Precise Guidance to Dynamic Test Generation

TheAnh Do, A. C. M. Fong and Russel Pears
School of Computing and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand

Keywords: Dynamic Symbolic Execution, Automated Test Input Generation, Software Testing, Data Flow Analysis.

Abstract: Dynamic symbolic execution has been shown an effective technique for automated test input generation. However, its scalability is limited due to the combinatorial explosion of the path space. We propose to take advantage of data flow analysis to better perform dynamic symbolic execution in the context of generating test inputs for maximum structural coverage. In particular, we utilize the chaining mechanism to (1) extract precise guidance to direct dynamic symbolic execution towards exploring uncovered code elements and (2) meanwhile significantly optimize the path exploration process. Preliminary experiments conducted to evaluate the performance of the proposed approach have shown very encouraging results.

1 INTRODUCTION

Testing is a widely adopted technique to ensure software quality in software industry. For about 50% of the total software project costs are devoted to testing. However, it is labour-intensive and error-prone. An attempt to alleviate those difficulties of manual testing is to develop techniques to automate the process of generating test inputs. For over the last three decades, techniques have been proposed to achieve this goal, ranging from random testing (Bird and Munoz, 1983), symbolic execution (King, 1976), search-based testing (McMinn, 2004), the chaining approach (Ferguson and Korel, 1996), to dynamic symbolic execution (Godefroid, 2005; Sen, 2005).

Among these techniques, dynamic symbolic execution has been gaining a considerable attention in the current industrial practice (Cadar et al., 2011). It intertwines the strengths of random testing and symbolic execution to obtain the scalability and high precision of dynamic analysis, and the power of the underlying constraint solver. One of the most important insights of dynamic symbolic execution is the ability to reduce the execution into a mix of concrete and symbolic execution when facing complicated pieces of code, which are the critical obstacle to pure symbolic execution. While effective, the fundamental scalability issue of dynamic symbolic execution is how to handle the combinatorial explosion of the path space, which is extremely large or infinite in sizable and complex programs. Dynamic symbolic execution therefore, if performed in a way to exhaustively explore all feasible program paths, often ends up with small regions of the code explored in practical time, leaving unknown understanding about the unexplored.

In fact, covering all feasible paths of the program is impractical. Besides, testing large programs and referring to sophisticated criteria can often be out of the limit of a typical testing budget. In the practice of software development, therefore, high code coverage has been long advocated as a convenient way to assess test adequacy (British Standards Institute, 1998; RTCA, Inc. 1993). Specifically, the testing process must ensure every single code element in the program is executed for at least once. In this context, dynamic symbolic execution can be conducted so as to cover all code elements rather than exploring all feasible program paths. This may lead to a significant reduction in the number of paths needed to explore. However, the question of "*how can we derive precise guidance to perform dynamic symbolic execution towards achieving high code coverage?*" becomes important. This question emphasizes two aspects: high coverage achievements and minimal path explorations. The second aspect is essential in the sense that the cost of performing dynamic symbolic execution is expensive, especially in large programs, any technique helping achieve high code coverage must optimize path explorations to be applicable within resources available e.g., CPU, memory and time.

Do T., Fong A. and Pears R..
Precise Guidance to Dynamic Test Generation.
DOI: 10.5220/0003969000050012
In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2012), pages 5-12
ISBN: 978-989-8565-13-6

To answer this question, we propose to apply data flow analysis to better perform dynamic symbolic execution in the test input generation process. Particularly, we utilize the chaining approach (Ferguson and Korel, 1996) to pull out precise guidance in order to direct dynamic symbolic execution towards *effectively* and *efficiently* exploring code elements. Specifically, given a test goal (an unexplored code element e.g., statement or branch), the chaining approach first performs data dependency analysis to identify statements that affect the execution of the test goal, and then it uses these statements to form sequences of events that is to be executed prior to the execution of the test goal. The advantage of doing this is twofold: (1) it precisely focuses on the cause of getting the test goal to be executed and (2) it slices away code segments that are irrelevant to the execution of the test goal. As we will show in the evaluation, these two strengths enable dynamic symbolic execution to achieve higher code coverage and at the same time significantly optimize the number of path explorations required to unclose high-complexity code.

The paper is organised as follows. Section 2 introduces dynamic symbolic execution and highlights the path space explosion problem. Section 3 provides a brief survey of related work. Section 4 illustrates the chaining approach. Section 5 explains the prototype implementation and discusses the experimental results. We discuss research issues and future work in Section 6, and conclude the paper in Section 7.

2 DYNAMIC SYMBOLIC EXECUTION

The key idea behind dynamic symbolic execution (Godefroid et al., 2005) is to start executing the program under test with *concrete values* while gathering symbolic predicates of corresponding *symbolic values* along the execution. By negating one symbolic predicate and solving the path constraint with an off-the-shelf constraint solver, it can obtain a new input to steer the execution along an alternative program path. This process is often performed in an attempt to exhaustively systematically explore all feasible paths of the program. Dynamic symbolic execution hence outperforms "classical" symbolic execution through being able to simplify complex constraints, and deal with complex data structures and native calls.

```
Node   typedef enum {false, true} bool;
       #define N 20
(s)    bool CheckArray(int A[N]) {
         int i;
(1)      bool success = true;
(2)      for (i = 0; i < N; i ++) {
(3)        if (A[i] != 25)
(4)          success = false;
         }
(5)      if (success) {
(6)        // target
         }
(e)    }
```

Figure 1: The *CheckArray* function checks if all elements of an input array equal 25.

To perform dynamic symbolic execution, code of the program is instrumented in a way that concrete execution can be executed simultaneously with symbolic execution. So, while the former drives the execution, the latter maintains a symbolic memory *S*, which maps memory addresses to symbolic expressions, and a symbolic path constraint *PC*, which is a first-order quantifier-free formula over symbolic expressions. In this way, once an expression is evaluated, it is evaluated both concretely and symbolically, and both physical memory and symbolic memory are updated accordingly. Similarly, once a conditional statement *if (e) then S1 else S2* is executed, *PC* is updated according to the "then" or "else" branch taken. If the "then" branch is taken, *PC* becomes $PC \wedge \sigma(e)$; otherwise, it is $PC \wedge \neg\sigma(e)$, where $\sigma(e)$ denotes the symbolic predicate obtained by evaluating e in symbolic memory. As a result, the symbolic path constraint *PC* presenting a symbolic execution of the program is as follows:

$$PC = \sigma_1 \wedge \ldots \wedge \sigma_{i-1} \wedge \sigma_i \wedge \sigma_{i+1} \wedge \ldots \wedge \sigma_n \qquad (1)$$

Every single symbolic predicate of *PC* represents one possibility to execute the program along an alternative path. That is one can randomly pick up a predicate, e.g., σ_i, negate it, and then form the constraint system $(\sigma_1 \wedge \ldots \wedge \sigma_{i-1} \wedge \neg\sigma_i)$ to be solved by the underlying constraint solver. The satisfiability of the constraint system results in an input that the execution of the program with this input will follow the previous path up to the corresponding conditional statement of the negated predicate, but afterwards change the flow of control to the other branch. Consequently, for every symbolic path constraint, the number of program paths can be 2^n or be exponential in the number of symbolic predicates. In practice, the number of symbolic

predicates of the program is often extremely large (or even infinite), especially in the presence of loops and/or recursions, causing dynamic symbolic execution to face with the combinatorial explosion of the path space. The *CheckArray* function in Figure 1 could be a good example to illustrate this phenomenon. It takes input an array of 20 elements and check if all elements equal 25. This yields 2^{20} (=1,048,576) paths with just 20 symbolic predicates. In practice, this problem becomes worse as the input of programs can be a stream of data with too large (or unknown) size. In case of *check_ISBN* and *check_ISSN* in the set of test subjects, for instance, both functions take an array with (4093+3) tokens (causing approximately $2^{(4093+3)}$ paths), making dynamic symbolic execution ill-suited for the goal of covering all code elements of programs. It is therefore necessary to devise appropriate search strategies to guide dynamic symbolic execution to achieve high code coverage within a minimal testing budget. In the next section, we provide a brief survey of related test input generation techniques based on dynamic symbolic execution to assess the current state of research.

3 RELATED WORK

Cadar et al (2011) give a preliminary assessment of the use of (modern) symbolic execution in academia, research labs and industry in which the authors emphasize "*A significant scalability challenge for symbolic execution is how to handle the exponential number of paths in the code*". In the context of using dynamic symbolic execution to generate test inputs for maximum code coverage, tools being in favour of depth-first explorations such as DART (Godefroid et al., 2005) and CUTE (Sen et al., 2005) deeply widen the program path space but lack the ability to forward the execution to further unexplored control flow points. These approaches when executed against large programs for finite time achieve very low code coverage. Pex (Tillmann and Halleux, 2008) is an automated structural testing tool developed at Microsoft Research. It combines a rich set of basic search strategies and gives a fair choice among them. While the combination helps maximize code coverage through attempting different program control flows, discovering code elements may require specific guidance of control and data flow analysis. Fitnex (Xie et al., 2009) further makes Pex more guided by using fitness functions to measure the improvement of the path exploration process. The main obstacle of this approach is the flag problem (Binkley et al., 2011), where fitness functions face a flat fitness landscape, giving no guidance to the search process. Flags, however, are widely used in real world software (Binkley et al., 2011). CREST (Burnim and Sen, 2008) is an extensible platform for building and experimenting with heuristics for achieving high code coverage. Among search heuristics implemented in CREST, CfgDirectedSearch is shown more effective than the others through the reported experimental data. This search strategy leverages the static control flow of the program under test to guide the search down short static paths to unexplored code. Theoretically, the control flow guidance may be imprecise since the execution of code elements may require data dependencies going beyond short paths and/or being calculated in dynamic paths.

Obviously, with sizable and complex programs, the difficulty of using dynamic symbolic execution to generate test inputs for maximum code coverage is among the far too many program paths, how to mine for appropriate paths to guide the search process towards exposing unexplored code elements. In the next section, we introduce the chaining mechanism in an attempt to address this issue.

4 THE CHAINING APPROACH

The chaining approach (Ferguson and Korel, 1996) was proposed to make use of data dependency analysis to guide the search process. The basic idea is to identify statements leading up to the goal structure, which may influence the outcome of the test goal. Those statements are sequences of events that the search process must walk along to target the test goal. The chaining approach can hence be considered as a slicing technique (Tip, 1995) which simplifies programs by focusing on selected aspects of semantics. As a result, the chaining approach can provide precise guidance since it forces the consideration of data flows, and it is effective since it slices away irrelevant code segments to the execution of the test goal. These two strengths can guide the search process into potentially unexplored but promising areas of the path space to unclose high-complexity code.

We illustrate the core of the chaining mechanism using again the function *CheckArray* in Figure 1 in which the test goal is to cover branch (5, 6). For this, the chaining mechanism first generates the following initial event sequence $E_0 = <(s, \emptyset), (6, \emptyset)>$ where each event is a tuple $e_i = (n_i, S_i)$ where n_i is a program node and S_i is a set of variables referred to

as a constraint set. Now suppose that the search process fails to find an input array with all elements equal 25 to execute the target branch, moving from node 5 to node 6. Node 5 is hence considered to be a *problem node*. Formally, a problem node refers to a conditional statement for which the search process within a fixed testing budget cannot find inputs to execute an intended branch from this node. The chaining mechanism then performs data flow analysis in respect of this problem node to identify statements that define data for variables used in the conditional expression. In this case, the conditional expression consists of variable *success*, which is defined at nodes 1 and 4. Two event sequences are constructed accordingly, E_1 and E_2, based on the initial event sequence.

$E_1 = <(s, \emptyset), (1, \{success\}), (5, \emptyset), (6, \emptyset)>$
$E_2 = <(s, \emptyset), (4, \{success\}), (5, \emptyset), (6, \emptyset)>$

Notice that for every two adjacent events in an event sequence, $e_i = (n_i, S_i)$ and $e_{i+1} = (n_{i+1}, S_{i+1})$ there must exist a path from n_i to n_{i+1} along which all variables in S_i are not redefined. Such a path allows the effect of a definition statement to be transferred up to the target structure. Obviously, the sequence E_2 cannot help to explore the test goal as the value of *success* variable is *false*, which leads to the execution of the "else" branch instead. The event sequence E_1, on the other hand, guides the search process to first reach node 1 from the function entry, which sets the value of *success* variable to the desired *true* value to explore branch (5, 6), and then continues from node 1 to node 5. When moving to node 5, the value of *success* variable may be killed at node 4 if branch (3, 4) is executed. If so, the search process is guided to change the flow of control at node 3 to the "else" branch, which prevents *success* variable from being set to the unwanted *false* value. This guidance is continuously refined throughout the *for* loop to preserve the constraint set {*success*} of event (1, {*success*}) while reaching to event (5, Ø). By doing so, the value of all elements in the input array is altered to 25, providing the desired input to expose the test goal, branch (5, 6).

We now formalize the process of creating a new event sequence from an existing sequence E. Let $E = <e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_m>$ be an event sequence. Suppose the search process driven by the event sequence guides the execution up to event e_i and a problem node p is encountered between events e_i and e_{i+1}. Let d be a definition statement of problem node p. Two events are generated, $e_p = (p, \emptyset)$ and $e_d = (d, D(d))$, corresponding to the problem node and its definition. A new event sequence is now created by inserting these two events into sequence E. Event e_p is always inserted between e_i and e_{i+1}. However, event e_d, in general, may be inserted in any position between e_1 and e_p. Suppose the insertion of event e_d is between events e_k and e_{k+1}. The following event sequence is then created:

$E' = <e_1, \ldots, e_{k-1}, e_k, e_d, e_{k+1}, \ldots, e_{i-1}, e_i, e_p, e_{i+1}, \ldots, e_m>$

Since new events are added to the sequence, the implication of data propagation may be violated. This requires modifications of the associated constraint sets of involved events. The update is done in three steps:

(1) $S_d = S_k \cup D(d)$
(2) $S_p = S_i$
(3) $\forall j, k + 1 \leq j \leq i, S_j = S_j \cup D(d)$

In the first step, the constraint set S_d of event e_d is initialized to the union of D(d) and the constraint set of the preceding event e_k. This modification ensures that the constraint set S_k of event e_k is preserved up to event e_{k+1} while getting through the new inserted event e_d. The second step also imposes the same requirement on event e_p by assigning S_i to its constraint set. In the final step, all constraint sets of events between e_{k+1} and e_i are modified by including a variable defined at d. By doing this, the chaining mechanism guarantees to propagate the effect of the definition at node d up to the problem node p.

Created event sequences may be organised in a form of a tree referred to as a *search tree*. The initial event sequence E_0 represents the root of the tree. Other levels of the tree are formed by event sequences created when problem nodes are encountered. One tree node represents a possibility to unclose the test goal. The search process when following an event sequence attempts to adjust the execution to move from one event to another without making the constraint set in the previous event violated. This suggests a systematic mechanism to propagate the effect of all possible data flows up to the target goal structure. When intertwined with dynamic symbolic execution, this search process specifically involves picking up relevant symbolic predicates, negating them, and then forming constraint systems to be solved by the underlying constraint solver for inputs. Dynamic symbolic execution directed by the precise guidance of the chaining mechanism can hence significantly strengthen the test input generation process in

Table 1: Percentage of branch coverage achieved by search strategies when executed on 9 test subjects.

Subject	loc	br	Random	DFS	CREST	Pex	Fitnex	STIG
sample	31	12	42	92	92	92	92	100
testloop	17	8	13	88	88	100	100	100
hello_world	37	32	34	56	91	91	91	100
netflow	28	6	83	83	83	100	100	100
moveBiggestInFront	37	6	100	83	83	100	100	100
handle_new_jobs	37	6	67	100	100	83	100	100
update_shps	52	10	60	90	90	100	100	100
check_ISBN	78	52	83	83	83	96	83	98
check_ISSN	78	52	83	83	94	96	83	98
Average	44	20	63	84	89	95	94	**100**$^{\approx}$

Table 2: Measurements of number of program explorations required by search strategies.

Subject	Random	DFS	CREST	Pex	Fitnex	STIG
sample	1000	1000	1000	1000	1000	13
testloop	1000	1000	1000	26	27	22
hello_world	1000	1000	1000	1000	1000	55
netflow	1000	2	1000	5	5	2
moveBiggestInFront	15	1000	1000	4	4	2
handle_new_jobs	1000	2	2	1000	99	34
update_shps	1000	1000	1000	5	7	4
check_ISBN	1000	1000	1000	234	313	51
check_ISSN	1000	1000	1000	234	313	45
Average	891	778	889	390	308	**25**

Notice: *loc* number of lines of code
br number of branches
Random random input search
DFS depth-first search

coverage achievements and path exploration optimizations.

5 PRELIMINARY EVALUATION

We have implemented our proposed approach as a search heuristic, called STIG (Scalable Test Input Generation), on the CREST platform (Burnim and Sen, 2008), an automatic test input generation tool for C, based on dynamic symbolic execution. Since CREST unrolls decisions with multiple conditions as an equivalent cascade of single condition decisions and converts every single conditional statement into a form of *if (e) then S1 else S2*, branch coverage achieved by CREST is comparable to condition coverage in the original program. For any specification clothed in forms of assertion calls, i.e., *assert (e)*, we transform into a conditional statement *if (!e) error();* to check for errors. In this case, the test goal is to explore the function call *error*.

To evaluate the effectiveness of our approach, we chose a set of test subjects and conducted experiments to compare STIG with two widely adopted search strategies, random input search and depth-first search, and with three test input generation tools CREST, Pex and Fitnex. For CREST, we chose the control-flow graph directed search strategy (CfgDirectedSearch) which was confirmed the "best" search algorithm through the reported experimental data. The test input generation CREST tool used in the whole paper thus refers to this chosen search strategy. For Pex, it implements dynamic symbolic execution to generate test inputs for .NET code, supporting languages C#, VisualBasic, and F#. Besides, Fitnex is an extension of Pex. The test subjects selected include the *sample* function which was borrowed from the work of Ferguson and Korel (1996). The next two test subjects, *testloop* and *hello_world*, were from the work of Xie et al. (2009). These test subjects were employed in literature to illustrate the essence of individual exploration problems. Thus we want to check if STIG by using data flow analysis is able to tackle those exploration problems. Notice that the *hello_world* function in this evaluation was modified to check an input array which must start with "Hello", end with "World!" and contain only spaces. This modification makes the function more difficult for search strategies to cover all its code coverage.

The rest of the test subjects were mentioned in the work of Binkley et al. (2011). These functions come from open-source programs; we hence want to evaluate the capability of STIG in dealing with the high complexity of real world programs as compared to the others'. For the sake of experiments, for some functions we just extracted part of their code. All the test subjects are in C code, to make comparison with Pex and Fitnex we converted them to C# code.

All experiments in the evaluation were run on 3GHz Core™2 Duo CPU with 4GB of RAM and running Ubuntu GNU/Linux for Random, DFS, CREST and STIG, and Windows for Pex and Fitnex.

The first purpose of the experiment is to evaluate the capability of each tool (or search strategy) in achieving high code coverage, hence it is fair to set up a fixed testing budget for all. For this, we chose 1000 runs as the limit to run every test subject on each tool. We measured the percentage of branch coverage obtained. The results are shown in Table 1. The second purpose is to evaluate the capability of each tool in optimizing the path exploration process. As mentioned, this is an important criterion to assess the effectiveness of any test input generation tool based on dynamic symbolic execution since the cost of performing dynamic symbolic execution is expensive, minimizing the number of path explorations is necessary to make the technique applicable in practice. For this, besides the first stop condition, 1000 runs, we also stopped tools when all branch coverage of the experimenting test subject is met. The results are given in Table 2.

Table 1 and Table 2 summarize the statistics we obtained after we carried out the experiments. It is clear from the statistics that Random is the worst approach to test input generation with the lowest average coverage (63%) obtained but the highest average number of runs (891 runs) exploited. DFS is an instance of using dynamic symbolic execution to systematically explore all feasible paths of the program. It lacks the ability to forward the execution to further unexplored control flow points and hence achieved very low branch coverage within the fixed testing limit. However, since DFS relies on the power of the underlying constraint solver, it must obtain higher coverage (84% in average) than Random. For CREST, it had 8 out of 9 cases failed to achieve full coverage. For these cases, the test subjects contain branches that require precise guidance of data flow analysis to be covered. Only CREST utilizes the static control flow and thus is not effective. Pex and Fitnex achieved quite similar average coverage results, 95% and 94%, respectively. While Pex failed in 5 cases to achieve full coverage, Fitnex had 1 fewer case. In cases of *check_ISBN* and *check_ISSN*, both Pex and Fitnex automatically terminated after 234 and 313 runs, respectively, although all coverage was not achieved. The comparison thus favours these tools in the aspect of path explorations. The results obtained by both Pex and Fitnex are better than Random, DFS and CREST in terms of coverage achievements and exploration optimizations. This highlights the power of bringing several search strategies together as well as the power of fitness functions in test input generation. For STIG, it failed to achieve 100% coverage in 2 cases, *check_ISBN* and *check_ISSN*. We manually investigated these test subjects and realized that the two functions contain one unreachable branch which is resulted from the instrumentation step where our tool normalizes every *if* statement to have the form *if (e) then S1 else S2*. Currently, STIG is not able to deal with infeasible code. But an interesting observation when we conducted experiments on these test subjects is that even though we set the testing limit to 1000 runs, STIG stopped the exploration process after 51 runs for *check_ISBN* and 45 runs for *check_ISSN*. This means the search process considered all possible combinations of data flows but none could help to explore the test goal. This suggests evidence that this code element is infeasible. We refer this situation to *saturated data propagation* and are working to give a formal proof for identifying infeasible code through exploring data flows. It is *worth* mentioning that on average STIG achieved the highest coverage (100% if infeasible code is not counted) and maintained a significantly small number of program explorations (25 runs compared to 308 and 390 of Fitnex and Pex, respectively, and 889 of CREST) on the selected test subjects. This shows the capability of utilizing data flow analysis to guide dynamic symbolic execution in the test input generation process.

6 DISCUSSION

The Chaining Approach. The chaining approach we utilized in this work is a test input generation technique (Ferguson and Korel, 1996), which relies on a local search method called the *alternating-variable method* to find test inputs but this is performed largely randomly. In addition, the chaining mechanism itself mainly focuses on propagating the effect of definition statements to the target structure but lacks the ability to consider at a definition it may need to perform certain

computations to satisfy the target predicate. This limitation was *partially* addressed in the work of McMinn and Holcombe (2006) and has been strengthened by STIG to be able to intertwine with dynamic symbolic execution.

Complexity. The cost of applying the chaining mechanism comes from two facets. One is performing data flow analysis to identify definition statements (or formally *reaching definitions*) (Aho et al., 2008) of problem nodes. This is a maximum fixedpoint algorithm operated statically on the source code of the program prior to dynamic symbolic execution. The algorithm complexity is the product of the height of the lattice and the number of nodes in the program flow graph, which is minor compared to the very expensive cost of performing dynamic symbolic execution. The other is the cost of performing dynamic symbolic execution with the guidance of event sequences. This cost results actually in the number of runs that STIG requires to execute the program, which was confirmed significantly smaller than other search strategies and tools. In fact, we observed from the experiments that CREST and STIG both executed the test subjects within a matter of a few seconds. Pex and Fitnex, however, consumed a considerable amount of time on all the test subjects.

Evaluation. The evaluation was conducted in a small set of test subjects. However, these test subjects reveal characteristic exploration problems of real world programs for which dynamic symbolic execution without guidance is ineffective to apply. Future work aims to extend the proposed approach and conduct experiments on large test subjects to properly assess the validity of our proposal and observations. We believe that when testing sizeable and complex programs, where the path space is too large to systematically exhaustively explore, the ability to break down the path space and to precisely guide the search process by centralizing on selected aspects of semantics of our proposed approach is essential in optimizing the very expensive cost of performing dynamic symbolic execution to maximize coverage achievements and enhance error-detection capabilities.

7 CONCLUSIONS

Achieving high code coverage is an important goal of software testing. Dynamic symbolic execution based techniques hold most promise to make this goal achievable. When applied to real world software, the scalability of dynamic symbolic execution, however, is limited due to the extremely large program path space. In this paper, we have proposed to apply data flow analysis to effectively and efficiently perform dynamic symbolic execution for maximum code coverage. The proposed approach alleviates the combinatorial path space explosion by guiding the search process to focus on code segments that truly affect the execution of uncovered code. The experimental evaluation shows that STIG is effective in maximizing code coverage, optimizing path explorations, and providing useful evidence to identify infeasible code elements. In most of the experiments, STIG achieves higher coverage with significantly small path explorations than popular state-of-the-art test case generation tools.

ACKNOWLEDGEMENTS

We thank Kiran Lakhotia for sending us source code of test subjects used in his work (Binkley et al., 2011). We are grateful to Nikolai Tillmann and Tao Xie for their help on Pex and Fitnex.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2008). *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition.

Binkley, D. W., Harman, M., and Lakhotia, K. (2011). FlagRemover: A testability transformation for transforming loop-assigned flags. *ACM Transactions on Software Engineering and Methodology 20(3)*.

Bird, D., and Munoz, C. (1983). Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3), 229-245.

British Standards Institute (1998). BS 7925-1 Vocabulary of Terms in Software Testing.

Burnim, J., and Sen, K. (2008). Heuristics for scalable dynamic test generation. In *ASE*, pp. 443-446.

Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. (2011). Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, pp. 1066-1071.

Ferguson, R., and Korel, B. (1996). The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1).

Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: directed automated random testing. In *PLDI '05*, pp. 213-223.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19, 385 - 394.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2).

McMinn, P., and Holcombe, M. (2006). Evolutionary Testing Using an Extended Chaining Approach. *Evolutionary Computation*, 14(1).

RTCA, Inc. (1993). Document RTCA/DO-178B. U.S. Department of Transportation, *Federal Aviation Administration*, Washington, D.C.

Sen, K., Marinov, D., and Agha, G. (2005). CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13*, pp. 263-272.

Tillmann, N., and Halleux, J. D. (2008). Pex–white box test generation for .NET. In *Beckert, B., Hahnle, R. (eds.) TAP 2008*. LNCS, vol. 4966, pp. 134-153. Springer, Heidelberg.

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 121-189.

Xie, T., Tillmann, N., Halleux, P. d., and Schulte, W. (2009). Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, pp. 359-368.