 Contexte[0][0][7][1] = 0
 Contexte[0][0][7][2] = 0
 Contexte[0][0][7][3] = 0

TEST CASE 6**Inputs:**

Contexte[0] = NULL;
 Fenetre = 3;
 SeuilBas = 262;
 SeuilHaut = 676;

Concrete Outputs:

returned value = 0
 Contexte[0][0][0] = 3
 Contexte[0][0][1] = 0
 Contexte[0][0][2] = 262
 Contexte[0][0][3] = 676
 Contexte[0][0][4] = 0
 Contexte[0][0][5] = -3
 Contexte[0][0][6] = 3
 Contexte[0][0][7][0] = 0
 Contexte[0][0][7][1] = 0
 Contexte[0][0][7][2] = 0

Path:

-256 -271 +271b -271c +271d -271e -271f -271g +300x2 -300

Path Predicate:

Contexte <> NULL AND
 0 <> ((int)Fenetre) AND
 SeuilBas > 0 AND

SeuilHaut > SeuilBas AND
0 <> ((int)(Fenetre-1)) AND
0 <> ((int)((Fenetre-1)-1)) AND
0 = ((int)(((Fenetre-1)-1)-1))

Symbolic Outputs:

returned value = 0
Contexte[0][0][0] = 3
Contexte[0][0][1] = 0
Contexte[0][0][2] = SeuilBas
Contexte[0][0][3] = SeuilHaut
Contexte[0][0][4] = 0
Contexte[0][0][5] = -3
Contexte[0][0][6] = 3
Contexte[0][0][7][0] = 0
Contexte[0][0][7][1] = 0
Contexte[0][0][7][2] = 0

TEST CASE 7

Inputs:

Contexte[0] = NULL;
Fenetre = 2;
SeuilBas = 463;
SeuilHaut = 534;

Concrete Outputs:

returned value = 0
Contexte[0][0][0] = 2
Contexte[0][0][1] = 0
Contexte[0][0][2] = 463
Contexte[0][0][3] = 534
Contexte[0][0][4] = 0
Contexte[0][0][5] = -2
Contexte[0][0][6] = 2
Contexte[0][0][7][0] = 0
Contexte[0][0][7][1] = 0

Path:

-256 -271 +271b -271c +271d -271e -271f -271g +300 -300

Path Predicate:

Contexte <> NULL AND
0 <> ((int)Fenetre) AND
SeuilBas > 0 AND
SeuilHaut > SeuilBas AND
0 <> ((int)(Fenetre-1)) AND
0 = ((int)((Fenetre-1)-1))

Symbolic Outputs:

returned value = 0
Contexte[0][0][0] = 2
Contexte[0][0][1] = 0

Contexte[0][0][2] = SeuilBas
Contexte[0][0][3] = SeuilHaut
Contexte[0][0][4] = 0
Contexte[0][0][5] = -2
Contexte[0][0][6] = 2
Contexte[0][0][7][0] = 0
Contexte[0][0][7][1] = 0

TEST CASE 8

Inputs:

Contexte[0] = NULL;
Fenetre = 1;
SeuilBas = 187;
SeuilHaut = 783;

Concrete Outputs:

returned value = 0
Contexte[0][0][0] = 1
Contexte[0][0][1] = 0
Contexte[0][0][2] = 187
Contexte[0][0][3] = 783
Contexte[0][0][4] = 0
Contexte[0][0][5] = -1
Contexte[0][0][6] = 1
Contexte[0][0][7][0] = 0

Path:

-256 -271 +271b -271c +271d -271e -271f -271g -300

Path Predicate:

Contexte <> NULL AND
0 <> ((int)Fenetre) AND
SeuilBas > 0 AND
SeuilHaut > SeuilBas AND
0 = ((int)(Fenetre-1))

Symbolic Outputs:

returned value = 0
Contexte[0][0][0] = 1
Contexte[0][0][1] = 0
Contexte[0][0][2] = SeuilBas
Contexte[0][0][3] = SeuilHaut
Contexte[0][0][4] = 0
Contexte[0][0][5] = -1
Contexte[0][0][6] = 1
Contexte[0][0][7][0] = 0

TEST CASE 9

Inputs:

Contexte[0] = NULL;
Fenetre = 0;

SeuilBas = -213;

SeuilHaut = 405;

Concrete Outputs:

returned value = 0

Contexte[0] = 0

Path:

-256 +271

Path Predicate:

Contexte <> NULL AND

0 = ((int)Fenetre)

Symbolic Outputs:

returned value = 0

Contexte[0] = 0

4.2. bloc_threshold_mesurage

Our second example is the more complicated `bloc_threshold_mesurage` function which is called after `bloc_threshold_initialisation` and which tests the validity of certain input values, sets the return value accordingly, and then, if the inputs are valid, compares the input parameter `Mesure` to an element of the data-structure pointed to by the `Contexte` parameter and then updates this data-structure and sets the `Detection` parameter.

4.2.1. Test context for bloc_threshold_mesurage

As in `bloc_threshold_initialisation`, the interval of the values of inputs declared as `double` (`Contexte->SeuilBas`, `Contexte->SeuilHaut`, `Contexte->Somme`, `Contexte->SommeBas`, `Contexte->SommeHaut`, `Mesure`) were limited to avoid arithmetic overflow and the upper limit of `Contexte->Fenetre` was set to 50. The interval of the values of the “dimension” of `Contexte->Historique`, was set to 1..50 because in the application code it is set to the same non-zero value as `Fenetre` in the call to `bloc_threshold_initialisation` and moreover, `Contexte->Historique` should be non-NULL because its elements may be read by `bloc_threshold_mesurage`. The interval of the values declared as `int` (`Detection` and the elements of `Contexte->Historique`) were limited to -45000..45000 to avoid arithmetic overflow.

In this function, `Contexte->Curseur` is used as an index to the `Contexte->Historique` array so it should be at least 0 and less than the dimension of `Contexte->Historique`. This is why the interval of the values of `Contexte->Curseur` was set to 0..49 and the precondition was coded to include the relation:

`Contexte->Curseur < dim(Contexte->Historique)`.

Moreover, this function ensures that `Contexte->Curseur` is always less than

`Contexte->Fenetre` so the following relation was also coded into the precondition:

`Contexte->Curseur < Contexte->Fenetre`.

Finally, `bloc_threshold_initialisation` (always called before `bloc_threshold_mesurage` according to the comments) ensures the following relations, which were also added to the precondition:

`Contexte->SeuilBas > 0`

`Contexte->SeuilHaut > Contexte->SeuilBas`.

Note that all the previous relations are only enforced if `Contexte` is non-NULL.

4.2.2. Path Coverage for `bloc_threshold_mesurage`

For this function, PathCrawler generated 20 test-cases to cover 20 feasible execution paths. Six execution path prefixes were found to be infeasible. There were no timeouts so coverage is 100%, as can be seen in the list below of covered paths and infeasible path prefixes.

```
+368 C1
-368 +368b C2
-368 -368b -384 -384_2 -388 +388_2 -407 +417 C5
-368 -368b -384 -384_2 -388 +388_2 -407 -417 +421 C3
-368 -368b -384 -384_2 -388 +388_2 -407 -417 -421 C4
-368 -368b -384 -384_2 -388 +388_2 +407 +417 C6
-368 -368b -384 -384_2 -388 +388_2 +407 -417 +421 C7
-368 -368b -384 -384_2 -388 +388_2 +407 -417 -421 C8
-368 -368b -384 -384_2 -388 -388_2 I
-368 -368b -384 -384_2 +388 -388_2 -407 +417 C11
-368 -368b -384 -384_2 +388 -388_2 -407 -417 +421 C9
-368 -368b -384 -384_2 +388 -388_2 -407 -417 -421 C10
-368 -368b -384 -384_2 +388 -388_2 +407 +417 C14
-368 -368b -384 -384_2 +388 -388_2 +407 -417 +421 C12
-368 -368b -384 -384_2 +388 -388_2 +407 -417 -421 C13
-368 -368b -384 -384_2 +388 +388_2 I
-368 -368b -384 +384_2 I
-368 -368b +384 +384_2 -407 +417 C15
-368 -368b +384 +384_2 -407 -417 +421 C16
-368 -368b +384 +384_2 -407 -417 -421 C17
-368 -368b +384 +384_2 +407 +417 C18
-368 -368b +384 +384_2 +407 -417 +421 C19
-368 -368b +384 +384_2 +407 -417 -421 C20
-368 -368b +384 -384_2 I
```

4.1.3. Generated test cases for `bloc_threshold_mesurage`

Here are the details of the first 3 test cases generated. The other 17 are included in Annex 2.

TEST CASE 1

Inputs:

Contexte = NULL;

Detection = NULL;

Mesure = -423

Concrete Outputs:

returned value = 1

Path:

+368

Path Predicate:

Contexte = NULL

Symbolic Outputs:

returned value = 1

TEST CASE 2

Inputs:

Detection = NULL;

Mesure = -411;

Contexte->Fenetre = 30;

Contexte->Curseur = 0;

Contexte->SeuilBas = 2;

Contexte->SeuilHaut = 117;

Contexte->Somme = -450;

Contexte->SommeBas = -49;

Contexte->SommeHaut = 288;

Contexte->Historique[0] = 5;

Concrete Outputs:

returned value = 1

Path:

-368 +368b

Path Predicate:

Contexte <> NULL AND

Detection = NULL

Symbolic Outputs:

returned value = 1

TEST CASE 3

Inputs:

Mesure = -54;

Detection[0] = 1;

Contexte->Fenetre = 32;

Contexte->Curseur = 0;

Contexte->SeuilBas = 51;

Contexte->SeuilHaut = 504;

Contexte->Somme = 395;

Contexte->SommeBas = -142;

Contexte->SommeHaut = 147;

Contexte->Historique[0] = 4;

Concrete Outputs:

returned value = 0

Contexte->Historique[0] = -1

Contexte->Somme = 390

Contexte->Curseur = 1

Detection[0] = 1

Path:

-368 -368b -384 -384_2 -388 +388_2 -407 -417 +421

Path Predicate:

Contexte <> NULL AND

Detection <> NULL AND

Mesure =< Contexte->SeuilHaut AND

Mesure < Contexte->SeuilHaut AND
(Contexte->Curseur+1) < ((int)Contexte->Fenetre) AND
((Contexte->Somme-((double)Contexte->Historique[Contexte->Curseur]))-1) > Contexte->SommeBas
AND
((Contexte->Somme-((double)Contexte->Historique[Contexte->Curseur]))-1) >=
Contexte->SommeHaut

Symbolic Outputs:

returned value = 0

Contexte->Historique[0] = -1

Contexte->Somme = (Contexte->Somme-((double)Contexte->Historique[Contexte->Curseur]))-1

Contexte->Curseur = Contexte->Curseur+1

Detection[0] = 1