All-Paths Test Generation for Programs with Internal Aliases

Nikolai Kosmatov
CEA LIST
Software Reliability Laboratory
91191 Gif-sur-Yvette France
Nikolay.Kosmatov@cea.fr

Abstract

In structural testing of programs, the all-paths coverage criterion requires to generate a set of test cases such that every possible execution path of the program under test is executed by one test case. This task becomes very complex in presence of aliases, i.e. different ways to address the same memory location. In practice, the presence of aliases may result in enumeration of possible inputs, generation of several test cases for the same path and/or a failure to generate a test case for some feasible path.

This article presents the problem of aliases in the context of classical depth-first test generation method. We classify aliases into two groups: external aliases, existing already at the entry point of the function under test (due to pointer inputs), and internal ones, created during its symbolic execution. This paper focuses on internal aliases.

We propose an original extension of the depth-first test generation method for C programs with internal aliases. It limits the enumeration of inputs and the generation of superfluous test cases. Initial experiments show that our method can considerably improve the performances of the existing tools on programs with aliases.

1 Introduction

Testing is nowadays the primary way to improve the reliability of software. Software testing accounts for 50% of the total cost of software development. Automated testing is aimed at reducing this cost. The increasing demand has motivated much research on automated software testing. Symbolic execution was proposed by J. C. King [12] in 1976. Constraint-solving techniques are commonly used in software testing since 1990's [5, 8, 16, 9, 17, 21, 22, 7, 18, 23, 2, 3, 1].

Among other novel techniques, various combinations of concrete and symbolic execution were developed during the last five years. They were successfully applied in implementation of several testing tools for C programs: PathCrawler [21, 22], DART [7], CUTE [18], EXE [2]. These techniques appeared to be particularly beneficial in *path-oriented* testing, according to the classification of [6]. For example, *the all-paths test coverage criterion* [26] requires to generate a set of test cases such that every possible execution path of the program under test is executed by one test case. This criterion being very strong and often unreachable, weaker path-oriented criteria [26] were proposed, requiring to cover only paths of limited length, or with limited number of loop iterations, etc. The paths are often explored in depth-first search [22, 7, 18], sometimes in breadth-first search [23] or by mixed heuristics [2].

One of the main difficulties in path-test generation for C-like programs is related to *aliases,* i.e. different ways to address the same memory location. In practice, the presence of aliases may result in non-covered paths (incomplete coverage), enumeration of possible inputs and/or generation of several test cases for the same path. We show that for path-test generation based on symbolic execution, it is convenient to distinguish two kinds of aliases that we call *external* and *internal aliases*. External aliases come into the function under test with its inputs when they contain pointers. Internal aliases are due to the instructions inside the function and occur during symbolic execution of a program path with unknown inputs. Testing programs with external aliases was studied in [19]. This paper focuses on internal aliases. The object of this work is to extend existing path-test generation algorithms for C-like programs to programs with internal aliases and to improve handling of aliases in the current version of PathCrawler.

Complete all-paths test generators can be also adapted to measure the worst-case execution time (WCET) of a program [20]. Recent research showed another possible application of path-oriented testing, in combination with static analysis techniques [14, 24, 10]. So, SYNERGY [10] simultaneously looks for bugs and proofs and tries to put information obtained in one search into the best possible use in the other search. Being less expensive than refinement,

```
1   //maximum in given array
2   int max3(int a[3]){
3     int max=a[0];
4     if( max < a[1] )
5       max=a[1];
6     if( max < a[2] )
7       max=a[2];
8     return max; }
```

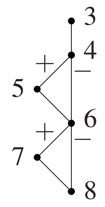


Figure 1. Function max3 (without aliases) returning the maximum in array a, and its CFG

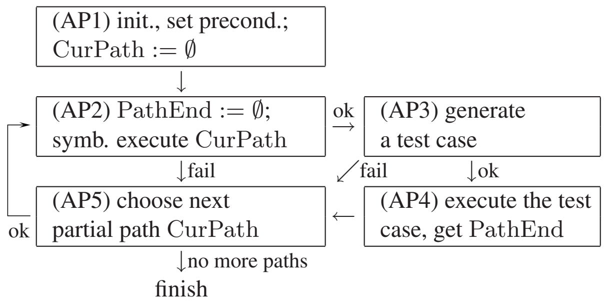

(AP1) init., set precond.; CurPath := ∅
(AP2) PathEnd := ∅; symb. execute CurPath
ok
(AP3) generate a test case
fail
fail
ok
(AP5) choose next partial path CurPath
(AP4) execute the test case, get PathEnd
ok
no more paths
finish

Figure 2. Schema of all-paths algorithm AP

tests give valuable information for choosing the next refinement and therefore contribute into the formal proof. The OSMOSE tool [1] has recently adapted a PathCrawler-like method for testing executables at the binary level, where the presence of aliases gives rise to similar problems. These various applications of path-oriented testing in software engineering give additional motivation to this work.

Different questions related to aliases were studied in *pointer analysis.* We refer the reader to [11] for a survey and more references on pointer analysis. Since it is undecidable, in general, to statically determine the possible runtime values of a pointer [15], a large collection of approximation algorithms were published, e.g. the recent work [25] on alias analysis for C. Unfortunately, static pointer analysis techniques are in general not applicable in all-paths test generation because they are approximate and usually do not take advantage of the constraints of the current path.

This paper makes the following contributions:

Presentation of the problem. In the context of the classical depth-first strategy of automatic all-paths test generation using symbolic execution, we present the difficulties of handling aliases in C functions, and show that it is convenient to classify possible aliases into two groups: external and internal ones, according to their origin (Section 2).

Late-aliases algorithm. We propose a new algorithm of test generation for programs with internal aliases. Collecting alias relations and delaying their application until the end of the (partial) path evaluation allows to avoid superfluous test generation and to limit enumeration. We present a toy implementation of this method in Prolog (Section 3).

Experimental results. We propose an interesting C program with aliases for experimental evaluation of test generators. Its properties allow to appreciate the performance of the generation on programs with lots of possible inputs and infeasible paths, and relatively few feasible paths. We evaluate two existing tools, CUTE and PathCrawler, and compare them to our method. The experiments show that our late-aliases algorithm can considerably improve the existing tools (Section 4).

2 Presentation of the Problem

We give a brief description of a PathCrawler-like method for generation of all-paths tests [21, 22] for programs without aliases. Similar ideas were used in DART [7] and CUTE [18]. Then we present the problem related to the presence of aliases and our classification of aliases.

2.1 All-Paths Test Generation in Depth-First Search without Aliases

PathCrawler tool, developed at CEA LIST, generates all-paths tests for a given C function. It is composed of two main modules. The first module, based on the CIL library [4], transforms the source code into an intermediate format (IF) and creates its instrumented version, such that any execution of the instrumented code prints the execution path. Next, the user can modify default parameters of testing (provide an oracle, a precondition, etc.) and starts the second module, test generator, implemented in Prolog language. It reads the IF version of the source code and adjusted parameters, and generates test cases satisfying the all-paths criterion. PathCrawler uses a powerful home-made constraint solver, recently renamed COLIBRI, developed at CEA LIST and also used by GATEL [16] and OSMOSE [1] testing tools.

Let us now describe the generation algorithm (denoted AP) for programs without aliases and illustrate it on function max3 of Figure 1. Given a 3-element array a, it returns the maximum of its elements. The control-flow graph (CFG) in Figure 1 shows that max3 has four different paths. Figure 2 gives an outline of the algorithm, whereas Figure 3 shows its application to the example. For simplicity of notation, paths will be written as sequences of line numbers.

For symbolic execution of a program in constraints for unknown inputs, the generator maintains

1. a database Mem that represents the program memory at each moment of symbolic execution. Mem can be seen as an association list of pairs $\mathrm{SmbName} \to \mathrm{Val}$, where to a symbolic name $\mathrm{SmbName}$ of a C variable

1) Mem	Constr.		Test 1 / PathEnd
a$[0] \to X_0$	$\langle$precond$\rangle$		$X_0 = 0$
a$[1] \to X_1$		$\mapsto$	$X_1 = 0$
a$[2] \to X_2$			$X_2 = 1$
CurPath: $\langle$empty$\rangle$			$3, 4^-, 6^+, 7, 8$

$\rightarrow$

2) Mem	Constr.		Test 2 / PathEnd
a$[0] \to X_0$	$\langle$precond$\rangle$		$X_0 = 0$
a$[1] \to X_1$	$X_0 \geq X_1$	$\mapsto$	$X_1 = 0$
a$[2] \to X_2$	$X_0 \geq X_2$		$X_2 = 0$
max $\to X_0$			
CurPath: $3, 4^-_{\text{fst}}, 6^-_{\text{snd}}$			...8

$\rightarrow$

3) Mem	Constr.		Test 3 / PathEnd
a$[0] \to X_0$	$\langle$precond$\rangle$		$X_0 = 0$
a$[1] \to X_1$	$X_0 < X_1$	$\mapsto$	$X_1 = 1$
a$[2] \to X_2$			$X_2 = 0$
max $\to X_0$			
CurPath: $3, 4^+_{\text{snd}}$			...$5, 6^-, 8$

$\rightarrow$

4) Mem	Constr.		Test 4 / PathEnd
a$[0] \to X_0$	$\langle$precond$\rangle$		$X_0 = 0$
a$[1] \to X_1$	$X_0 < X_1$	$\mapsto$	$X_1 = 1$
a$[2] \to X_2$	$X_1 < X_2$		$X_2 = 2$
max $\to X_1$			
CurPath:$3, 4^+_{\text{snd}}, 5, 6^+_{\text{snd}}$			...$7, 8$

Figure 3. Depth-first generation of all-paths tests for the function max3 of Figure 1

(an array element, etc.) is associated its current value Val that may be a constant or a logical variable. Mem is efficiently implemented as a hash table.

2. current partial path CurPath in the program under test. For each conditional node θ in the current path

$$\alpha, \beta, \gamma, \ldots, \eta, \theta, \iota, \ldots$$

we store if θ is true or false on this path (denoted in figures by a "$+$" or a "$-$" resp.), and if the partial path $\alpha, \beta, \gamma, \ldots, \eta, \neg\theta$ was already considered by the algorithm or not (denoted "snd" or "fst" resp.).

3. a constraint store (the column Constr. in figures) containing at each moment the constraints added by symbolic execution of current partial path CurPath.

We write *in italic font* the main steps of AP (All-Paths) algorithm in order to separate them from the example.

(AP1) First, create a logical variable for each input and associate it with the input. Set initial values and constraints for the precondition if necessary. Let the current partial path CurPath *be empty. Continue to (AP2).*

In 1) of Figure 3, the logical variable X_1 represents the input variable a$[1]$, and a$[1] \to X_1$ shows that X_1 is the value of a$[1]$ at this moment. This value may change if another value is assigned to a$[1]$. If the precondition of max3 is $0 \leq$ a$[0]$, a$[1]$, a$[2] \leq 5$, then $\langle$precond$\rangle$ in Figure 3 denotes $X_0 \in [0, 5],\ X_1 \in [0, 5],\ X_2 \in [0, 5]$.

(AP2) Let PathEnd *be empty. Execute symbolically* CurPath*, i.e. add constraints and update the memory according to the instructions of the path. If some constraint fails, continue to (AP5). Otherwise, continue to (AP3).*

CurPath being empty, (AP2) adds no constraints here.

(AP3) The constraint solver is called to produce a test case, i.e. concrete values for the inputs, satisfying the current constraints. If it fails, continue to (AP5). Otherwise, continue to (AP4).

It can be shown that the choice of the first test case is not important for a complete depth-first search: we can start from any initial test case, and will make a complete exploration if and only if we make a complete exploration from any other first test case. Sometimes random generation is used (as you see, the author was not a very good random number generator), sometimes deterministic strategies are preferred (e.g. the minimal values tried first).

(AP4) Execute the program on the test case generated in (AP3). The complete executed path must start by CurPath*. Write the remaining part into* PathEnd*. Continue to (AP5).*

PathCrawler uses concrete execution of instrumented code to obtain the path. In Figure 3, $\mapsto$ denotes application of (AP3) and (AP4). Here, (AP3) produces Test1, and (AP4) finds PathEnd $= 3, 4^-, 6^+, 7, 8$.

(AP5) Let AllPath *be the concatenation of* CurPath *with* PathEnd*. Exit if there is no not-yet-negated conditional node in* AllPath*. Otherwise, find in* AllPath *the last conditional node $\theta^{\pm}_{\text{fst}}$ which was not yet negated, the conditional nodes from* PathEnd *being always considered as not yet negated. Set* CurPath *to the subpath of* AllPath *before θ, and add $\theta^{\mp}_{\text{snd}}$. Continue to (AP2).*

In other words, if AllPath is $\alpha, \beta, \gamma, \ldots, \eta, \theta^{\pm}_{\text{fst}}, \iota, \ldots$ where $\theta^{\pm}_{\text{fst}}$ is the last conditional node not yet negated, i.e. the last one marked fst, then negate it and set CurPath to $\alpha, \beta, \gamma, \ldots, \eta, \theta^{\mp}_{\text{snd}}$. This rule ensures depth-first exploration of program paths.

In Figure 3, $\rightarrow$ denotes application of (AP5) and (AP2). We are now moving from 1) and Test1 to 2) in Figure 3. (AP5) sets AllPath $= 3, 4^-_{\text{fst}}, 6^+_{\text{fst}}, 7, 8$ (all conditional nodes from PathEnd are always marked fst), and therefore CurPath $= 3, 4^-_{\text{fst}}, 6^-_{\text{snd}}$. Now (AP2) has to symbolically execute, in constraints, node by node, this path for unknown inputs. The execution of the assignment 3 adds max $\to X_0$ to Mem. The execution of the conditional node 4^-_{fst} adds the constraint $X_0 \geq X_1$. The execution of 6^-_{snd} adds the constraint $X_0 \geq X_2$. We did not detail these intermediate steps in Figure 3 (but they exist and are used in backtracking!).

An evaluation routine is called each time in order to find the current value of an expression (r-value), or the correct symbolic name for the variable being assigned (l-value). Evaluation of complex expressions may introduce additional logical variables and constraints. For example, if we had an assignment $\texttt{a}[2] = \texttt{a}[0] + 3 * \texttt{a}[1]$, its symbolic execution now would add new variables Z, Z', the line $\mathrm{a}[2] \to \mathrm{Z}$ to Mem and, after evaluation of $\mathrm{a}[0]$ to X_0 and $\mathrm{a}[1]$ to X_1, new constraints $\mathrm{Z} = \mathrm{X}_0 + \mathrm{Z}'$, $\mathrm{Z}' = 3\mathrm{X}_1$.

Next, (AP3) generates a new test case Test2 satisfying the current constraints of the constraint store. (AP4) executes the program on Test2 and computes $\mathrm{PathEnd} = 8$.

We are now going from 2) and Test2 to 3) in Figure 3. (AP5) sets $\mathrm{AllPath} = 3, 4^-_{\mathrm{fst}}, 6^-_{\mathrm{snd}}, 8$, where the last not-yet-negated conditional node is 4^-_{fst}, therefore $\mathrm{CurPath} = 3, 4^+_{\mathrm{snd}}$. Next, (AP2) sets the constraint store into the state in which it would be after the symbolic execution of $\mathrm{CurPath}$. We can do it from our initial state 1) by executing the assignment 3 and the conditional node 4^+_{snd}, which will add the constraint $X_0 < X_1$. In practice, backtracking is intelligently used here to come back to the closest state (here it corresponds to $\mathrm{CurPath} = 3$) from which we can reach the destination in minimal number of steps.

Next, (AP3) generates Test3, and (AP4) sets $\mathrm{PathEnd}$.

We are now moving from 3) and Test3 to 4) in Figure 3. (AP5) computes $\mathrm{AllPath} = 3, 4^+_{\mathrm{snd}}, 5, 6^-_{\mathrm{fst}}, 8$, hence $\mathrm{CurPath} = 3, 4^+_{\mathrm{snd}}, 5, 6^+_{\mathrm{snd}}$. We have to set the constraint store into the state in which it would be after the symbolic execution of $\mathrm{CurPath}$. It is reached starting from 3) by execution of the assignment 5, which sets $\max \to X_1$ in Mem, and of the conditional node 6^+_{snd}, which adds the constraint $X_1 < X_2$ after the evaluation of both sides.

Next, (AP3) generates Test4 and computes $\mathrm{PathEnd}$. Finally, (AP4) finds no not-yet-negated conditional node, so the complete exploration of execution paths is finished.

Notice that if during some execution of (AP2) or (AP3), the constraints appear to be unsatisfiable and no test case can be generated, then $\mathrm{CurPath}$ is infeasible and the algorithm goes to (AP5) to continue the exploration of other paths normally. If it happens at the very first iteration of (AP3), that is, the precondition is unsatisfiable, then the algorithm stops at (AP5) because $\mathrm{AllPath}$ is empty.

2.2 What Changes for Aliases? External and Internal Aliases

External aliases. One of the main difficulties in path-test generation for C-like programs is related to *aliases,* i.e. different ways to address the same memory location. The best-known type of aliases appear when the function under test contains pointer inputs and some memory location is reachable by (or starting from) two different pointers contained in inputs. Such aliases exist at the entry point of the function, so we call them *external aliases.* 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. External aliases were studied in detail in [19] where the reader will find more examples and a method of test generation for functions with external aliases.

```
int a[5]={6,7,6,6,7};
int max3Als(int i0,int i1,int i2){
  int max=a[i0];
  if( max < a[i1] )
    max=a[i1];
  if( max < a[i2] )
    max=a[i2];
  return max; }
```

Figure 4. Function max3Als returns the maximum of the given three elements in array a

We focus here on functions without external aliases. This restriction is also made in PathCrawler, where the user may explicitly indicate external aliases. It can be shown that the all-paths test generation problem remains NP-hard even for programs with internal aliases only.

Internal aliases. 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. Unexpectedly, the difficulty is not related to equal-value pointers or data structures with pointers created inside the function. If some program path creates a circular doubly-linked list `dl` of two elements, its symbolic execution will collect all necessary information about the list and will know that `dl->left` and `dl->right` refer to the same memory location. Such aliases will be resolved at evaluation step.

The difficulty arises from *unknown inputs used as offsets.* The curious reader may try to apply the algorithm of Section 2.1 to the program of Figure 4. Indeed, what happens if symbolic execution of $\mathrm{CurPath}$ in our algorithm encounters an assignment $\texttt{a}[\texttt{i1}] = 5$, or $\texttt{max} = \texttt{a}[\texttt{i1}]$, or a condition $\texttt{if}(\texttt{max} < \texttt{a}[\texttt{i1}])$, where `i1` is an input variable that may have different values at this point? Or more generally, if `i1` was assigned a value that is (or depends itself on) some input variable and is not necessarily a constant?

The algorithm of Section 2.1 will not work because, when `i1` is not a constant, the symbolic execution of such instructions will not know where to read or where to write the value of $\texttt{a}[\texttt{i1}]$. In other words, we have a *non-trivial alias* $\texttt{a}[\texttt{i1}]$ for one of the elements of `a`. *Trivial aliases,* such as $\texttt{a}[\texttt{i1} - \texttt{i1}]$ and $\texttt{a}[0]$, are resolved at the evaluation step.

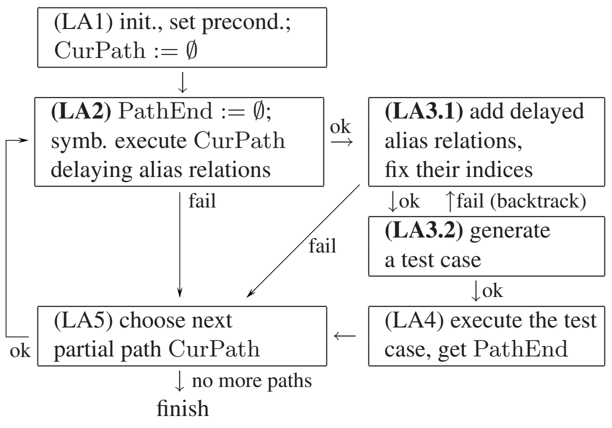

(LA1) init., set precond.; CurPath := ∅
(LA2) PathEnd := ∅; symb. execute CurPath delaying alias relations
ok
(LA3.1) add delayed alias relations, fix their indices
fail
ok
fail (backtrack)
(LA3.2) generate a test case
fail
ok
(LA5) choose next partial path CurPath
(LA4) execute the test case, get PathEnd
ok
no more paths
finish

Figure 5. Schema of late-aliases algorithm LA

To symbolically execute such an instruction, the current version of PathCrawler fixes the value of the index `i1`, e.g. `i1` $= 0$. It means that an additional constraint is added, e.g. $\texttt{X}_1 = 0$, and our constraints do not determine the same domain of input values any more, but a smaller one. In particular, if a longer path appears to be infeasible, we cannot be sure that it would be infeasible with another value of `i1`, e.g. `i1` $= 1$. So, PathCrawler has to enumerate all possible aliases at each point, until test cases for all longer paths are generated, or until the enumeration is finished.

Furthermore, when trying a next value, e.g. `i1` $= 1$, PathCrawler tries to generate test cases for all longer paths again, even if some of them were covered with some previously tested value, e.g. `i1` $= 0$. It results in superfluous constraint solving and test case generation. The other choice would be to maintain the list of infeasible paths, and systematically consult and update this list while trying other possible indices. This will make the algorithm considerably slower and may need much more memory. To avoid these drawbacks, we propose another method in Section 3.

3 Late-Aliases Algorithm

We propose a new algorithm LA of all-paths test generation for programs with internal aliases. Its toy implementation in Prolog language (<500 lines of code including comments, for the ECLiPSe Constraint Programming System[1]) is available at [13]. It was developed to prepare this publication and is intentionally incomplete (otherwise intellectual property issues would not allow the author to publish it). Unlike PathCrawler, it uses symbolic execution of a generated test case instead of concrete execution, which makes it slower. However, it still appears to be more efficient for programs with aliases as we will see in Section 4. Unlike PathCrawler, it uses the *fd* solver provided by ECLiPSe rather than the home-made solver of CEA LIST, so it may be easily tested by the reader. It needs manual translation of a C program into an intermediate format and treats a small fragment of C language containing integers, arrays, conditionals and loops. These restrictions made in order to provide a light and easily understandable version [13] are not crucial for the algorithm itself. Actually, delaying alias relations proposed here is independent of most concepts of C language and may be integrated in a complete tool like PathCrawler. Today, PathCrawler treats all integer arithmetic operations, one-dimensional arrays, function calls (with available source code), structures, pointers (except to functions) and pointer arithmetics, bitwise operations; it does not treat functions with variable number of arguments, floating-point types, some pointer casts, library function calls; multidimensional arrays, recursive functions and types are under development and should work soon. The only limitation of LA is the absence of multiple indirections: we work on a modified algorithm for them.

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

Let us present our late-aliases algorithm (denoted LA). Figure 5 gives an outline of the algorithm, where we write in bold font the numbers of steps whose actions or transitions were modified compared to AP. The application of LA to function max3Als of Figure 4 is detailed in Figure 6. This function, given three indices in the global array a, returns the maximum of these three elements. It has the same CFG as in Figure 1.

In addition to Mem, $\mathrm{CurPath}$ and $\mathrm{Constr.}$ described in Section 2.1, LA maintains a list of alias relations $\mathrm{AlsRlns}$ that can be seen as delayed constraints, not yet added to the constraint store. This list may contain relations of two types: $\mathrm{Eq(a[U], V)}$ represents the delayed equality $\mathrm{a[U] = V}$, and $\mathrm{Aff(a[U], V)}$ represents the delayed assignment (in French, "Affectation") of V to $\mathrm{a[U]}$. In both cases, the index U may be a logical variable, not yet instantiated.

The first step of LA is essentially the same as in AP:

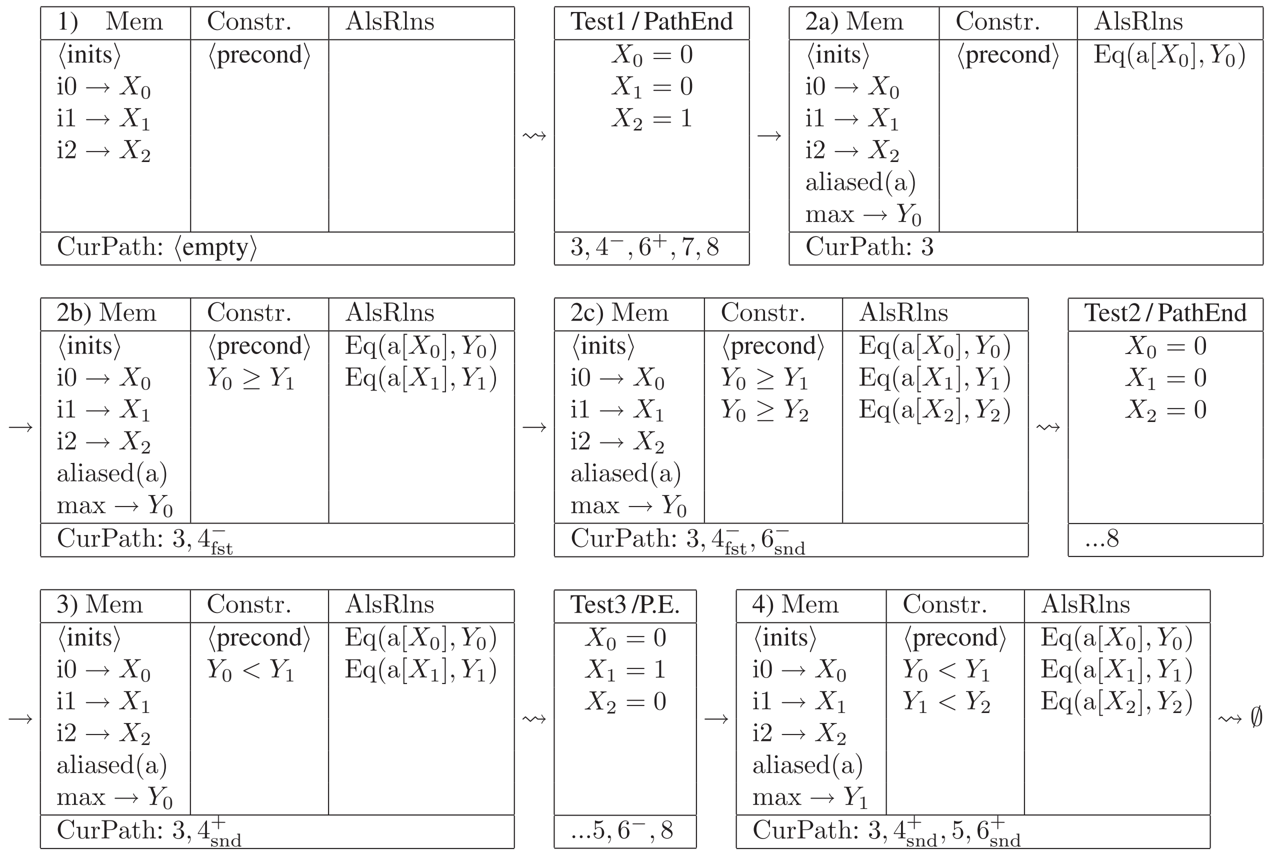
(LA1) First, create a logical variable for each input and associate it with the input. Set initial values and constraints for the precondition if necessary. Let the current partial path $\mathrm{CurPath}$ *and the delayed alias relation list* $\mathrm{AlsRlns}$ *be empty. Continue to (LA2).*

We define the precondition of max3Als for the array indices: $0 \le \mathrm{i0, i1, i2} \le 4$. Then $\langle\text{precond}\rangle$ in Figure 6 denotes the constraints: $X_0 \in [0,4]$, $X_1 \in [0,4]$, $X_2 \in [0,4]$. $\langle\text{inits}\rangle$ denotes $\mathrm{a[0]} \to 6, \ldots, \mathrm{a[4]} \to 7$.

(LA2) Let $\mathrm{PathEnd}$ *and* $\mathrm{AlsRlns}$ *be empty. Execute symbolically* $\mathrm{CurPath}$*, and add delayed alias relations to* $\mathrm{AlsRlns}$ *respecting their order. If some constraint fails, continue to (LA5). Otherwise, continue to (LA3.1).*

As in AP, the first symbolic execution in (LA2) is always trivial since $\mathrm{CurPath}$ is empty. We are at 1) of Figure 6.

(LA3.1) If $\mathrm{AlsRlns}$ *is empty, continue to (LA3.2). Other-*



1) Mem | Constr. | AlsRlns
⟨inits⟩ i0 → X_0 i1 → X_1 i2 → X_2 | ⟨precond⟩ |
CurPath: ⟨empty⟩
⇝
Test1 / PathEnd
$X_0 = 0$ $X_1 = 0$ $X_2 = 1$
$3, 4^-, 6^+, 7, 8$
→
2a) Mem | Constr. | AlsRlns
⟨inits⟩ i0 → X_0 i1 → X_1 i2 → X_2 aliased(a) max → Y_0 | ⟨precond⟩ | Eq(a[X_0], Y_0)
CurPath: 3
→
2b) Mem | Constr. | AlsRlns
⟨inits⟩ i0 → X_0 i1 → X_1 i2 → X_2 aliased(a) max → Y_0 | ⟨precond⟩ $Y_0 \geq Y_1$ | Eq(a[X_0], Y_0) Eq(a[X_1], Y_1)
CurPath: $3, 4^-_{\text{fst}}$
→
2c) Mem | Constr. | AlsRlns
⟨inits⟩ i0 → X_0 i1 → X_1 i2 → X_2 aliased(a) max → Y_0 | ⟨precond⟩ $Y_0 \geq Y_1$ $Y_0 \geq Y_2$ | Eq(a[X_0], Y_0) Eq(a[X_1], Y_1) Eq(a[X_2], Y_2)
CurPath: $3, 4^-_{\text{fst}}, 6^-_{\text{snd}}$
⇝
Test2 / PathEnd
$X_0 = 0$ $X_1 = 0$ $X_2 = 0$
...8
→
3) Mem | Constr. | AlsRlns
⟨inits⟩ i0 → X_0 i1 → X_1 i2 → X_2 aliased(a) max → Y_0 | ⟨precond⟩ $Y_0 < Y_1$ | Eq(a[X_0], Y_0) Eq(a[X_1], Y_1)
CurPath: $3, 4^+_{\text{snd}}$
⇝
Test3 /P.E.
$X_0 = 0$ $X_1 = 1$ $X_2 = 0$
$...5, 6^-, 8$
→
4) Mem | Constr. | AlsRlns
⟨inits⟩ i0 → X_0 i1 → X_1 i2 → X_2 aliased(a) max → Y_1 | ⟨precond⟩ $Y_0 < Y_1$ $Y_1 < Y_2$ | Eq(a[X_0], Y_0) Eq(a[X_1], Y_1) Eq(a[X_2], Y_2)
CurPath: $3, 4^+_{\text{snd}}, 5, 6^+_{\text{snd}}$
⇝ ∅

Figure 6. Depth-first generation of all-paths tests for the function max3Als of Figure 4

wise remove the next alias relation R *from* AlsRlns. *Generate the next possible value for the index* U *in* R. *If an index value is generated, try to set the delayed alias constraint: add the assignment of* V *to* a[U] *if* R = Aff(a[U], V)*, or the equality constraint* a[U] = V *if* R = Eq(a[U], V)*, and continue (LA3.1) for the next alias relation. If the solver fails to generate an index value or to add a constraint, backtrack in the usual way to generate another index value or, when impossible, to previous alias relations to change previous indices, and continue. After trying all possibilities without a successful test case generation, continue to (LA5).*

The object of (LA3.1) is to find a combination of values for indices in the delayed alias relations such that the constraints for these relations may be added and a test case satisfying the complete constraint store may be generated. Notice that the delayed alias relations should be added in the same order in which they were created, so we start by fixing the index in the first relation in AlsRlns. The index being fixed, we can symbolically execute this relation: add the corresponding constraint or assignment. Then we continue in the same way for the following relations in AlsRlns. Classical backtracking is used to come back and to try other combinations of indices, until a test case is successfully generated (by (LA3.2)), or until all combinations are tested but no test case exists (CurPath is infeasible).

AlsRlns being empty after a trivial (LA2), the first execution of (LA3.1) is always trivial and continues to (LA3.2). Non-trivial steps (LA2) and (LA3.1) will appear below.

(LA3.2) The constraint solver is called to produce a test case. If it fails, backtrack in the usual way to index choices in (LA3.1) in order to try other possible indices. If a test case is generated, continue to (LA4).

Unlike in AP, when test case generation fails, LA does not continue to explore other possible paths. Indeed, we cannot immediately deduce that CurPath is infeasible, since the alias relations were added only for one combination of indices. We have to backtrack into (LA3.1) to try other combinations of indices, and to generate a test case for them. Only after trying all possible combinations without successful test case generation, we know that CurPath is infeasible, so (LA3.1) goes to (LA5) to explore other paths.

(LA4) As in (AP4), execute the program on the generated test case, store the executed PathEnd. *Continue to (LA5).*

In Figure 6, ⇝ denotes Steps (LA3.1), (LA3.2) and (LA4). (LA3.2) generates Test1, and (LA4) executes it.

(LA5) Define CurPath *as in (AP5). Continue to (LA2).*

We are moving from 1) and Test1 to 2a), 2b), 2c) in Figure 3. (LA5) computes the next $\mathrm{CurPath} = 3, 4^-_{\mathrm{fst}}, 6^-_{\mathrm{snd}}$. Next, (LA2) has to symbolically execute this path.

In general, symbolic evaluation of some r-value (resp., l-value) a[i] adds a delayed alias relation Eq (resp., Aff) into AlsRlns if i is evaluated to a variable, or if a is already marked as *aliased* (even if i is a constant). In the first case, the whole array becomes *aliased*, denoted by $\mathrm{aliased}(a)$ in Mem: at least one reading/writing operation in a is delayed, therefore its contents cannot be read/written before this delayed operation is executed. Hence, after delaying the first aliased operation on an array, all the following operations are delayed as well (the second case).

Here, the symbolic execution of the assignment 3 has to evaluate a[i0] with unknown value X_0 of i0. In this case, mark a as *aliased*, create a new variable Y_0 representing the value of a[i0], add $\max \rightarrow Y_0$ to Mem to represent the assignment 3, and add $\mathrm{Eq}(a[X_0], Y_0)$ to AlsRlns to delay the equality $a[i0] = Y_0$, as shown by 2a) in Figure 3. The symbolic execution of 4^-, i.e. the condition $\max \geq a[i1]$, evaluates max to Y_0 and has to evaluate a[i1] with unknown value X_1 of i1. In this case, the array a being already marked as aliased, create a new variable Y_1 representing the value of a[i1], add a new constraint $Y_0 \geq Y_1$, and add $\mathrm{Eq}(a[X_1], Y_1)$ to AlsRlns (cf 2b) in Figure 3). 2c) shows the symbolic execution of the conditional node 6^-.

Similarly, assignments are delayed using Aff. For example, array a being aliased, if we had an assignment $\mathtt{a[0]} = \mathtt{3} * \mathtt{a[1]}$, its symbolic execution now would add new variables Z_1, Z_0, a new constraint $Z_0 = 3Z_1$ and delayed relations $\mathrm{Eq}(a[1], Z_1)$, $\mathrm{Aff}(a[0], Z_0)$ to AlsRlns.

Let us now move from 2c) to Test2 in Figure 3. (LA3.1) treats the delayed alias relations. For $\mathrm{Eq}(a[X_0], Y_0)$, it tries the index value $X_0 = 0$ and, since $a[0] = 6$, adds the constraint $Y_0 = 6$. For $\mathrm{Eq}(a[X_1], Y_1)$, it tries $X_1 = 0$ and sets $Y_1 = 6$. Next, for $\mathrm{Eq}(a[X_2], Y_2)$, it tries $X_2 = 0$ and sets $Y_2 = 6$. All inputs being instantiated and the constraints being satisfied, (LA3.2) trivially produces Test2.

Moving from 2c) and Test2 to 3) in Figure 3 being analogous to previous steps, let us now go from 3) to Test3. (LA3.1) treats first $\mathrm{Eq}(a[X_0], Y_0)$, tries $X_0 = 0$ and sets $Y_0 = 6$. For $\mathrm{Eq}(a[X_1], Y_1)$, it tries $X_1 = 0$ and sets $Y_1 = 6$, which is incompatible with $6 = Y_0 < Y_1$ and fails. The algorithm backtracks to previous index choices, tries $X_1 = 1$ and, since $a[1] = 7$, successfully sets $Y_1 = 7$.

Finally, let us explain the last step after 4) in Figure 3. It is clear that Test4 cannot be generated because $a[i0] < a[i1] < a[i2]$ is unsatisfiable in this array with only two different elements. Here, (LA3.1) needs only 49 failures for partial index combinations to see that all $5 \cdot 5 \cdot 5 = 125$ full combinations are impossible. Our backtrack-based algorithm allows to detect the unsatisfiability for most combinations early due to previously added constraints. It avoids the complete enumeration.

```
#define N 5 // number of elements
typedef int perm[N];
int getOrder(perm p){
  // p is a permutation of 0,...,N-1
  int i, order=1, isId;
  perm power, tmp;
  for( i=0; i<N; i++ )
    power[i]=p[i];
  while(1){
    for(i=0,isId=1; i<N && isId; i++)
      if( power[i] != i ) isId=0;
    if( isId ) return order;
    for( i=0; i<N; i++ )
      tmp[i]=power[i];
    for( i=0; i<N; i++ )
      power[i] = tmp[p[i]];
    order++;
  }
}
```

Figure 7. getOrder calculates the order of p.

4 Experiments

In this section, we compare our late-aliases generation algorithm LA with CUTE and the current version of PathCrawler. The experiments were made on a laptop Intel Core 2 Duo with 1GB RAM. On the function max3Als of Figure 4, LA generates one test case for each of the three feasible paths. CUTE generates only one test case (2,1,0) and covers only one path: $3, 4^+, 5, 6^-, 8$. CUTE approximates path constraints for internal aliases, so it cannot guarantee that the generated test case will execute the desired path, and fails to cover some paths. PathCrawler generates 125 test cases and covers the three feasible paths (77, 30 and 18 times). PathCrawler fixes the index of an alias relation in a classical backtrack-based search as soon as it meets the relation, so it has to try all other index values after backtracking (cf Section 2.2).

Let us now evaluate our late-aliases algorithm using the program in Figure 7 depending on a parameter N. This program is an excellent example for evaluation of algorithms of all-paths testing with aliases. It has a very big number of operations with aliases and infeasible paths, a factorially increasing number of possibles inputs and a much smaller number of feasible paths.

4.1 Example Description

Let us recall some notation and facts known from algebra. We denote the set of permutations of $\{0, ..., N-1\}$ by

Order k	1	2	3	4	5	6
Inputs of order k	1	25	20	30	24	20
Paths of order k	1	4	3	2	1	5

Figure 8. Number of permutations and feasible paths for possible orders for $N = 5$.

S_N. We can multiply permutations since the composition of two permutations $p_1, p_2 \in S_N$ is a permutation again: $(p_1 p_2)(i) = (p_1(p_2(i)),\ i \in \{0, ..., N-1\}$. For any permutation $p \in S_N$, we can find the least integer $k \geq 1$ such that p^k is the identity map. This k is called *the order of* p. For example, the following $p \in S_5$ is of order 6:

$$p: \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 2 & 0 & 4 & 3 \end{array}$$

We include into the precondition of function getOrder of Figure 7 that p is a permutation of $\{0, ..., N-1\}$. Note that CUTE, PathCrawler and our toy implementation allow to add the precondition to the constraint store at the beginning of the generation, so only test cases satisfying the precondition may be generated.

The function getOrder calculates the order of p. Its main loop progressively computes the powers p^l for $l = 1, 2, \ldots$ and compares them to the identity map. The variables `order` and `power` contain the current value l and the last computed power p^l. They are initialized to 1 and p respectively (lines 5–8). Then the program enters the main loop (lines 9–18). It checks first (lines 10–11) if the current `power` is the identity and sets the flag `isId` if it is. If `isId` is true, the function returns the current `order` which contains the order of p (line 12). Otherwise, the current `power` is stored in `tmp` (lines 13–14) and the next power is computed using the formula $p^{l+1}(i) = p^l(p(i))$ (lines 15–16). Then the value of `order` is incremented (line 17).

4.2 Analysis of Results

The results are shown in Figure 9. The first three columns contain the value of N, the number of possible inputs (satisfying the precondition) and the number of feasible execution paths, respectively. Since $|S_N| = N! = 1 \cdot 2 \cdot \ldots \cdot (N-1) \cdot N$, we have $N!$ allowed inputs for each N. The number of feasible paths is much less. At first glance, these numbers may seem strange to the curious reader, so we explain them in more detail in the next paragraph.

Take $N = 5$, then we have $5! = 120$ permutations (allowed inputs), but only 16 feasible execution paths. This is illustrated by Figure 8. Its first line contains the possible orders for $p \in S_5$. The second line shows the number of permutations of each order k. The last line shows the number of different feasible paths they execute. For example, there are 20 possible inputs of order $k = 3$, but these 20 inputs activate only 3 different execution paths in the function getOrder. Although the main loop (line 9) is entered exactly $k = 3$ times for these 20 inputs, the number of iterations of the loop in lines 10–11 depends on the input. Here, for $k = 3$, the flag `isId` may become false and force the loop at line 10 to exit when $i = 0$ or 1 or 2. Hence the three possible paths correspond to the following three cases: the first element of p moves ($p(0) \neq 0$); the second element moves but not the first one ($p(0) = 0,\ p(1) \neq 1$); the third element moves while the first and the second ones do not ($p(0) = 0,\ p(1) = 1,\ p(2) \neq 2$). The situation $p(0) = 0,\ p(1) = 1,\ p(2) = 2$ does not occur for a permutation $p \in S_5$ of order 3. In the same way we may compute the number of feasible paths for other values of N and k, though the conditions become more complex to write for composite numbers k. Notice also that this program has a big number of infeasible paths (most of the paths feasible for a bigger N are not feasible for a smaller N).

The next columns in Figure 9 show the test generation results obtained by CUTE, the current algorithm of PathCrawler and our late-aliases method. For each method, the columns "tests", "cover." and "time" show the number of (different) generated test cases, the path coverage and total execution time, respectively. The coverage indicates the percentage of paths covered.

For CUTE, we also indicate the exact number of paths covered (since its coverage is not always complete) and the number of reported failures. CUTE may fail to correctly predict the path for a test case and may generate test cases for already covered paths because it approximates exact path constraints for aliases. For example, for $N = 5$, CUTE executed the function under test $33 + 21 = 54$ times, but there were only 33 different test cases while the other 21 turned out to be repetitions due to failures. We interrupted the generation when it took more than 12 hours. We report here partial results (followed by "?") and time (denoted by ">12h") of this incomplete generation session.

Path coverage. The results show that CUTE does not meet the all-paths criterion already for $N \geq 5$ (programs with ≥ 16 paths), and its path coverage seems to become worse and worse for greater N. PathCrawler and our late-aliases algorithm activate all feasible execution paths, so the all-paths test generation is complete.

Superfluous test cases. CUTE generates some superfluous test cases (the same test cases several times). The current version of PathCrawler generates many superfluous test cases (in this example, it generates all possible permutations). For example, for $N = 5$, it generates up to 24 test cases for the same path (cf Figure 8), with the average of 7,5 test cases per path, and this average grows very quickly

N	Inputs	Paths	CUTE					PathCrawler			Late aliases		
			tests	fail.	paths	cover.	time	tests	cover.	time	tests	cover.	time
3	6	4	6	2	4	100%	0.17s	6	100%	0.49s	4	100%	0.18s
4	24	7	15	8	7	100%	2.33s	24	100%	0.60s	7	100%	0.21s
5	120	16	33	21	13	81%	2m31s	120	100%	1.75s	16	100%	0.31s
6	720	30	63	44	21	70%	4h17m	720	100%	9.59s	30	100%	1.26s
7	5040	62	7 ?	5 ?	6 ?	10% ?	>12h	5040	100%	1m44s	62	100%	39.3s
8	40320	110	1 ?	– ?	1 ?	1% ?	>12h	40320	100%	23m6s	110	100%	11m48s

Figure 9. Test generation results on the function getOrder of Figure 7 for different N.

with N. Our late-aliases algorithm generates exactly one test case per feasible path.

Efficiency. CUTE is dramatically slower than the other two methods. It is about 80 times slower than PathCrawler for $N = 5$ (program with 16 paths) and thousands of times slower for $N \geq 6$ (programs with ≥ 30 paths). The late-aliases method is the fastest algorithm. It is considerably faster than the current PathCrawler's algorithm: 2–7 times faster depending on the value of N.

5 Related Work and Discussion

Some of numerous papers on symbolic execution and test generation were mentioned in Section 1. For a path in a program with pointer inputs (e.g. different types of linked lists, graphs, trees) that may contain *external aliases*, [19] proposes a two-phase algorithm that generates first the least restrictive shape of the input data structure, and second the data values of the test case. Our contribution is complementary to [19]: we consider *internal aliases*, while [19] focuses on *external ones*. Separating data shape and data value generation provides a speedup in both cases, but the algorithms are quite different since the origin of aliases is different.

PathCrawler proceeds in a classical backtrack-based search and fixes the index as soon as it meets an operation with an internal alias. This method is complete, but generates many superfluous test cases and takes more time. Handling aliases in CUTE [18, § 4.1.6] is incomplete: it approximates path constraints to accelerate the generation. Figure 8 shows that simplifying alias relations in CUTE decreases its path coverage, however it does not necessarily make it faster. Our motivation (cf Section 1) encourages us to search for complete algorithms.

The authors of [18, § 4.1.6] believe that charging the constraint solver with exact path constraints arising for aliases would result in intractable test generation. This opinion is confirmed by some experiments with the very powerful memory model and propagation algorithms proposed in [3], which are unable to treat real-sized programs. Their propagators are very expensive, and the number of variables increases very fast because an aliased operation on an array causes all array elements to be duplicated by fresh variables. We avoid this problem in LA: an alias relation (introducing only one new variable) is delayed and added at the end of the symbolic execution of the path after fixing the index.

[2] reports on an efficient bug-finding tool called EXE where path constraints are solved by STP, a SAT-based solver with bitvectors. Array constraints of [2] are incompatible with multiple indirections: EXE always approximates double pointers and will be incomplete on a simple program like `int main(int argc, char ** argv){...}`. Although EXE's objectives and search heuristics are different from those studied in this paper, it would be interesting to explore the possibilities to apply EXE-like methods to depth-first all-paths test generation with aliases (we did not yet manage to obtain EXE for experiments).

6 Conclusion and Future Work

In the context of classical depth-first path test generation for C-like programs, it is convenient to classify aliases into two groups: internal and external ones. External aliases are due to pointer inputs. They were considered in [19].

This paper focused on internal aliases, i.e. those appearing during symbolic execution of the function under test with unknown inputs. We think that fine dosing of propagation and enumeration is necessary for efficient test generation for internal aliases. We proposed a new algorithm delaying the application of alias relations until the end of the path evaluation. A toy implementation in Prolog language allowed us to compare our algorithm to PathCrawler, testing tool developed at CEA LIST, and to CUTE, the only publically available tool. Our experiments show that this method may considerably improve the performances of existing tools on programs with aliases.

We are now developing an extension of our late-aliases method applicable to programs with multiple indirections (double pointers and more). Future work includes further experiments with late-aliases algorithm, its evaluation in comparison to other tools (if they become available to the author) and its integration into PathCrawler. Another research direction is to find a suitable combination of our method for internal aliases and that of [19] for external ones in order to treat all types of aliases in C programs.

Acknowledgments. The author would like to thank Sébastien Bardin, Bernard Botella, Mickaël Delahaye, Arnaud Gotlieb, Philippe Herrmann, Bruno Marre and Nicky Williams for useful discussions, as well as Darko Marinov and the anonymous referees for many helpful comments.

References

[1] S. Bardin and P. Herrmann. Structural testing of executables. In *the First IEEE International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 22–31, Lillehammer, Norway, April 2008.

[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 322–335, Alexandria, Virginia, USA, November 2006.

[3] F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. In *Testing: Academic and Industrial Conference, Practice and Research Techniques (TAIC-PART'07)*, Windsor, UK, September 2007.

[4] CIL: C Intermediate Language. *Infrastructure for C Program Analysis and Transformation*. 2005/2006. http://manju.cs.berkeley.edu/cil/.

[5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering (TSE)*, 17(9):900–910, 1991.

[6] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223, Chicago, IL, USA, June 2005.

[8] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *the ACM SIGSOFT 1998 International Symposium on Software Testing and Analysis (ISSTA'98)*, pages 53–62, Clearwater Beach, Florida, USA, March 1998.

[9] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. *Lecture Notes in Computer Science*, 1861:399–413, July 2000.

[10] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, pages 117–127, Portland, Oregon, USA, November 2006.

[11] M. Hind. Pointer analysis: haven't we solved this problem yet? In *the ACM SIGPLAN-SIGSOFT 2001 Workshop on Program Analysis For Software Tools and Engineering (PASTE'01)*, pages 54–61, Snowbird, Utah, USA, June 2001.

[12] J. C. King. Symbolic execution and program testing. *Communications of the ACM (CACM)*, 19(7):385–394, 1976.

[13] N. Kosmatov. A toy implementation of late-aliases algorithm for all-paths test generation for C programs with internal aliases, May 2008. http://frama-c.cea.fr/test/aliases/index.htm.

[14] D. Kröning, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via program execution. In *the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, pages 224–238, Seattle, WA, USA, November 2004.

[15] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[16] B. Marre and A. Arnould. Test sequences generation from Lustre descriptions : GATeL. In *the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 229–237, Grenoble, France, September 2000.

[17] C. Meudec. ATGen : automatic test data generation using constraint logic programming and symbolic execution. *Software Testing Verification and Reliability*, 11(2):81–96, June 2001.

[18] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *the 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272, Lisbon, Portugal, September 2005.

[19] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 149, Edinburgh, Scotland, UK, September 2002.

[20] N. Williams. WCET measurement using modified path testing. In *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, July 2005.

[21] 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 *the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 290–293, Linz, Austria, September 2004.

[22] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *the 5th European Dependable Computing Conference (EDCC'05)*, pages 281–292, Budapest, Hungary, April 2005.

[23] Z. Xu and J. Zhang. A test data generation tool for unit testing of C programs. In *the 6th International Conference on Quality Software (QSIC'06)*, pages 107–116, Beijing, China, October 2006.

[24] G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *ACM/SIGSOFT 2006 International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 145–156, Portland, Maine, USA, July 2006.

[25] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 197–208, San Francisco, California, USA, January 2008.

[26] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.