Automating Structural Testing of C Programs: Experience with PathCrawler[*]

Bernard Botella   Mickaël Delahaye   Stéphane Hong-Tuan-Ha   Nikolai Kosmatov
Patricia Mouy   Muriel Roger   Nicky Williams

CEA LIST, Software Reliability Laboratory
91191 Gif-sur-Yvette France
Email: firstname.lastname@cea.fr

Abstract

Structural testing is widely used in industrial verification processes of critical software. This report presents PathCrawler, a structural test generation tool that may be used to automate this activity, and several evaluation criteria of automatic test generation tools for C programs. These criteria correspond to the issues identified during our ongoing experience in the development of PathCrawler and its application to industrial software. They include issues arising for some specific types of software. Some of them are still difficult open problems. Others are (partially) solved, and the solution adopted in PathCrawler is discussed. We believe that these criteria must be satisfied in order for the automation of structural testing to become an industrial reality.

1 Introduction

Structural testing (also called white-box testing) is meant to assure that the software has been thoroughly exercised by execution of the test set. Classically, it is used at unit level and, depending on the test strategy, the coverage criterion is all-branches, all-paths or MC/DC (see for example [19]). When the tests are constructed manually, the coverage exhibited by the test sets with respect to these criteria is often less than 80%, and even lower for a more complicated criterion such as MC/DC.

Automation of structural test generation is possible. In the past, tools were based on random strategies with coverage results not much better than those obtained by humans. Today they are based on a precise analysis of the source code of the software and a conversion of each elementary objective (branch, path or partial path) into a constraint system which is then solved using some automatic constraint solving techniques [32, 13, 14]. They provide an input data set and may allow the user to restrict the possible values for these data to take contextual information into account during testing. The user has then to execute, using some specialised tool, the program with this input data set on the target platform, to verify the results against specification and check the effective coverage.

Automation of test case generation brings obvious benefits. In critical systems processes where structural testing is required by the development norm, manually creating tests from the specification fails to achieve complete satisfaction of the coverage criterion. In this case, automatic methods help to reach the objectives which are not covered and provide corresponding path conditions that may be used to refine the specification if needed. They may also determine whether the objectives which are not yet covered are really infeasible. When the development process does not impose any structural testing activity, the use of a structural test generation tool is a way to increase the quality of the software with a very low cost overhead.

Automatic structural test generators may also be used for other purposes, for example they may be used to find execution errors [11, 27, 7], to verify conformity to specifications [28, 2, 8] or to verify non-functional properties [30].

In this article we present the PathCrawler structural test generator for C and C++ programs. We briefly introduce its functions and the method it uses to generate test cases (Section 2). Our contribution is to expose the main difficulties such a tool has to face in order to work on real industrial software and the solutions that we have adopted in PathCrawler (Section 3). These difficulties were identified during our ongoing experience in the development of PathCrawler and its application to industrial software.

2 Presentation of the PathCrawler Tool

PathCrawler is a test generation tool for C functions respecting the all-paths criterion, or the k-path criterion (for a given $k \geq 0$), which restricts the generation to the paths

[*]This work has been partially funded by the ANR-PREDIT MASCOTTE, ITEA SPICES, ITEA TWINS and ANR CAVERN projects.

```
int bsearch(int a[4], int key) {
  int low = 0; int high = 3;
  while (low <= high) {
    int mid = low + (high-low)/2;
    int midVal = a[mid];
    if (midVal < key) {
      low = mid+1;
    } else if (midVal > key) {
      high = mid-1;
    } else {
      return mid;
    }
  }
  return -1;
}
```

Figure 1. C function for binary search

with at most k consecutive iterations of each loop. The user provides the ANSI C source files containing the function under test, which we denote by f, and other functions called by f. Test generation with PathCrawler contains two major phases.

In the first phase, PathCrawler extracts the inputs of f and instruments the source code in order to create a test driver. This phase uses the CIL library [25]. The extracted inputs include the formal parameters of f and the non constant global variables. A test case will provide a value for each input of f. The user may remove some variables from the inputs, define the domains of the inputs, a test context and an oracle.

The second phase generates test cases for f with the selected criterion. Implemented in Eclipse constraint logic programming environment[1], the generator combines symbolic execution in constraints and concrete execution. The paths of f are explored in a depth-first search. Let us describe (a simplified version of) the PathCrawler test generation method in more detail.

We can denote an execution path by a sequence of decisions, e.g. a^+, b^-, c^-, d^+, where a, b, c, d designate control points (in some conditional or loop statements). A decision is denoted by the control point followed by a "$+$" if the condition is true, and by a "$-$" otherwise. The mark "$\star$" after a decision indicates that the other branch has already been explored (it will be explained in detail below).

The generator needs the test driver with the instrumented version of f to trace the execution path on a generated test case. The generator's main loop is rather simple: given a partial program path π, the main idea is to symbolically execute it using constraints. A solution of the resulting constraint solving problem will provide a test case exercising a path starting with π. Then the trick is to use concrete execution of the test case on the instrumented version to obtain the complete path. The partial paths are explored in a depth-first search.

For symbolic execution of a program in constraints, PathCrawler maintains:

- a memory state of the program at each moment of symbolic execution. It is basically a mapping associating a value to a symbolic name. The symbolic name is a variable name or an array element. The value is a constant or a logical variable.
- the current partial path π in the program. When a test case is successfully generated for the partial path π, the remaining part of the path it activates is denoted by σ.
- a constraint store with the constraints added by the symbolic execution of the current partial path π.

The method contains the following steps:

(Initialisation) Create a logical variable for each input and associate it with this input. Set initial values of initialised variables. Add constraints for the precondition. Let the initial partial path π be empty. Continue to (Step 1).

(Step 1) Let σ be empty. Symbolically execute the partial path π, that is, add constraints and update the memory according to the instructions in π. If some constraint fails, continue to (Step 4). Otherwise, continue to (Step 2).

(Step 2) Call the constraint solver to generate a test case, that is, concrete values for the inputs, satisfying the current constraints. If it fails, go to (Step 4). Otherwise, continue to (Step 3).

(Step 3) Run the test driver with traced execution of f on the test case generated in (Step 2) to obtain the complete execution path. The complete path must start with π. Save the remaining part into σ. Continue to (Step 4).

(Step 4) Let ρ be the concatenation of π and σ. Try to find in ρ the last unmarked decision, i.e. the last decision without a "$\star$" mark. If ρ contains no unmarked decision, exit. Otherwise, if $d^{\pm}$ is the last unmarked decision in ρ, set π to the subpath of ρ before $d^{\pm}$, followed by $d^{\mp}_{\star}$ (i.e. the negation of $d^{\pm}$ marked as already processed), and continue to (Step 1).

Notice that Step 4 chooses the next partial path in a depth-first search. It changes the last unmarked decision in ρ to look for differences as deep as possible first, and marks a decision by a "$\star$" when its negation (i.e. the other branch from this node in the tree of all execution paths) has already been fully explored. For example, if $\rho = a^+\, b^-_{\star}\, c^-\, d^-\, e^+_{\star}$, the last unmarked decision is d^-, so we take the subpath of ρ before this decision $a^+\, b^-_{\star}\, c^-$, and add $d^+_{\star}$ to it to obtain the new partial path $\pi = a^+\, b^-_{\star}\, c^-\, d^+_{\star}$.

We will use as a running example the C function shown in Figure 1. To simplify the example, we limit the array size to 4, and the domain of elements to $[0, 100]$. The function `bsearch` takes two parameters: an array `a` of four integers $\in [0, 100]$ and an integer `key` $\in [0, 100]$. Given that `a` is sorted in ascending order, the function returns the index of some occurrence of `key` in `a` if `key` is present in `a`, or -1

[1]http://www.eclipse-clp.org

(1) Memory	Constrs
`a`$[0] \mapsto X_0$	$\langle pre \rangle$
`a`$[1] \mapsto X_1$	
`a`$[2] \mapsto X_2$	
`a`$[3] \mapsto X_3$	
`key` $\mapsto X_4$	
$\pi = \epsilon$	

$\rightsquigarrow$

Test case 1
$X_0 = 5$
$X_1 = 17$
$X_2 = 42$
$X_3 = 70$
$X_4 = 23$
$\sigma = 3^+ 6^+ 3^+ 6^- 8^+ 3^-$

$\rightarrow$

(2) Memory	Constraints
[...]	$\langle pre \rangle$
`low` $\mapsto 2$	$0 \leq 3, X_1 < X_4$
`high` $\mapsto 1$	$2 \leq 3, X_2 \geq X_4$
`mid` $\mapsto 2$	$X_2 > X_4$
`midVal` $\mapsto X_2$	$2 \leq 1$
$\pi = 3^+ 6^+ 3^+ 6^- 8^+ 3^+_\star$	

$\rightsquigarrow$ infeasible $\rightarrow$

(3) Memory	Constraints
[...]	$\langle pre \rangle$
`low` $\mapsto 2$	$0 \leq 3, X_1 < X_4$
`high` $\mapsto 3$	$2 \leq 3, X_2 \geq X_4$
`mid` $\mapsto 2$	$X_2 \leq X_4$
`midVal` $\mapsto X_2$	
$\pi = 3^+ 6^+ 3^+ 6^- 8^-_\star$	

$\rightsquigarrow$

Test case 2
$X_0 = 25$
$X_1 = 40$
$X_2 = 47$
$X_3 = 97$
$X_4 = 47$
$\sigma = \epsilon$

$\ldots$

Figure 2. Depth-first generation of all-paths test cases for `bsearch`, where $\rightarrow$ denotes application of Steps 2, 3 and $\rightsquigarrow$ denotes application of Steps 4, 1.

otherwise. Here, the precondition contains the definition of the variables' domains and the property that `a` is sorted in ascending order. We assume the oracle is provided, and focus on the generation of test data.

Figure 2 shows how our method proceeds on this example. The empty path is denoted by ϵ. In the state (1), we see that the initialisation step associates a logical variable X_i to each input, i.e. to each element of `a` and to `key`, and posts the precondition $\langle pre \rangle$ to the constraint store. Here, $\langle pre \rangle$ denotes the constraints: $X_0, \ldots, X_4 \in [0, 100]$ and $X_0 \leq X_1 \leq X_2 \leq X_3$.

As the original prefix π is empty, Step 1 is trivial and adds no constraints. Step 2 consists of choosing a first test case. In Step 3, we retrieve the complete path traced during the concrete execution of Test case 1, and obtain $\sigma = 3^+ 6^+ 3^+ 6^- 8^+ 3^-$. (We use abbreviated path notation where we write decisions only.)

Step 4 sets $\rho = 3^+ 6^+ 3^+ 6^- 8^+ 3^-$ and, therefore, the new path prefix $\pi = 3^+ 6^+ 3^+ 6^- 8^+ 3^+_\star$ by negating the last not-yet-negated decision. Now, Step 1 symbolically executes this path prefix in constraints for unknown inputs, and the resulting state is shown in (2). Let us explain this execution in detail. First, the execution of line 2 adds `low` $\mapsto 0$ and `high` $\mapsto 3$ into the memory. The conditional expression at line 3 is interpreted as a constraint $0 \leq 3$ after replacing the variables by their current values in the memory map. The assignments of lines 4 and 5 add `mid` $\mapsto 1$ and `midVal` $\mapsto X_1$, X_1 being the current symbolic value of $a[1]$. The conditional expression at line 6 gives the constraint $X_1 < X_4$, since X_4 is the symbolic value associated to `key`. At line 7, the memory is updated with `low` $\mapsto 2$. Line 3 adds the trivial constraint $2 \leq 3$. Lines 4 and 5 update the memory map with `mid` $\mapsto 2$ and `midVal` $\mapsto X_2$. Because of the minus sign, the expression at line 6 is negated and gives the constraint $X_2 \geq X_4$. Line 8 posts the constraint $X_2 > X_4$. Line 9 changes the value of `high` to 1 in the memory. Finally the last conditional node 3^+ gives the false constraint $2 \leq 1$, so the path prefix is infeasible. The last constraint obviously fails, which is detected by our solver at the propagation step while posting the constraint, and Step 1 continues directly to Step 4. The intermediate states were not detailed in Figure 2.

We are now going from (2) to (3) in Figure 2. Step 4 computes the complete path $\rho = 3^+ 6^+ 3^+ 6^- 8^+ 3^+_\star$. As $3^+_\star$ means that its negation has already been explored, the new prefix π is $3^+ 6^+ 3^+ 6^- 8^-_\star$. Next, Step 1 symbolically executes this partial path. It can be done from the initial state (1). However, in practice, backtracking allows us to come back to the closest intermediate state (here, the state just before $X_2 > X_4$ was posted by the previous execution), from which we can reach the current path prefix in a minimal number of steps. Next, Step 2 generates Test case 2. Step 3 sets σ to ϵ. Step 4 computes the new prefix $\pi = 3^+ 6^+ 3^+ 6^+_\star$, and so on. The reader will find applications of this method to other examples in [31, 32, 16].

3 Towards an Automatic Testing Tool

Over the last few years we have applied the PathCrawler prototype to many examples of industrial software, especially embedded software. Scaling-up to programs of hundreds or thousands of lines of code has not really proved a problem. PathCrawler is robust and efficient, capable of generating test cases which cover millions of paths, which can have hundreds of control points, of a function under test. However, real industrial software raises other issues that are not seen in trials on academic examples.

In this Section, we start by examining the properties needed for a test generation tool to satisfy a coverage criterion. We discuss how we can realistically interpret and apply the rather naive and badly-defined all-paths criterion. We explain how the user can avoid detecting irrelevant bugs by defining the context in which the function under test will be called, its precondition. We discuss features of real programs that are rarely addressed in the literature, such as library calls and floating-point numbers. Real programming languages have very complicated semantics, which makes it difficult to translate branch conditions into constraints. There are the classic problems of aliasing and pointers in C and the semantics of C++ is even more complicated than that of C. We examine the factors that influence the efficiency of automatic test-case generation. Finally we point out that automatic test-case generation must be specialised to treat the types of software that occur very frequently in embedded systems.

These are the criteria which we believe must be satisfied by test-case generation tools in order for the automation of structural testing to become an industrial reality.

3.1 Soundness and Completeness

Soundness and completeness of generated test cases are important evaluation criteria for automatic test generation tools because they are necessary for 100% satisfaction of coverage criteria.

Test case generation is *sound* when each test case activates the test objective (path, branch, instruction, etc.) for which it was generated and *complete* when absence of a test for some test objective means this test objective is infeasible.

The soundness of the PathCrawler method presented in Section 2 is verified by concrete execution of generated test cases on the instrumented version of the program under test. The path trace obtained by the concrete execution of a test case confirms that this test case really executes the path for which it was generated.

Completeness can only be guaranteed when symbolic execution of all features of the program is correct and when constraint solving terminates within a reasonable timeout for all paths. This is difficult for real-life code, as explained in Sections 3.5, 3.6 and 3.8.

Note that completeness and the verification of soundness on the instrumented code actually require symbolic execution of program features to be adapted to the target platform (compiler optimisations, libraries, floating-point unit, etc) of the function under test and also PathCrawler's execution of the tests on the instrumented code to be carried out in the same environment. PathCrawler is currently only adapted to our Linux development environment and Intel-based platform.

The depth-first search of the PathCrawler method enables iteration over all feasible paths of the program, which is necessary for completeness, for all terminating programs with finitely many paths. Programs containing infinite loops cannot be tested in any case in the way we propose here as the execution of the program on the test inputs would never terminate. Any infinite loop which has been introduced as the result of a bug can only be detected by a timeout on the execution of each test-case on the instrumented code. Terminating programs with an infinite number of paths must have an infinite number of inputs and this is another class of programs that cannot be tested using the PathCrawler method.

Unlike concolic tools such as CUTE [27] and DART [11], for which soundness and completeness are lower priorities than the treatment, even if partial, of all programs, PathCrawler guarantees satisfaction of the all-paths criterion for a certain class of programs.

3.2 Limiting Path Explosion

The practical limitation to completeness is the number of feasible execution paths in the program under test. Programs do not need to have very many lines of code, or even control points, to have an astronomical number of execution paths. Such a combinatorial explosion in the number of execution paths can be due to 3 factors:

- long sequences of conditional instructions : the number of paths can be 2^l where l is the number of conditions. All feasible execution paths of such programs can often only be covered if the pre-condition on the context in which the program is to be tested (see Section 3.4) happens to eliminate many potential paths. Otherwise, path coverage may have to be abandoned in favour of branch coverage.

- Loops with a variable number of iterations : for each path containing 0 iterations of such a loop, there is another path containing 1 iteration etc. up to the maximum number of iterations. In many cases, the regularity in the loop means that if the test results are correct for all paths with a small number of iterations, then they will be for all paths with any number of iterations. The all-feasible-paths criterion can then be relaxed to a criterion such as the classic *k*-path, proposed by PathCrawler (and described in [31]), in which only feasible paths with up to *k* iterations of such loops are tested, where *k* is chosen by the user. However, the danger of the *k*-path criterion is that some path suffixes after exit from the loop may only be feasible in case of more than *k* iterations of the loop.

- Function calls : many test-case generators treat function calls by inlining the source code of called functions if it is available (for the case where the source code is not available, see Section 3.3). This combines the number of paths in the function under test with the number of paths in the called function, which greatly increases the number of paths to be tested and may result in many tests covering different paths through the called function for the same path in the calling function. In this case too, the best solution seems to be a more precise interpretation of the all-paths criterion. If bottom-up unit testing is being carried out then called functions will be path-tested before the calling function and it is sufficient to test just all feasible paths of the calling function.

The following table shows experimental results of test generation with different criteria for the function `Merge` (see [32]) that takes two sorted arrays of length ≤ 10 and merges them to a new sorted array. We see that the *k*-path criterion considerably reduces test generation time and the number of test cases.

criterion	$k = 2$	$k = 5$	$k = 10$	$k = 15$	all-paths
time (s)	0.33	0.80	37.2	876.65	3 407.98
♯ test cases	19	337	12 798	216 371	705 431

In the case of loops and of function calls, we would like to explore additional iterations of the loop or additional paths only when it is necessary in order to cover all paths of the function under test, i.e. when a path in the rest of the function under test is only feasible in the context of additional loop iterations or an unexplored path in the called function. We are currently studying the modification of PathCrawler's strategy to enable this minimal exploration. Our approach is based on the storage of infeasible path suffixes used in [23], optimised by taking into account the dependencies between the infeasible suffix and the path through the loop or function call. A similarly "lazy" approach is proposed for the treatment of function calls in [10], but this approach stores not infeasible path suffixes but the result of symbolic execution of each path through a called function which has already been explored.

Among other approaches to the path explosion problem in all-paths testing, CUTE [27] proposes to approximate function return values by concrete values, but this endangers completeness. Path exploration can be guided by particular heuristics [7], or using a combination of random testing and symbolic execution [17]. State-caching, a technique arising from static analysis, is used by [4] to prune the paths which are not interesting with respect to given test objectives.

3.3 Treating Library Function Calls

Real code often contains calls to functions whose source code is not available but many structural test-case generation tools cannot treat these calls in a satisfactory way. In PathCrawler we propose a novel method to overcome this limitation of structural testing when the called function is a library or off-the-shelf software component (COTS) for which there is a detailed description of the functionality and restrictions on usage. As far as we know, it is the only work which addresses this problem.

When the source code of the called function is not available, testing traditionally uses stubs [24]. These are built manually in an ad-hoc way and are often an incomplete description of the called function, which can lead to incomplete testing. They cannot be used for the automatic generation of unit tests.

Our method is based on a formal specification of the called function. This is why it can also be used when the source code of the called function is available and the called function has already been validated using a formal specification. Indeed, we believe that to achieve increased test automation users are often prepared to formalise the specifications of called functions if the specification language is appropriate.

We therefore propose a language which uses the same function names and types as the C code and corresponds to first-order logic on finite domains. It is similar to the usual languages used for defining assertions in source code. The specifications are structured as pre/post-condition couples [15]. This format is easy for users to understand. Furthermore, it is already widely used in industry, for example to specify conditions in state-transition systems.

We use the specification to abstract the called function. The idea is to abstract the internal structural paths of the called function by the definition of the corresponding functional domains.

In the method described in Section 2, the C instructions are translated into constraints by PathCrawler so that producing a new test case becomes a constraint solving problem. Now, the specification of the called function is also interpreted as constraints.

We have defined two different coverage criteria. The first corresponds to the coverage of all feasible paths of the function under test and all the functional domains for every calling context of the called functions. However, if we only need to cover all the paths in the calling function, then this criterion sometimes results in redundant tests. These are the tests which exercise different functional domains within a called function but are identical within the calling function. The second criterion requires just all-paths coverage of the calling function, with the least possible exploration of the functional domains of called functions. We have modified PathCrawler's method to generate tests respecting either of these criteria. More details can be found in [23].

In our approach, the maximum number of cases to be considered depends on the number of pre/post cases in the specification which is unlikely to be more, and may be far less, than the number of feasible execution paths. However, our approach preserves the completeness of the coverage of paths in the calling functions. It is an example of grey-box test selection strategy that advantageously combines white-box (structural) and black-box (functional) strategies in order to achieve automation of unit testing.

3.4 Enabling Definition of Test Contexts

Automated testing tools must offer a means for the user to define a context for the function under test. Indeed, although defensive programming advocates embedding run-time precondition verification in the function code, many functions are programmed without such safety mechanisms, notably because of performance issues or an unknown specification. Moreover, time and other resources may be too scarce for testing numerous out-of-domain behaviours.

Program subroutines often come with formal or informal conditions on the input. These may correspond to the definition domain, for instance: the C standard library function `sqrt`, which takes a `double` and returns its square root as a `double`, is actually defined only for non-negative numbers. But the user may wish to impose additional restrictions on the context in which the function is to be tested.

Let us call the conditions on the inputs for which a function is to be tested the *precondition* of the function. In other words, the test domain is obtained from the input variables' types filtered by the precondition, for a function f with n inputs of types t_1 to t_n:

$$testdom(f) = \{X \in t_1 \times \cdots \times t_n | Pre(f, X)\} .$$

However, expressing the precondition is not easy. The tester needs to actually know the formal precondition. He must also code the precondition in such a way that the testing tool can use it. For constraint-based testing, the natural way is to code these conditions into constraints. Even though bound checking is often trivial to code, harder preconditions, like order or balancing requirements, can be difficult to code in constraints for an imperative language programmer.

This is why PathCrawler offers two ways to express the precondition of the function under test. First, it accepts a precondition expressed in constraints on the inputs, to be posted before test case generation. Second, PathCrawler offers an original method to write the precondition as a C function. The precondition function takes the same inputs as the function under test and returns true if and only if the inputs respect the precondition, that is, belong to the function domain. Let us give a precondition function for the function `bsearch` of Figure 1. It returns one if the array is sorted in ascending order, zero otherwise:

```
int bsearch_precond(int a[4], int key) {
  int i;
  for (i = 1; i < N; i++)
    if (a[i] < a[i-1]) return 0;
  return 1; }
```

To solve the precondition, i.e. to generate only the inputs for which it returns true, PathCrawler uses symbolic execution and concrete execution of the precondition function. The challenge consists in finding valid inputs lazily without actually exercising the precondition. Despite promising experiments, this remains an active research direction because it affects the scalability of the overall method.

Similarly, other tools like Java PathFinder [29] and CUTE [27] allow the user to provide a consistency check for structure invariants in the tested language. Some tools also allow the user to describe how to construct a valid input (rather than how to check whether a test-case is valid), also called *finitization* [6]. In the same logic, the authors of PEX [28] propose to write basic test scenarios and to generalise them by replacing constants with parameters in order to obtain *parameterised unit tests*.

3.5 The Memory Model

The treatment of arrays, pointers, pointer casts, type unions and primitive C operations on bits is one of the difficult aspects of automatic test generation for languages such as C. Unfortunately, these constructions are often found in industrial software.

PathCrawler only partially treats these constructions at the present time. This is also the case for comparable tools CUTE [27] and EXE [7], each tool having its own strong and weak points.

The treatment of dereferenced pointers, such as in the branch condition `if(*p == *q)`, where `int *p,*q;` poses no problem for most tools. However, in order to treat branch conditions such as `if(p == q)` (with `int *p,*q;`), it must be possible to post constraints on the values of pointers on input. These values are memory addresses that can change with each execution; they cannot be generated as test inputs but must be represented symbolically (such as in CUTE) in order to handle such conditions.

Pointer arithmetic is treated by PathCrawler as long as there are no explicit or implicit casts of pointers. Some uses of type unions are equivalent to pointer casts and so cannot be treated either by PathCrawler. Treating pointer casts necessitates a low-level model of the memory, including the size in bits of each variable and their relative positions. Constraint solving techniques may also have to be adapted to treat bit-level representations. Although PathCrawler cannot handle pointer casts, it does have special constraints to treat operations on bits. EXE has a low-level memory model and is based on a SAT solver. It can handle bit operations, pointer arithmetic and also pointer casts, but its use of bit vectors to model the memory means it can only treat one level of pointer dereferencing. It therefore cannot treat `**p`, which can be treated by PathCrawler in the absence of pointer casts.

The constructions above pose the additional difficulty of *aliases*, i.e. different ways to address the same memory location. Some of them (*external aliases*) appear when the allowed inputs of the function under test may address the same memory location in two different ways. For example, in a circular doubly-linked list `dl`, some of `dl->left->`. . . `->left` and `dl->right->`. . . `->right` are aliases. If an input is (or contains) a data structure with aliases, the test generator has to find the shape of the data structure as well as its data values. By default, PathCrawler supposes there are no external aliases, but allows the user to define external alias relations in the precondition.

In functions without external aliases, *internal aliases* are due to instructions inside the function and occur during symbolic execution of a program path with unknown inputs. The difficulty arises from *unknown inputs used as offsets*, e.g. in instructions like `a[i]=5` or `if(max<a[i])`

where `i` is (or depends itself on) an unknown input. Symbolic execution of such instructions will not know where to read or where to write the value of `a[i]`. An original method for treating internal aliases in PathCrawler was proposed in [16]. Its main idea is to delay alias relations occurring in a path until the end of the symbolic execution of the path. CUTE prefers to approximate alias relations, that may make test generation incomplete and slower as shown in [16].

3.6 Treating Floating-Point Numbers

More and more frequently, critical industrial systems use floating-point numbers. The V3F project [3] studied the issues concerning the automatic generation of tests for programs manipulating floating-point numbers.

The arithmetic properties of floating-point numbers are very poor [12]: addition and multiplication are neither associative nor distributive. The limited representation used has some annoying effects, in particular absorption and cancellation. Absorption occurs when a small floating-point number is added to a much bigger one, in this case the addition acts as a null operation. Cancellation results from the subtraction of two neighbouring quantities that may lead to 0.0 even if the quantities were different.

Since the principle of test generation tools is to transform an objective into a constraint system, a naive solution would be to use a constraint solver on real or rational numbers to deal with instructions containing floating-point numbers. Due to semantic differences between floating-point and real arithmetic, this would compromise soundness: a path may be inferred as infeasible on real numbers although there exist floating point input data that satisfy it, and conversely it may have a real solution while there is no solution on floating-point numbers. Some attempts have been made to define heuristic strategies in order to combine constraint solving on real numbers and evaluation on floating-point numbers in [21], but this work has shown that it was better to develop a specific floating-point solver. This kind of solver is based on bounds consistency (interval propagation) and tries to implement precisely the floating-point semantics defined in the IEEE 754 standard, its principles and the rules that it applies are described in details in [5]. PathCrawler uses such a solver, see Section 3.8.

However, floating-point arithmetic is highly context sensitive. In particular, there are still difficulties concerning the particularities of some floating-point units, the optimisations made by the compiler and the use of mathematical libraries. The Intel floating-point units use registers bigger than those required by the standard (80 bits instead of, for example, 64 bits for the double type). Every time a computation is performed and the result stored in the registers, the precision is better than it would be with the standard type. Every time a piece of data is transferred from a register to the memory, a rounding operation is applied. This means that the semantics of the program instructions depends on the register scheduling decided by the compiler. The second difficulty comes from the fact that compilers often make optimisations that are not conservative according to the floating-point semantics. The third difficulty is linked to the fact that only a limited number of operations are standardised in IEEE 754, and for example the precision of transcendental functions is not specified, making their modelling harder. All these points are described in detail in [22], and they are still open issues in the domain of automatic test case generation.

3.7 Extension to C++

In the interests of structuration and reutilisation, much industrial software is now written in C++. Instead of C, automatic test generation tools must thus be able to treat this language. Many C++ constructs are syntactic sugar and can be understood as C constructs. Classes can be seen as structures, and method calls as function calls taking the caller object as an additional parameter. References, which allow direct manipulation of pointers to be avoided, can be translated into addresses of variables and dereferences of pointers. In a similar way to compilers, inheritance has to be flattened out in a simple array, where members of base classes are put at the beginning and members of the class at the end. Templates are also syntactic sugar and it is sufficient to consider instantiated templates.

However there are still some problems with C++ which must be pointed out. In the case where you have an object as parameter, or if you want to test a method, the internal state of the object cannot be randomly chosen but has to respect preconditions. As usual, an object is initialised by a constructor and modified only by its methods. The semantics of exceptions is complicated because in particular it requires recording creations of local objects in order to be able to delete them if an exception occurs. Exceptions in the context of multiple inheritance are even more tricky. The semantics of virtual calls requires a ”virtual method table” in each object, to be read before each call. It also means that for a virtual object as a parameter of the function under test, we have to introduce choices in the execution path which depend on the object type. This type can be seen as a hidden parameter of the function under test.

The extension of PathCrawler to treat C++ is in progress. It is based on ELSA [20] instead of CIL [25] to parse C++ and to manipulate the abstract syntax tree. Our modification for C++ has the two following restrictions: virtual objects as a parameter of the function under test and methods as a function under test are forbidden. These restrictions can be bypassed by defining a wrapper function which constructs a valid object and then calls the desired function or method. As all objects are instantiated, the types of objects are known, and concrete execution of the instrumented program allows to determine which virtual function is called or which exception handler is triggered.

3.8 Efficiency

An important criterion for the evaluation of an automatic test generation tool is its performance. There are two main reasons for the combinatorial explosion of constraint-based test generation. On the one hand, even with a finite number of potential solutions, constraint solving problems can be in general NP-hard, so generating a test case to cover one test objective may take exponential time. An efficient constraint solver and appropriate heuristics may provide a considerable speed-up in many cases. The PathCrawler tool uses COLIBRI, an efficient constraint solver developed at CEA LIST and shared with GATeL [18] and OSMOSE [1] testing tools. This solver is dedicated to program verification and is able to solve linear and non-linear constraints on integer, floating-point and real numbers, with the capacity to speed up reasoning by exploiting congruence and distance relations between variables.

On the other hand, rigorous structural testing often has to cover a very large number of test objectives (cf Section 3.2). Hence, any optimisation of the generation method, multiplied by the number of test objectives, may significantly improve overall performance. In the PathCrawler method for the all-paths criterion, various solutions were found to optimize the test generation process.

Incrementality. Exploring the program paths in a depth-first search allows maximum reuse of the results of symbolic execution. Each instruction of any given partial path π is executed exactly once, independently of how many paths start with π. The longer the program paths are, the more significant the gain in time due to incremental search. This is possible thanks to constraint logic programming, which offers backtracking.

Fast entry. Concrete execution of instrumented code enables a complete feasible path in the program to be quickly deduced for every test case. Compared to symbolic execution of a generated test case, concrete execution of binary code is many times faster and therefore offers a significant speed-up for the complete test generation session.

3.9 Treating Specific Types of Software

The all-paths structural testing criterion must be carefully interpreted in order to apply it to certain types of software. We have already seen in Section 3.2 that this is true even for programs containing loops or function calls. It is even more so for cyclical reactive or multi-tasking software.

Cyclical reactive software is common in embedded systems. It typically memorises a “state” using variables declared in C as “static”. Each time such a function is run, it reads the current value of certain external inputs, represented by global variables or the function parameters, and also that of its own state variables and then modifies the state variables as well as the usual outputs.

To apply path-testing to a single execution of a cyclical reactive function, the state variables must be treated as inputs, because their value can influence the choice of branches. However, such systems have a well-defined initial state (or set of initial states) which then evolves due to the modifications of the state variables at each execution. Many states (i.e. combinations of values of state variables) are not in fact reachable. Path testing must only generate tests which start from a reachable state, in order to ensure that bad test results indicate a problem that can really arise. In theory, the precondition (see Section 3.4) can be used to characterise the reachable combinations of values of the state variables but in practice the user does not usually know the definition of the reachable states.

Another approach is to generate tests which activate a sequence of executions, starting from an initial state. This can be done by generating tests for a loop which calls the function under test a certain number of times and in which the state variables become local variables. However, in this approach the size and complexity of the function under test is greatly increased and it may only be feasible to generate relatively short sequences.

Path testing also needs to be adapted to multi-threaded and other parallel programs. The naive solution of exploring all possible interleavings is not realistic because of the state-explosion problem. However, we can consider instructions to be independent if their execution order does not change the result of the program. This enables us to define equivalence classes of execution paths, which contain the same independent instructions in different orders. We can then test just one execution path in each equivalence class. In [9] the partial order relation between instructions of parallel programs is investigated. A dynamic method is proposed to efficiently compute a subset of executions which represents all equivalence classes. [26] applies this method of automatic test generation for parallel program to the jCUTE tool.

4 Conclusion

The test-input generation method implemented in the PathCrawler tool has proved its effectiveness in the successful generation of test cases covering different paths in numerous examples of C code. However much more is needed before an effective test generation machine can become a useful tool. First, it must be able to treat all the features frequently encountered in the sort of code it will be used on. We have mentioned several of these above, such as floating-point numbers and library function calls. Moreover, once the user has decided how and why structural testing fits into the test process, it must be possible to use the tool in this process. This raises more questions discussed above, such as the extent to which satisfaction of a coverage criterion is guaranteed by the tool and how a test context can be defined. Evidently all tools cannot serve all purposes and other tools may address the issues we raise in other ways. In this article we have identified the criteria which seem the most important in our experience of applying PathCrawler to examples of industrial, embedded software. They are the areas we are currently working on in order to demonstrate that test automation can really be used in an industrial setting.

Acknowledgements. The authors would like to thank Bruno Marre, Benjamin Blanc and Franck Vedrine for developing and maintaining the constraint solver COLIBRI used by PathCrawler.

References

[1] S. Bardin and P. Herrmann. Structural testing of executables. In *ICST'08*, pages 22–31, Lillehammer, Norway, April 2008. IEEE Computer Society.

[2] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA'08*, pages 3–14, Seattle, WA, USA, 2008. ACM.

[3] B. Blanc, F. Bouquet, A. Gotlieb, B. Jeannet, T. Jeron, B. Legeard, B. Marre, C. Michel, and M. Rueher. The V3F project. In *CSTVA'06*, Nantes, France, 2006.

[4] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: attacking path explosion in constraint-based test generation. In *TACAS'08 (Part of ETAPS'08)*, pages 351–366, Budapest, Hungary, March–April 2008.

[5] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations: Research articles. *Softw. Test. Verif. Reliab.*, 16(2):97–121, 2006.

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *ISSTA'02*, pages 123–133, New York, NY, USA, 2002. ACM.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS'06*, pages 322–335, Alexandria, VA, USA, November 2006.

[8] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS'06*, pages 182–196, 2006.

[9] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL'05*, pages 110–121, New York, NY, USA, 2005. ACM Press.

[10] P. Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 42(1):47–54, 2007.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI'05*, pages 213–223, Chicago, IL, USA, June 2005.

[12] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[13] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA'98*, pages 53–62, Clearwater Beach, FL, USA, March 1998.

[14] A. Gotlieb, B. Botella, and M. Watel. INKA : Ten years after the first ideas. In *ICSSEA'06*, Paris, France, December 2006.

[15] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):567–580, 1969.

[16] N. Kosmatov. All-paths test generation for programs with internal aliases. In *ISSRE'08*, pages 147–156, Seattle, WA, USA, November 2008. IEEE Computer Society.

[17] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE'07*, pages 416–426, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[18] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions : GATeL. In *ASE'00*, pages 229–237, Grenoble, France, September 2000.

[19] A. P. Mathur. *Foundations of Software Testing*. Pearson Editions, 2008.

[20] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In E. Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2004.

[21] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *CP'01*, pages 524–538, London, UK, 2001. Springer-Verlag.

[22] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[23] P. Mouy, B. Marre, N. Willams, and P. Le Gall. Generation of all-paths unit test with function calls. In *ICST'08*, pages 32–41, Lillehamer, Norway, 2008. IEEE Computer Society.

[24] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[25] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC'02*, Grenoble, France, 2002.

[26] K. Sen and G. Agha. jCUTE : Automated testing of multithreaded programs using race-detection and flipping. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.

[27] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE'05*, pages 263–272, Lisbon, Portugal, September 2005.

[28] N. Tillmann and J. de Halleux. White box test generation for .NET. In *TAP'08*, volume 4966 of *LNCS*, pages 133–153. Springer, April 2008.

[29] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[30] N. Williams. WCET measurement using modified path testing. In *WCET'05*, Palma de Mallorca, Spain, July 2005.

[31] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In *ASE'04*, pages 290–293, Linz, Austria, September 2004.

[32] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC'05*, pages 281–292, Budapest, Hungary, April 2005.