2009 Ninth International Conference on Quality Software

Towards Scalable Compositional Test Generation

Tao Sun, Zheng Wang, Geguang Pu, Xiao Yu
Shanghai Key Laboratory of Trustworthy Computing
East China Normal University
Shanghai, China, 200062
{*suntao,wangzheng,ggpu,lesteryu*}*@sei.ecnu.edu.cn*

Zongyan Qiu
School of Math.
Peking University
Beijing, China, 100871
qzy@math.pku.edu.cn

Bin Gu
Beijing Institute of Control Engineering
Beijing, China, 100080
gubin88@yahoo.com.cn

Abstract

One difficulty of automated test case generation is to deal with compositional units that brings in compositional space explosion of program states. We present a new dynamic execution framework which analyzes program behaviors dynamically for automatic test inputs generation. We utilize forward slicing to explore those functions affecting conditional predicates in program under test. The functions that do not affect the conditional predicates are not in need of being analyzed symbolically. Pointer alias analysis is adopted to make slicing in the presence of pointers more precise. A dynamic partial execution technique is proposed to accelerate the speed of searching the unit space. The proposed approach can be applied to real programs and the experiments are also very encouraging.

Keywords

Program Slicing, Compositional Testing, Partial Execution, Automatic Test Generation

1. Introduction

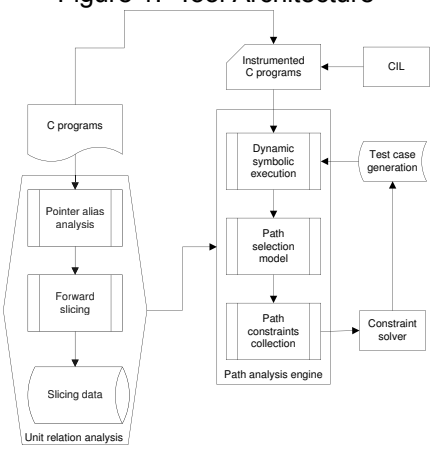
Automated generation of test cases [11], [16] is very attractive. A number of approaches have been proposed towards automatic test data generation. Random testing is a simple strategy that can actually be used to generate numerous input values [5], [9], [21], but it is hard to control the generation of redundant test data or the coverage of program behaviors. Symbolic execution [17] is another well known program analysis technique for automating test cases in several tools, e.g. DART [13], CUTE [22]. But real programs usually lead to the compositional space explosion which is quite similar with the case in model checking [10], [15].

The most of efforts of automatic testing are spent on the symbolic analysis. Our work is trying to reduce the cost on symbolic analysis. The basic idea is: we will only do the symbolic analysis on those called functions which may affect the path selection in the unit under test. Moreover, we develop a dynamic partial execution technique which searches only the necessary part of the space instead of the whole one. The main contributions of our work are as follows:

- Detect the effect on path selection by called functions using forward slicing technique
- Use dynamic partial execution for called functions to improve the performance of test case generation further

We have implemented the analysis techniques for compositional testing and integrated it into the tool named CAUT [24]. CAUT can detect many standard run-time errors and assertion violations. This extension makes it scalable and applicable to real compositional programs.

Figure 1. Tool Architecture



2. Overview

The framework of CAUT [1] is based on dynamic symbolic execution, which is similar with CUTE and DART [22], [13], with its architecture shown in Figure 1. The main difference between CAUT and others is the path selection mechanism.

1. available at http://caut.lab205.org

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.53

Figure 2. Dependence graph of fib

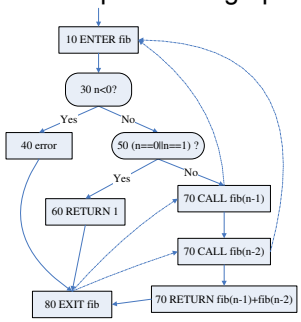
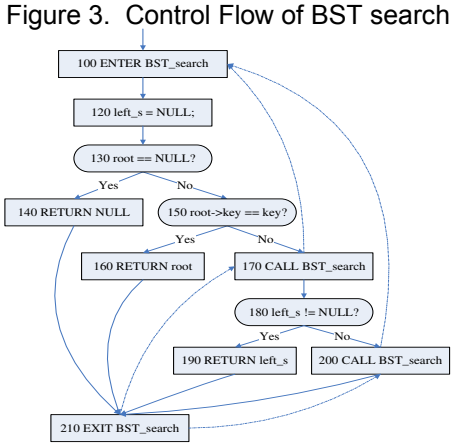


Figure 3. Control Flow of BST search



2.1. Motivating Examples

Dealing with compositional functions is a challenge for automated test case generation while the symbolic analysis is a highly time-consuming job. We develop two methods to reduce the burden of symbolic analysis for this compositional testing.

The first method is *dependence analysis*, which detects whether the called units probably affect the path selection in the unit to be tested. If not, they are not necessarily analyzed by symbolic execution. Here is an example:

```
int fib (int n) {
    if (n < 0)
        error;
    else if (n == 0 || n == 1)
        return 1;
    else return fib(n - 1) + fib(n - 2);
}
```

In order to cover all the paths of fib, four test cases should be generated : $n < 0$, $n == 0$, $n == 1$ and $n > 1$. The dependence graph of this example is illustrated in Figure 2, where the units recursively called do not affect the conditional nodes (round rectangles) at all.

To handle the dependence analysis, we use the forward slicing technique [25], [4] to slice the variables which are likely to affect the future paths in the unit under test. Besides, pointer alias model is introduced to make the implicit affect clear during analysis.

The second method we developed is called *dynamic partial execution*. As discussed previously, if there are some called functions which may influence the path decision in the unit under test, we have to trace the symbolic values into this function. The basic idea of the dynamic partial execution is: instead of exploring the whole path space of called function, a simple algorithm is developed to traverse the path space of the called function partially. Here is an example: when using the partial execution technique, the symbolic analysis of this example will only trace two-level recursive calls at node 170 in Figure 3 and terminate while all the branches in BST_search are covered. This certainly reduces the time cost of symbolic analysis on target space.

```
struct BST {
    int key;
    struct BST *left;
    struct BST *right;
    ...
};

struct BST *
BST_search(struct BST *root, int key) {
    struct BST *left_s = NULL;
    if (root == NULL)
        return NULL;
    else if (root->key == key)
        return root;
    left_s = BST_search(root->left,key);
    if (left_s != NULL)
        return left_s;
    else return BST_search(root->right,
                                    key);
}
```

3. Dependence Analysis

Towards symbolic execution, the functional dependence introduces a new problem: whether and how to execute symbolically the functions called by the unit under test. These called functions which do not affect the branch predicates are not necessarily to be carried out the symbolic analysis. In this section, a solution is provided even facing with pointer variables.

3.1. Pointer Alias Model

With the source code having been normalized by CIL, pointer alias is the major issue while dealing with the depen-

Table 1. Scope Analysis Algorithm

```
analyze_scope(b)
//input: b(block of statements)
if is_instructions(b) then
    foreach s in b analyze_statement(s);
else if is_loop(b) then
    bd = get_loop_body(b);
    analyze_scope(bd);
    analyze_scope(bd); //should be executed twice!
else if is_branch(b) then
    ifb = get_if_branch(b);
    elseb = get_else_branch(b);
    analyze_scope(ifb);
    analyze_scope(elseb);
```

dence analysis. Moreover, the type cast of pointers should also be considered towards this problem. We adopt the point-to relationship to represent the relations. Meanwhile, a set denoted as PSA for these point-to relationships will be maintained to keep a track of pointer states.

- $\mathsf{p} = \&\mathsf{v}$, where pointer variable p obtains the memory address of variable v and it makes p point to v.
- $\mathsf{p1} = \mathsf{p2}$, where p1 and p2 are both of pointer type, otherwise it will be ignored.
- $\mathsf{v} = (\mathsf{type})\mathsf{p}$. This occasion happens due to the the type cast, when the types of variable v and p are different.

We can define the level of pointers formally to keep records of pointer types. For instance, $(\mathsf{int} * **)$ type will be regarded as 3-level type pointer.

With the help of PSA constructed by the above rules, pointer states on each point can be recorded. Alias can be picked out from the alias set PSA easily.

3.2. Forward Slicing with Pointer Alias

Variables in conditional or loop branches are responsible for determining the path selection of program testing. Thus, if those variables in branches are affected by function calls, these called functions should be carried out in the symbolic way. This approach considers both branch statements and the function calls as the program slicing criterion (direction) to decide whether to apply symbolic analysis for them.

- Active set A, it keeps tracking variables modified by function calls directly or indirectly.
- Definition set def, it contains the defined variables in a statement.
- Reference set ref, its variables are the ones that are referred in an statement.

The analysis work is manipulated through scanning the source code of the function under test scope by scope. The algorithm on scope analysis is presented in table 1. In this static analysis, both branches of branch scopes are supposed to be explored, while the loop scopes should be processed twice because the first execution may affect the next one.

Table 2. Forward Slicing Algorithm

```
analyze_statement(s, pss)
//input: s (statement)
rst = true;
pss = analyze_pointer_status(s);
if is_instruction(s) then
    if is_function_call(s) then
        if !created then
            a = create_active_set(); created := true;
            a = get_side_effect(s);
        else ref(s) =ref(s) ∪ find_pointer_content(ref(s), pss);
        if is_empty(ref(s) ∩ a) then
            a = a ∪ def(s);
        else a = a - def(s);
else if is_branch(s) then
    ref(s) =ref(s) ∪ find_pointer_content(ref(s), pss);
    if is_empty(a ∩ ref(s)) then rst = rst && false;
    else rst = rst && true;
return rst;
```

The forward slicing shown in Table 2 is actually carried on statements and is the key aspect of the whole solution.

1 From the first statement in the function, the pointer analysis is started from the beginning to the end. Once a function call statement is encountered which has not been tackled, it will create an active set.
2 If there are some variables existing in both the ref and A, it means this statement is affected by the function call corresponding with the A. These variables need to be pushed into set A. If the referred variables are not in the set A, but the defined variables are in, then the defined variables have to be removed from the active set.
3 It is easy to decide whether the ref set and A set have intersection to reveal the influence on the current branch. This process will perform iteratively until all the function calls and branches are handled.

Example. Table 6 shows an instance of forward slicing towards the code picked from the real tool TAR (list.c)[1], which is used frequently in Unix or Linux system. Here, function *xheader_set_keyword_equal* is analyzed.

4. Dynamic Partial Execution

As seen in the previous section, some function calls do not affect the executing paths in the calling function (usually the function under test). In this case, it is unnecessary to perform symbolic analysis on them. Thus the space of paths in the target program can be reduced, which helps to improve performance of path exploration. But in most cases, results from function calls usually decide path selections of program execution. In order to reduce the scale of exploration space, we introduce a simple technique called *dynamic partial execution* shown in table 4. It makes symbolic analysis tackle with only the part of the path space of called functions instead of the whole one.

Table 3. Forward Slicing Example

n	stmt	ref	def	psa	A	affected
1	p = eq;	eq	p	*p,*eq	ϕ	
2	if(eq[-1] == ':')	eq	ϕ	*p,*eq	ϕ	
	{				ϕ	
3	p--;	p	p	*p,*eq	ϕ	
4	global = false;	ϕ	global	*p,*eq	ϕ	
	}				ϕ	
5	tmp1 = p > kw;	p, kw	tmp1	*p,*eq	ϕ	
6	tmp2 = isspace(*p)	*p	tmp2	*p,*eq	tmp2	
7	tmp3 = tmp1&&tmp2;	tmp1, tmp2	tmp3	*p,*eq	tmp2,tmp3	
8	while(true)	ϕ	ϕ	*p,*eq	tmp2,tmp3	
	{					
9	if(tmp3)	tmp3	ϕ	*p,*eq	tmp2,tmp3	@6
10	p--;	p	p	*p,*eq	ϕ	
11	else break;	ϕ	ϕ	*p,*eq	ϕ	
	}					
12	*p = 0;	ϕ	*p	*p,*eq	ϕ	
13	while(true)	ϕ	ϕ	*p,*eq	ϕ	
	{			*		
14	if(tmp4)	tmp4	tmp4	*p,*eq	tmp4	@16
	{					
15	p = eq + 1;	eq	p	*p,*eq	tmp4	
16	tmp4 = isspace(*p);	*p	tmp4	*p,*eq	tmp4	
17	tmp4 = tmp4 &&(*p);	tmp4, *p	tmp4	*p,*eq	tmp4	
18	p++;	p	p	*p,*eq	tmp4	
	}					
19	else break;	ϕ	ϕ	*p,*eq	ϕ	
20	tmp5 = strcmp(kw, "extender.name");	kw	tmp5	*p,*eq	tmp5	
21	if(tmp5 == 0)	tmp5	ϕ	*p,*eq	tmp5	@20
22	assign_string(&exthder_name, p);	exthder_name, p	ϕ	*p,*eq	*p	no
23	else {...}	ϕ	ϕ	*p,*eq	ϕ	

The algorithm introduces a structure named path_tree to record an path execution tree which can be generated dynamically with its formal definition provided in Table 5. The path_tree is a tree structure. An inner node in the tree corresponds to a decision statement in the target program. It has five fields: fun_id, line_num, predicate, leftchild and rightchild. Fields fun_id, line_num are used to locate the decision statement, field predicate is the condition of the statement. A leaf node has only one field denoting whether the corresponding path has been explored or not.

The algorithm is given in Table 5. It iterates and selects an unexplored leaf each time. Then it traverses from the leaf to the root node entry_fun_id one node by one node. The predicates of these nodes which compose the path traversed are collected to construct path constraints (pc). This process terminates if there are no unexplored paths or pc is unsatisfiable which means that is an infeasible path.

5. Experimental Evaluation

We have implemented the two techniques proposed in this paper into CAUT, a prototype of our automatic testing framework. CAUT improves its performance dramatically by carrying out dependence analysis and dynamic partial execution. To illustrate the effectiveness of our methods, we took the experiments on real C programs, and made some comparisons with CUTE as well. The experiments were

Table 4. Dynamic Partial Execution Algorithm

```
input: path_tree
output: pc(path constraints)
1. leaf = select_not_explored_leaf(path_tree)
     pc = true
     if leaf = null
          algorithm terminates
     else set_explored(leaf)
2. node = leaf → parent
3. if node → fun_id ≠ entry_fun_id
          node = node → parent
          goto 3.
4. if is_not_explored(node → left)
          pc = pc ∧ (node → predicate)
          set_explored(node → left)
     else
          pc = pc ∧ ¬(node → predicate)
          set_explored(node → right)
5. ancestor = node → parent
     repeat
          if ancestor → left = node
               pc = pc ∧ ¬(ancestor → predicate)
          else pc = pc ∧ (ancestor → predicate)
          ancestor = node → parent
     until ancestor = null
6. if is_satisfiable(pc)
          algorithm terminates
     else goto 1.
```

Table 5. Structure of Path Tree

< path_tree >	::=	(fun_id, line_num, predicate, < path_tree >|< path_leaf >, < path_tree >|< path_leaf >)
< path_leaf >	::=	(id, explored | not explored)

carried out on an Intel(R) Core(TM)2 Duo CPU T7300@ 2.00GHz with 2 GB memory, and the results of experiments are shown in Table 6, and Figure 4. These three examples are from different kinds of application libraries.

A bug is found in program *graph* by both. The number of iterations performed to locate the bug in program *graph* is shown in Table 6 on the predefined search depth. Depth is set because loop existing in program leads to infinite cases. From Table 6, if the search depth was set 10, then CUTE

Figure 4. Experimental Results

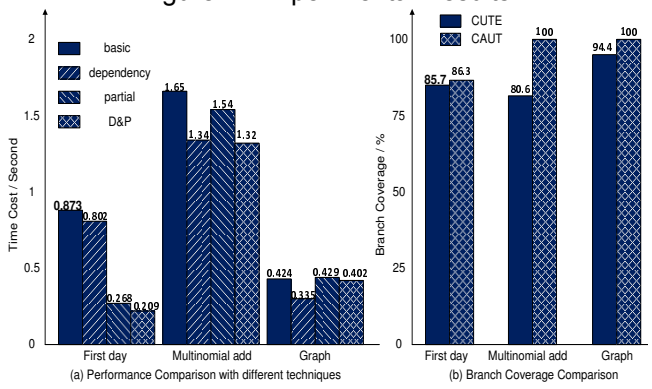


Table 6. Results of Experiment on Graph

	No. of iteration before finding bug	
depth	CUTE	CAUT
10	-	8
20	82	26
30	87	61

cannot find the bug while CAUT can quickly locate the bug in only 8 iterations. In the other two cases where the search depth was set 20 and 30 respectively, our tool adopting the techniques proposed here outperforms CUTE as well. From Figure 4(b), we also see that our tool can achieve better branch coverage rate than CUTE.

The second experiment is to evaluate the different effects on target programs with different techniques. The first case is not using any technique. The second case only uses the dependence analysis. The third case only uses the dynamic partial execution, and the last case uses both the dependence analysis and partial execution techniques denoted with "DP". Figure 4(a) shows the run time results with the depth set to be 20. For the program *first day*, the partial execution technique is very effective, for there exist many branches in the functions called by the main function. But most of them are irrelevant to the contribution for the path selection. Thus, many branches are avoided to be searched by the partial execution technique. The dependence analysis technique does not make evident contributions for the first example because all the called functions affect the path branches. For the other two examples *multinomial add* and *graph*, the method dependence analysis technique makes more remarkable impact than the one dynamic partial execution does. Because some called functions in the two examples will not affect the path selection, and then CAUT ignores symbolic analysis.

6. Related Work

Automated test case generation is intensively studied recently. It is targeting at automated testing and bug detection. Dynamic symbolic execution [18], [19], a variant of symbolic execution, has got much attention and achieved a good evaluation as well. DART [13] blends it with model checking techniques with the goal of systematically executing all feasible paths of a program while detecting various types of errors. CUTE [22] extends the work of DART, and it can handle programs with pointers and data structures by using memory graphs. Cadar *et al.*[8] proposed a similar approach as CUTE, which can automatically generate test cases by using a combination of symbolic and concrete executions. Gulavani *et al.* developed an algorithm for property checking called SYNERGY[14] which combines testing and verification and helps to check whether a program satisfies a set of properties. Recently, Pex [23] has been developed by Microsoft Research which uses dynamic symbolic execution as well as systematic program analysis. Pex is potential to be a practical tool for testing.

There are other approaches to apply test case generation and bug detection techniques to large-scale program. The first category tries to improve path exploration algorithms. Majumdar *et al.* presented *hybrid concolic testing* [20], which interleaves random testing with dynamic symbolic execution to obtain deeper and wider exploration of program state space. The experiments suggest that this method can handle large programs and achieve better branch coverage than both random testing and concolic testing. Jacob *et al.* proposed several heuristics methods for scalable dynamic test generation in [6], towards better branch coverage.

Dealing with compositional units, which will lead to compositional explosion of state space, is another challenge that makes automated testing be hard to apply to real programs. Godefroid [12] developed a technique called *compositional dynamic test generation* for this challenge. This approach extends DART algorithm by modelling and reusing function summaries with dynamic execution, where one function summary consists of a set of pre and post conditions for the summarized function. This approach can reduce the computational complexity but still retain high branch coverage. The disadvantage is that the cost of solving constraints with function summaries is very high. The author improved their technique in [2], in which they adopted a kind of demand-driven symbolic analysis.

7. Discussions and Conclusion

The interesting point with our pointer alias analysis involves the type cast of multi-level pointers. When implementing this pointer alias with type cast, we design a backtrack algorithm to trace the alias link-list, which helps to find the correct type of the current pointer.

The idea of the dynamic partial execution technique introduced in this paper is close to the demand-driven symbolic execution proposed in [2]. Demand-driven symbolic execution emphasizes the summaries collected by test tools on demand instead of building all the test summaries from the bottom-up way [12]. Those two methods are different in the following two aspects:

- The target is different. Demand-driven symbolic execution targets a specific branch or statement to generate as few interprocedural paths as possible. The dynamic partial execution tries to achieve better coverage of the unit under test with exploring only part of spaces in the called functions.
- The execution mechanism is different. Demand-driven symbolic execution tries to store those test summaries on demand and reuses them later. The dynamic partial execution does not collect summaries, and it is just a simple algorithm for exploring path space.

We believe the performance of compositional testing can be improved further by using the test summary technique. We will integrate the technique proposed in [2] into our tool in the future.

We have proposed two techniques to improve the performance of automated testing for the compositional testing. These techniques together can really help the automated testing to scale to practical programs based on the promising results of the experiments.

Acknowledgement

Tao Sun is partially supported by 863 Project No. 2007AA010302 and 973 Project No. 2005CB321904. Geguang Pu is partially supported by NSFC Project No. 60603033 and Qi Mingxin Project No. 07QA14020. Bin Gu is partially supported by NSFC Project No. 90818024.

References

[1] Gnu tar. http://directory.fsf.org/project/tar/.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. *TACAS*, pages 367–381, 2008.

[3] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 3–14. ACM, 2008.

[4] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, A. Kiss, and L. Ouarbya. Formalizing executable dynamic and forward slicing. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 43–52, Washington, DC, USA, 2004. IEEE Computer Society.

[5] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. *23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[7] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. *SPIN*, 2005.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM.

[9] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[10] O. G. Edmund M. Clarke and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

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

[12] P. Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[14] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–127. ACM, 2006.

[15] E. Gunter and D. Peled. Model checking, testing and verification working together. *Form. Asp. Comput.*, 17(2):201–221, 2005.

[16] M. Harman. Automated test data generation using search based software engineering. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[17] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[18] B. Korel. A dynamic approach of test data generation. *Software Maintenance*, pages 311–317, November 1990.

[19] B. Korel. Automated test data generation for programs with procedures. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 209–215, New York, NY, USA, 1996. ACM.

[20] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[21] A. J. Offutt and J. H. Hayes. A semantic model of program faults. *SIGSOFT Softw. Eng. Notes*, 21(3):195–200, 1996.

[22] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13*, pages 263–272, New York, USA, 2005. ACM.

[23] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. *The Second International Conference on Tests and Proofs*, pages 171–181, 2008.

[24] Z. Wang, G. Pu, X. Yu, and J. He. Scalable path search for automated test case generation. *Technical Report, http://caut.lab205.org/publication/publication.htm*, 2009.

[25] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.