Characteristic Studies of Loop Problems for Structural Test Generation via Symbolic Execution

Xusheng Xiao
North Carolina State University
Raleigh, NC, USA
xxiao2@ncsu.edu

Sihan Li, Tao Xie
University of Illinois at Urbana-Champaign
Champaign, IL, USA
{sihanli2, taoxie}@illinois.edu

Nikolai Tillmann
Microsoft Research
Redmond, WA, USA
nikolait@microsoft.com

***Abstract*—Dynamic Symbolic Execution (DSE) is a state-of-the-art test-generation approach that systematically explores program paths to generate high-covering tests. In DSE, the presence of loops (especially unbound loops) can cause an enormous or even infinite number of paths to be explored. There exist techniques (such as bounded iteration, heuristics, and summarization) that assist DSE in addressing loop problems. However, there exists no literature-survey or empirical work that shows the pervasiveness of loop problems or identifies challenges faced by these techniques on real-world open-source applications. To fill this gap, we provide characteristic studies to guide future research on addressing loop problems for DSE. Our proposed study methodology starts with conducting a literature-survey study to investigate how technical problems such as loop problems compromise automated software-engineering tasks such as test generation, and which existing techniques are proposed to deal with such technical problems. Then the study methodology continues with conducting an empirical study of applying the existing techniques on real-world software applications sampled based on the literature-survey results and major open-source project hostings. This empirical study investigates the pervasiveness of the technical problems and how well existing techniques can address such problems among real-world software applications. Based on such study methodology, our two-phase characteristic studies identify that bounded iteration and heuristics are effective in addressing loop problems when used properly. Our studies further identify challenges faced by these techniques and provide guidelines for effectively addressing these challenges.**

I. Introduction

Software testing is one of the most commonly used techniques for improving software quality, but it is typically a labor-intensive process that accounts for high cost in software development and maintenance. To address the issue, Dynamic Symbolic Execution (DSE) [18], [34], [41], recently made popular with the advent of powerful constraint-solving tools [7], [14], [19], can be used to automatically generate high-covering tests by systematically exploring program paths.

DSE executes the program under test symbolically with arbitrary or default inputs and collects path conditions as the constraints on inputs from the executed branch statements. DSE then systematically negates part of these constraints to form new path conditions, and leverages a constraint solver to solve these path conditions for obtaining new test inputs. These new test inputs steer the future explorations towards different paths of the program, iteratively collecting new constraints and achieving new structural coverage, such as statement and branch coverages [5]. With the advances of research on constraint solvers, e.g., Z3 [14] and CVC3 [7], DSE-based tools, such as SAGE [19] and Pex [48], become promising in generating test inputs for unit testing and (security) system testing [10], [11], [19], [41], [48]. Since 2008, SAGE has been continually running on more than 100 machines in a security testing lab, and Pex found serious faults from an already-well-tested component of the .NET runtime.

A significant challenge for DSE is how to handle loops, which may compromise DSE's effectiveness in terms of structural coverage and fault detection. As a special type of branches, loops can cause the number of paths to be explored to grow exponentially. Even worse, the number of paths becomes infinite due to the presence of input-dependent loops (IDLs)[1], causing DSE to run out of resources (e.g., the allocated time or number of explored paths) before achieving satisfactory coverage. For example, a recent study [37] shows that DSE may keep unfolding an IDL without achieving coverage of any new branches. Recent research has tried to address this challenge by using bounded iteration to constrain loop unrolling [19], using search-guiding heuristics to guide path exploration [11], [48], [54], or using loop summarization to summarize loops based on inferred loop invariants [20], [40], [42] (referred to as *loop summarization*).

Although loops whose iteration counts are statically bounded may still pose problems to DSE, IDLs are considered as much more severe problems since IDLs may cause the search space to become infinite. Some previous work [20], [28], [29] proposes effective techniques to assist DSE in dealing with IDLs (referred to as *loop techniques*), but these techniques have their limitations in handling different kinds of IDLs. For example, bounded iteration and search-guiding heuristics effectively prune the search space, but they may prevent DSE from covering subsequent branches that require loops to iterate more than a certain number of times [20]. Inferring loop invariants as conducted by loop summarization requires some abstractions to be performed on the symbolic states, and may lose precision on summarizing the loops [28], [29]. Moreover, loop summarization requires detection of induction variables whose values are modified by a constant

[1]Input-dependent loops are loops whose iteration numbers depend on some unbounded input.

value or constant times for each loop iteration. However, only a few IDLs can be summarized by induction variables. Godefroid et al. [20] report that their technique can summarize only 33% (6 out of 19) of the detected IDLs in their evaluations.

Unfortunately, there exists no literature-survey or empirical work that shows the pervasiveness of loop problems, or identifies challenges faced when these techniques are applied on real-world applications. To fill such a significant gap, we provide two-phase characteristic studies on loop problems for structural test generation, with the focus on test-generation tools based on DSE. Our proposed study methodology starts with conducting a literature-survey study to investigate how technical problems such as loop problems compromise automated software-engineering tasks such as test generation, and which existing techniques are proposed to deal with such technical problems. Then the study methodology continues with conducting an empirical study of applying the existing techniques on real-world applications sampled based on the literature-survey results and major open-source project hostings. This empirical study investigates the pervasiveness of the technical problems and how well existing techniques can address such problems among real-world applications.

In our literature-survey study, we survey the literature from a comprehensive bibliography of papers on symbolic execution and tasks assisted by symbolic execution [1]. DSE is a symbolic-execution-based technique, and thus faces the same path-explosion problem as symbolic execution. The analysis results of loop problems and loop techniques from the surveyed articles of symbolic execution are applicable and beneficial to loop problems for DSE. In particular, we seek to answer the following questions:

RQ1.1: Are loop problems a kind of major problems to compromise the effectiveness of symbolic execution in various tasks? Symbolic execution has been applied for various tasks, such as test generation [18], [25], [38], [41], [43], [46], [47], [51], [53], debugging [31], security analysis [9], [12], [49], verification [21], [27]. The answer to this question can help understand which of these tasks would need to specifically address loop problems.

RQ1.2: What kinds of techniques are proposed to deal with loops, and how widely are the techniques used across various tasks assisted by symbolic execution? In addition, we study the advantages and disadvantages of different kinds of techniques, providing guidelines on choosing techniques for various tasks.

In our empirical study, we choose certain representative techniques (found from our literature-survey study) to apply on real-world applications. We sample subject applications from certain categories (data structures and parsers) that are found to typically have loop problems from the surveyed articles, and select subject applications randomly from major open-source project hostings including CodePlex [2] and GitHub [3]. In particular, we seek to answer the following questions:

RQ2.1: How pervasive are IDLs in open-source applications? What are the distribution of nested loops? We classify loops as Fixed-Iteration Loops (FILs) or Input-Dependent Loops (IDLs), and identify nested loops that pose challenges for various techniques, such as loop summarization. The classification results of FILs and IDLs can show the pervasiveness of IDLs, and help understand the causes of IDLs. Since nested loops pose extra challenges, the distributions of nested loops provide insights on what kinds of applications may need manual assistance in dealing with nested loops.

RQ2.2: How well can existing techniques, such as bounded iteration and search-guiding heuristics, address the loop problems posed by IDLs on real-world applications? As our literature-survey study identifies bounded iteration and search-guiding heuristics as the most widely used techniques across various tasks, we apply Pex, the state-of-the-art DSE-based tool, with the mixed search strategy [54][2] and Pex with bounded iteration (referred to as Pex-bounded) on the sampled software applications to explore IDLs. The results can show how well these techniques handle loop problems on real-world applications. Moreover, our studies identify challenges faced by existing techniques and propose guidelines on how to address these challenges.

Our paper makes the following main contributions:

- We present the first characteristic studies to guide research on addressing loop problems for structural test generation.
- We propose a two-phase methodology on studying technical problems in automated software-engineering tasks: (1) a literature-survey study that investigates how technical problems may compromise automated software-engineering tasks such as test generation, and which existing techniques are proposed to deal with such technical problems; (2) an empirical study that investigates the pervasiveness of the technical problems and how well existing techniques can address such problems among real-world software applications.
- We find out that bounded iteration and search-guiding heuristics are two most widely used techniques across various tasks assisted by symbolic execution, and both techniques are effective in addressing loop problems on real-world applications when used properly.
- We identify various challenges that compromise the effectiveness of existing loop techniques and provide valuable guidelines for addressing these challenges.

II. Literature-Survey Study

In this section, we present the details of the literature-survey study and its implications. Our literature-survey study investigates how loop problems compromise various tasks assisted by symbolic execution, such as testing and security analysis, and what techniques are proposed to deal with loop problems.

A. Study Setup

In our literature-survey study, we survey the literature from a comprehensive bibliography of articles on symbolic execution

[2] A search strategy that combines different search strategies in a fair round-robin scheme.

TABLE I
RESEARCH AREAS OF SURVEYED ARTICLES

Area	# Articles
Software Engineering	74 (46.5%)
Security	22 (13.8%)
Systems	15 (9.4%)
Programming Languages	17 (10.7%)
Verification	29 (18.2%)
Database	2 (1.3%)
Total	159

and its applications [1]. The articles are published in a wide range of venues, including conferences in software engineering (e.g., ICSE, FSE, ISSTA), systems (e.g., ASPLOS, OSDI), security (e.g., CCS, Oakland, USENIX Security), software verification (e.g., CAV, TACAS, VMCAI), programming languages (e.g., POPL, PLDI, SAS, CC), and database (e.g., SIGMOD). Since the bibliography includes only a few articles published in 2012, we manually collect more articles in 2012 for the study. The details of the study and results can be found on our project website [4].

Study Design. We first classify the articles into different tasks assisted by symbolic execution. We then investigate the percentage of the articles that propose techniques to deal with loops (referred to as *loop articles*) in each task. Such data can help draw conclusions on how loops may compromise tasks assisted by symbolic execution.

We then categorize the techniques proposed by the studied articles to address loop problems. This categorization can indicate not only which technique is frequently used in which tasks, but can also identify which technique is widely used across different tasks.

Statistics of Literature-Survey Study. In our literature-survey study, we survey 159 articles in total. Table I shows the number of articles that we survey in each research area. We next describe the detailed results for answering the questions.

B. RQ1.1: Loop Problems in Various Tasks

Table II shows the task categories of our literature-survey study. Column "Total" shows the number of articles that belong to various tasks assisted by symbolic execution, and Column "Loop" shows the number of loop articles. We categorize the articles into five major types of tasks: (1) software testing: running a program with generated test inputs and dynamically checking its behavior (Row "Testing"); (2) formal verification: checking the correctness of a program by verifying that the program meets requirements or given properties, often in the form of static analysis (Row "Verification"); (3) security analysis: identifying security vulnerabilities of a program (Row "Security") (4) debugging: locating and removing faults in a program (Row "Debugging") (5) other: other applications such as language transformation [26] and program synthesis [23] (Row "Other"). Note that some techniques proposed in some articles may fall into more than one categories.

Results and Implications. In recent years, with the advances of computing powers and research on constraint solvers and theorem provers, symbolic execution has been developed

TABLE II
TASKS ASSISTED BY SYMBOLIC EXECUTION

Type	Total	Loop
Testing	84 (52.8%)	36 (22.6%)
Verification	42 (26.4%)	26 (16.4%)
Security	25 (15.7%)	12 (7.5%)
Debugging	11 (6.9%)	3 (1.9%)
Other	5 (3.1%)	1 (0.6%)

a lot and been used in various research areas. As shown in Table II, more than 150 articles (151 = 84 + 42 + 25, 95.0%) have been published on applying symbolic execution for software testing, program verification, and security analysis. Given the popularity of symbolic execution, improving it in terms of precision, effectiveness, or scalability would bring substantial benefits to various software engineering or security tasks.

As described in Section I, IDLs may compromise the effectiveness of the proposed techniques using symbolic execution. However, based on the results of our literature-survey study, we find that to what extent loops may compromise the effectiveness of applying symbolic execution depends on the tasks. For example, for program verification, more than 60% of the articles provide various techniques to deal with loops; the remaining articles focus on other aspects of symbolic execution, such as improving constraint solvers to deal with floating-point computation or solving string constraints. The main reason is that verification tasks often require high precision for achieving their objectives, such as proving reachabilities of certain program locations.

Using symbolic execution to enumerate all feasible paths would fail the tasks due to the not only large but infinite search space induced by IDLs. But for some tasks, such as debugging or vulnerability detection in security analysis, loops may not be a major obstacle for two major reasons. (1) When debugging a discovered failure or exploit, a particular failure-inducing path is already known, and bounded iteration can produce satisfactory approximations of loops [22]. The main reason is that paths closer to the failure-triggering path are more likely to be related to the same fault and more error-prone. (2) Abstractions used by some static-analysis approaches abstract away the complexities of unbounded loops, and the analysis always reaches a fixed point to produce a conservative result, such as safe or unsafe in proving memory safety for loops [17].

Among the articles that apply symbolic execution for software testing, nearly half of them (32/84, 42.9%) propose techniques to deal with loops, indicating that loops indeed are also concerns when conducting symbolic execution for software testing. Moreover, research on symbolic execution for software testing can benefit from research on symbolic execution for other research tasks. For example, certain articles on software testing adapt techniques from articles on program verification, such as loop summarization [20] and abstraction [35].

Summary. These results show that loop problems are one of the major kinds of problems that compromise the effectiveness of symbolic execution for tasks that require high precision for

TABLE III
TECHNIQUES TO DEAL WITH LOOPS

Type	Total	Testing
Bounded Iteration	38 (23.9%)	24 (15.1%)
Search-Guiding Heuristics	16 (10.1%)	8 (5.0%)
Loop Summarization	8 (5.0%)	3 (1.9%)
Abstraction	7 (4.4%)	3 (1.9%)
Loop Invariant	4 (2.5%)	0 (0.0%)
Other	3 (1.9%)	0 (0.0%)

achieving their objectives, such as verification and testing; for tasks whose objectives focus on a small set of specific paths, such as debugging or vulnerability detection, loop problems may be alleviated with bounded iteration and abstraction.

C. RQ1.2: Techniques to Deal With Loops

Table III shows loop techniques in our literature-survey study. Column "Total" shows the number of articles that apply symbolic execution for any purpose, and Column "Testing" shows the number of articles that apply symbolic execution for software testing. Each row shows the number of articles that provide certain techniques to deal with loops.

As shown in Table III, most of these techniques fall into four categories: (1) bounded iteration, (2) search-guiding heuristics, (3) loop summarization, and (4) abstraction. Bounded iteration bounds either loop iteration or input range to make the whole search space finite, thus addressing the infinite-loop problem with the loss of completeness. Search-guiding heuristics attempt to guide symbolic execution to focus on exploring paths that are more likely to achieve certain objectives faster (such as structural coverage), thus preventing symbolic execution from being stuck in loops. Loop summarization summarizes a loop into a set of formulas for addressing infinite loops in a program. These formulas can be solved by using constraint solvers to determine whether the objective can be achieved, such as reachability of certain program locations. Abstraction uses an abstracted model of program states. Although this model loses some information, the model is more compact and easier to manipulate. To maintain soundness, the analysis must produce a result that would be true regardless of the information in the abstracted-away state components. Such techniques can be used to model the states of loop iterations, handling infinite loops with symbolic execution.

Results and Implications. Bounded iteration and search-guiding heuristics have relatively low analysis cost, and yet are incomplete or unsound. Loop summarization and abstraction have relatively high analysis cost (facing scalability issues on large programs), but could be used for proof. Understanding the advantages and disadvantages of these techniques can help better make design choices of adapting these techniques to enhance symbolic execution. For example, for verification tasks that require completeness, it is better to choose loop summarization or abstraction. For testing tasks that focus on structural coverage, bounded iteration or search-guiding heuristics are a good fit, since these two techniques can prevent symbolic execution from being stuck in loops. In addition, we could even adapt different techniques to complement each other. For example, for software testing, we could achieve general structural coverage by employing search-guiding heuristics to avoid being stuck in certain loops. For certain program locations that are likely to trigger runtime failures (such as exceptions or assertion violations), we could use loop summarization or abstraction to verify their reachability.

Among these 72 loop articles, we further look into the subject programs used in their evaluations. We find that each article may use one or more kinds of subject programs in their evaluations, and these subject programs are mostly benchmark programs (34/72, 47.2%) and real-world applications (42/72, 58.3%[3]). These results show that loop techniques have been used and evaluated not just on small programs used for illustration purposes, but also on real programs that have practical usage.

From Table III, we can also see that the most widely adopted techniques are bounded iteration: 52.7% (38/72) of the loop articles and 66.7% (24/36) of the loop articles for software testing, and search-guiding heuristics: 22.2% (16/72) of the loop articles and 22.2% (8/36) of the loop articles for software testing. In addition, both of these techniques are widely used for the tasks of software testing, security analysis, and verification. For software testing and security analysis, these two techniques are used to prevent symbolic execution from being stuck in IDLs, and spend more efforts in exploring other parts of the programs to improve structural coverage. For verification, some verification techniques consider bounds to limit their state space, such as bounded model checking [13], and thus bounded iteration is also naturally used to deal with unbound loops in limiting the number of paths. Similar to testing and security analysis, in verification, search-guiding heuristics are also used to improve coverage for symbolic execution. Although these lightweight techniques (bounded iteration and search-guiding heuristics) are incomplete when solving some problems posed by loops, they could still be applicable and effective on various tasks, including verification tasks. We should not rule out such techniques in our design choices without deeper investigation of the programs under analysis, even if our tasks require high precision (such as verification tasks).

Summary. Loop techniques have been used for various tasks and evaluated on real programs. Among these techniques, Bounded iteration and search-guiding heuristics are the most widely used loop techniques across various tasks. When the bounded assumption is acceptable, these techniques can also be used for tasks that require high precision, such as verification tasks.

III. OPEN-SOURCE APPLICATION STUDY

In this section, we present the details of the empirical study on open-source applications and its implications. We further propose guidelines based on our findings.

[3]Among the 72 articles, 9 articles use industrial programs, 13 articles use programs of operating systems, and 20 articles use open-source programs [4].

A. Subjects and Study Setup

To conduct the study, we choose the DSE tool, Pex [48], a state-of-the-art test-generation tool developed by Microsoft Research. The main reasons are (1) Pex has been applied internally in Microsoft to test core components of the .NET runtime infrastructure and finds serious faults [48]; (2) Pex can automatically instrument .NET programs for generating test inputs; (3) Pex is integrated with various kinds of search strategies [48], [54].

In this study, we select subject applications from two sources: (1) 4 open-source applications from CodePlex [2] and 4 from GitHub [3], both of which are the largest application hostings for open-source software written in .NET languages; (2) 8 open-source applications from two categories (4 from each category): application parsers and data structures/algorithms. Some previous studies report that several types of open-source applications have loop problems for compromising the effectiveness of their techniques, such as parsers [20]. Since these applications are not .NET programs that can be instrumented by Pex, we select the same type of applications from open-source .NET programs. Moreover, applications of data structures/algorithms tend to have many IDLs, and many studies extract benchmarks from these applications.

Random Sampling. We randomly select 8 applications from the top 100 most-downloaded or most-watched applications from open-source application hostings: 4 from CodePlex and 4 from GitHub. We base our selection on applications tagged as being written in the C# language. At the time of our study, CodePlex had 32,477 such applications in total, and GitHub listed the top 200 such applications. Applying Pex to explore all loops in all the applications and studying the results would not be feasible in reasonable time. Therefore, we sample the dataset of the top 100 applications, randomly choosing one application out of these applications at a time.

Sampling from Specific Categories. To sample applications from the categories of application parsers and data structures/algorithms, we use the keywords *parsers*, *data structure*, and *algorithm* to search the repositories and rank the matched applications by download counts and watch counts. For each category, we manually inspect the ranked applications in order and select 4 applications from each repository.

Selection Criteria. For each chosen application, we manually inspect the application description and the source code to determine whether it is suitable for our study. We prune the applications based on three criteria. (1) Applications whose majority parts are UI-related or environment-dependent (such as database-related). These applications typically have many external-method calls to libraries of UI or database, and Pex has limitations in handling these external-method calls. (2) Applications that are built using ASP.NET, WPF, or other web-related and mobile-related frameworks. These applications have a substantial part of HTML or XAML code that is used to craft web pages or views, and most of their methods are written to accept framework-specific objects as inputs, such as objects for HTTP responses. We do not focus on web pages, and Pex has limited capabilities in generating test inputs for framework-specific objects. (3) Applications that have concurrency behaviors introduced by multi-threaded programming, such as web servers that spawn threads for clients and maintain queues for threads. Concurrency behaviors bring extra difficulties for Pex and are not the focus of our study.

TABLE IV
STATISTICS OF OPEN-SOURCE APPLICATIONS

Name	# Class	# LOC	# Files
Confuser	914	52,830	301
DotNetZip	697	47,147	118
PSDPlugin	54	3,750	34
Wix	695	295,827	1,009
Mono.Cecil	615	31,750	225
GitSharp	1,101	84,745	756
Spine-Runtimes	34	1,819	19
TweetSharp	166	10,688	54
DSA	60	5,155	51
NGenerics	1,005	53,529	925
QuickGraph	319	34,196	536
Algorithmia	114	7,513	82
Commandline	183	7,431	125
HTMLAgilitypack	70	8,859	55
SharpNLP	219	24,438	217
Sprache	123	2,249	43
Total	6,369	671,926	4,550

We keep sampling applications until we find 16 applications that satisfy our selection criteria. In particular, for each chosen application, we download the most recent source code from the corresponding source repository and try to build the application. For applications that cannot be built directly, we search for dependent libraries and provide specific environment settings. If we still fail to build the applications, we exclude the applications for our study. In total, we consider 74 (34 from CodePlex and 40 from GitHub) applications until we have a set of 16 applications for our study. Table IV shows the statistics of these applications.

Study Design. We first identify loops in the subject programs, and classify them as Fixed-Iteration Loops (FILs) or Input-Dependent Loops (IDLs). The results of the classification show the distribution of FILs and IDLs, and provide us insights on how to automatically identify IDLs and FILs for further analysis. Moreover, nested loops may cause loss of precision for loop-invariant techniques or summarization techniques [42], and pose more challenges in the scale of path explorations. Thus, we also identify the nested loops among the studied loops. These classification results answer RQ2.1.

Among the IDLs, we manually inspect the code to determine whether these IDLs have side effects on variables that are used to decide subsequent branches. We refer to such IDLs as IDLCs (denoting IDLs affecting Coverage). We then apply Pex (reflecting search-guiding heuristics) and Pex with bounded iteration (referred to as Pex-bounded) to explore the identified IDLCs. Our study focuses on only IDLCs since IDLs that do not compromise coverage of subsequent branches can be easily handled by bounded iteration and search-guiding heuristics, while symbolic execution faces challenges to reason

TABLE V
LOOP STATISTICS AND CLASSIFICATION FOR RQ2.1

Name	# Loops	# IDL	# FIL	# N.
Confuser	441	373	68	26
DotNetZip	425	254	171	65
PSDPlugin	69	66	3	9
Wix	320	286	34	19
Mono.Cecil	246	245	1	13
GitSharp	175	173	2	21
Spine-Runtimes	57	56	1	8
TweetSharp	28	26	2	1
DSA	82	82	0	17
NGenerics	228	121	107	39
QuickGraph	142	84	58	6
Algorithmia	100	60	40	12
Commandline	35	33	2	2
HTMLAgilitypack	71	71	0	1
SharpNLP	259	259	0	45
Sprache	8	7	1	0
Total	2,686	2,196	490	284

about the side effects of IDLCs [20], [40], [42]. Pex by default uses the mixed search strategy that combines different search strategies: depth-first search, random search, and fitness search [54], making Pex a representative tool for search-guiding heuristics. Pex-bounded bounds the iteration counts for the explored loops, thus reducing the search space. In our study, we choose 2 as the loop-iteration bound. Such bound makes sure that the back edge of the loop is traversed at least once and is commonly used in both dynamic techniques and static techniques [19], [35]. The results of exploring IDLCs answer RQ2.2.

Finally, by studying the IDLCs that neither Pex or Pex-bounded can cover, we identify the major challenges and propose guidelines to address these challenges.

B. RQ2.1: Statistics and Classification of the Studied Loops

To identify loops in the applications, we develop a static-analysis tool that detects loops using back edges of dominators [45]. Among the detected loops, we manually classify them as FILs or IDLs and identify nested loops. Table V shows the statistics of the studied loops and the results of loop classification. Column "# Loops" shows the number of studied loops rather than the number of all loops for each application. In our study, every application has at least two or more sub-projects or components that would be compiled as a Dynamic Link Library (DLL) or an executable. Some DLLs or executables may contain more than 500 loops (e.g., one DLL of DotNetZip has 640 loops). Manually inspecting all these loops and studying the results would not be feasible in reasonable time. Thus, for components that contain more than 100 loops, we randomly choose 100 loops to study, and the total number of loops studied for each application is shown in Column "# Loops". Column "# IDL" shows the number of studied loops that are IDLs. Column "# FIL" shows the number of studied loops that are FILs. Column "# N." shows the number of studied loops that are nested loops.

Results and Implications. As shown in the results, in some applications, a substantial portion of loops are FILs. For example, 46.9% (107/228) of the loops in NGenerics are FILs. Most of these FILs are from test cases that construct objects of fixed-size matrix. However, most of the loops (81.8%, 2196/2686) in the studied applications are IDLs. The pervasiveness of IDLs indicates that there is a strong need to provide techniques for dealing with unbound loops. Moreover, certain techniques that keep unrolling loops, such as the Depth-First Search (DFS) strategy, should be applied with other search strategies to avoid getting stuck in IDLs.

Loop summarization [20], [40] uses patterns to identify IDLs and symbolic analysis to infer the relationship between program inputs and iteration counts. Thus, to understand the effectiveness of their techniques, we study how to determine whether a loop is an IDL and how to infer its iteration count using symbolic execution. Based on our study, we find out that the iteration counts of IDLs mainly depend on two types of inputs: (1) **program inputs**: input parameters of the containing method or fields of the containing method's receiver objects; (2) **environmental inputs**: variables storing inputs returned from external method calls, such as calls related to files or random generators. Previous work [20], [40] based on patterns and symbolic analysis can detect IDLs that depend on program inputs of primitive types, but further work is needed to detect IDLs that depend on fields of receiver objects. In addition, our previous technique [52] can be used to compute the data dependencies between loops and external method calls, detecting IDLs that depend on values returned from external method calls.

FILs can be unrolled statically and thus can be treated as a series of branches in path explorations. Typically, the iteration counts of FILs depend on numerical constants or the length of fixed-size arrays. However, we find out that the iteration counts of some FILs may depend on static variables or return values of external method calls. For example, an FIL may depend on the number of values in an *enum* type through the external method call of `Enum.GetNames(typeof(SomeEnum)).Length`. Thus, static analysis on possible values of static variables and certain external method calls can be leveraged to determine the number of iterations for such FILs.

As shown in Table V, nested loops appear more often in algorithm-based applications, such as DotNetZip, NGenerics, and SharpNLP. Since nested loops are shown to pose challenges for techniques of loop invariants or summarization, loop invariants or summaries for such applications may be provided by developers to improve the precision and scalability of the applied techniques.

Summary. Most of the loops (81.8%: 2196/2686) in the studied applications are IDLs. Techniques of identifying IDLs and their iteration counts require analysis on not only program inputs, but also environmental inputs. Algorithm-based applications tend to have more nested loops that may affect effectiveness of loop techniques.

TABLE VI
RESULST OF RQ2.2

Name	# IDLC	# EXP	# Pex	# Bound
Confuser	33	25	12	12
DotNetZip	27	16	9	11
PSDPlugin	4	2	1	0
Wix	47	31	19	17
Mono.Cecil	47	45	37	32
GitSharp	68	39	33	30
spine-runtimes	8	6	5	5
TweetSharp	8	6	2	2
DSA	11	11	11	7
NGenerics	11	11	3	3
QuickGraph	7	7	2	2
Algorithmia	2	0	0	0
Commandline	14	14	14	14
htmlagilitypack	15	13	13	13
sharpnlp	79	40	25	24
sprache	0	0	0	0
Total	381	266	186	172

C. RQ2.2: Loop Explorations

We apply Pex and Pex with bounded iteration to explore the methods containing IDLCs (in short as MCLs) identified in our previous step and the methods that invoke an MCL if the MCL returns a value that may compromise coverage. We use Pex's default bounds of various resources, such as the running time, constraint-solving time, and number of paths.

As shown in our previous work [52] and other studies [16], [30], structural test generation faces two major problems: object-creation problems (OCPs) and external-method-call problems (EMCPs). Since these problems are not our study focus, we manually provide factory methods to address OCPs, and mock external methods to address EMCPs. For example, we provide factory methods to guide Pex to generate matrix objects using integer arrays, and mock calls to file systems using string-matching functions.

Table VI shows the statistics of the IDLCs and the results of loop explorations. Column "# IDLC" shows the numbers of studied loops that are IDLCs. We prune the IDLCs that we cannot provide factory methods or mock objects: 31 for dependencies on GUI objects, 43 for Input/Output (IO) stream objects, 31 for complex objects that are difficult to generate using factory methods, and 10 for various problems that prevent Pex from exploring the code, such as failures on instrumenting the DLLs. Column "# EXP" shows the IDLCs that are explorable by Pex with manually provided factory methods and mock objects. Columns "# Pex" and "# Bound" show the number of IDLCs that can be covered by Pex and Pex with bounded iteration, respectively. Note that covering an IDLC is to cover the subsequent branches that use variables whose values are modified by the IDLC.

Results and Implications. Overall, for the 266 IDLCs that are explorable with our provided factory methods and mock objects, Pex and Pex-bounded achieve similar results for covering the IDLCs (69.9%, 186/266, and 64.7%, 172/266). Both techniques achieve good coverage for DSA as expected, since such library applications tend not to have many callers on

```
1  public static Version ParseVersion(string ver) {
2      ...
3      int dotCount = 0;
4      for (int i = 0; i < ver.Length; i++) {
5          char c = ver[i];
6          if (c == '.') dotCount++;
7          ...
8      }
9      ...
10     if (dotCount == 0) { ver = ver + ".0"; }
11     else if (dotCount > 3) {
12         string[] verSplit = ver.Split('.');
13         ver = String.Join(".", verSplit, 0, 4);
14     }
15     ...
16 }
```

Fig. 1. A simplified code snippet from Wix

```
1  public static void Main(String[] args) {
2      ...
3      for (int i = 1; i < args.Length; i++) {
4          switch (args[i]) {
5              case "−keep": keepOriginal = true; break;
6              case "−f": force = true; break;
7              case "−v": verbose = true; break;
8              default: throw new ArgumentException(args[i]);
9          }
10     }
11     string fname = args[0];
12     bool decompress = (fname.ToLower().EndsWith(".bz") ||
           fname.ToLower().EndsWith(".bz2"));
13     string result = decompress ? Decompress(fname, force) :
           Compress(fname, force);
14     if (result==null) {
15         Console.WriteLine("No action taken. The file already exists.");
16     }
17     else {
18         if (verbose) {
19             ...
20             if (decompress) { ... } else { ... }
21         }
22         if (!keepOriginal) { ... }
23     }
24 }
```

Fig. 2. A simplified code snippet from DotNetZip

methods that contain IDLs to do computations. For randomly-chosen applications, both techniques can cover about 65% of the IDLCs. For NGenerics and Quickgraph, both techniques obtain worst results due to validation on input objects (described in Section IV). We next provide explanations and examples to compare the performance of these two techniques.

Bounded iteration reduces the search space for Pex. However, under some cases, bounding the iterations of a loop would cause some subsequent branches not to be covered. Figure 1 shows a simplified example from Wix. To cover the true branch (`dotCount > 3`) at Line 11, we need the loop at Line 4 to have at least 4 iterations. Thus, by bounding the number of loop iterations to 2, Pex-bounded cannot cover the true branch at Line 11, while Pex without bounded iteration can cover both branches at Line 11.

On the other hand, if we do not bound the iterations of IDLs, Pex may unroll the loop too many times, and would not achieve the coverage of subsequent branches before running out of resources. Figure 2 shows an example that Pex-bounded achieves better coverage than Pex. The iteration count of IDL at Line 3 depends on the number of arguments in `args`. In each iteration of this IDL, the `switch` statement at Line 4

TABLE VII
OUR MAJOR FINDINGS ON LOOP PROBLEMS AND THEIR IMPLICATIONS

RQ	Findings of Literature-Survey Study	Implications
RQ1.1	**(F.1)** In recent 10 years, more than 150 articles published in major conferences of various areas are related to symbolic execution and tasks assisted by symbolic execution.	Given the popularity of symbolic execution, improving it in terms of precision, effectiveness, or scalability would bring substantial benefits to various tasks, such as software testing, program verification, and security analysis.
	(F.2) Few articles on debugging or security analysis propose specific techniques to address loop problems, while more than 61.9% (26/42) of the articles provide various techniques to deal with loops; nearly half of the articles on software testing (32/84, 42.9%) propose techniques to deal with loops.	To what extent loops compromise the effectiveness of applying symbolic execution depends on the tasks assisted by symbolic execution; loop problems are one of the major problems when conducting symbolic execution for software testing.
RQ1.2	**(F.3)** 75.0% of the articles that provide techniques to address loop problems adopt bounded iteration and search-guiding heuristics.	Bounded iteration and search-guiding heuristics are the most widely used loop techniques across various tasks due to their ease of implementation and effectiveness in handling loops for different objectives of these tasks.
	(F.4) For tasks of software testing and security analysis, bounded iteration and search-guiding heuristics are used to prevent symbolic execution from being stuck in IDLs; for tasks of program verification, these two techniques are used to limit the state space.	Lightweight techniques (bounded iteration and search-guiding heuristics) could be applicable and effective on various tasks, including verification tasks (when the bounded assumption is acceptable).
RQ	**Findings of Empirical Study**	**Implications**
RQ2.1	**(F.5)** Most of the loops (81.8%, 2196 / 2686) in studied applications are IDLs.	Techniques that keep unrolling loops, such as the Depth-First Search (DFS) strategy, should be applied with other techniques to avoid getting stuck in loops.
	(F.6) The iteration counts of IDLs depend on two types of inputs: program inputs (e.g., primitive values or object fields) and environmental inputs (e.g., return values of external method calls); existing techniques mainly focus on IDLs that depend on program inputs.	To improve identification of IDLs, extra symbolic analysis should be performed on values returned or modified by external method calls.
	(F.7) Nested loops pose challenges to compromise the precision and scalability of loop techniques, and appear more often in algorithm-based applications.	For algorithm-based applications, developers may specify loop invariants or summaries to improve the precision of the applied loop techniques.
RQ2.2	**(F.8)** IDLs that do not compromise the coverage of subsequent branches can be easily handled by bounded iteration and search-guiding heuristics. These two loop techniques can address loop problems caused by about 65% of IDLCs.	In general, bounded iteration and search-guiding heuristics can effectively address loop problems. However, in some cases, these techniques still face challenges in covering the branches decided by IDLCs.

results in 4-way branches, each of which would cause a local variable (i.e., `force`, `verbose`, and `keepOriginal`) to become true or throw an exception. At the later part of the example, branches at Lines 14, 18, and 22 depend on the values of `force`, `verbose`, and `keepOriginal`, respectively. If no specific bound is provided for the IDL at Line 3, the combinations of branches taken inside the IDL grow exponentially when the iteration count increases. Moreover, the value of `args[0]` used in Lines 12 and 13 makes the number of paths grow even more drastically. Thus, Pex fails to achieve coverage of all the branches at Lines 14, 18, and 22 due to path explosion. When Pex is applied with bounded iteration, the number of arguments in `args` is constrained to be 2, and Pex can achieve full coverage of the example within the given bounds of resources.

Summary. In general, bounded iteration and search-guiding heuristics can effectively address loop problems caused by IDLs, while in some cases these two techniques still face challenges in dealing with IDLCs (about 35%).

IV. CHALLENGES AND GUIDELINES

Our characteristic studies identify the widely used loop techniques and show their effectiveness. The major findings and implications of our studies are summarized in Table VII. For the IDLCs that cannot be covered by either Pex or Pex-bounded, we further inspect these IDLCs and identify several major challenges.

Data Structures. As shown in Table VI, both Pex and Pex-bounded achieve poor IDLC-coverage results for the applications NGenerics (3/11) and Quickgraph (2/7). The main reason is that the IDLCs in these data structure/algorithm applications require input objects to pass the validation, i.e., using `repOk` to guard against invalid objects. Figure 3 shows such an IDL that checks whether a matrix is a symmetric matrix. These validations typically use IDLs to validate each item of a data-structure object, and generate enormous constraints after the validation. For example, objects of the symmetric matrix in NGenerics require a validation for checking whether values in a 2-dimensional array conform to the symmetric constraints. Graphs of various edges and vertices require validation of vertices' existence before edges can be added and different shapes of graphs impose further validations that involve both vertices and edges. In these cases, the validation constraints are combined with other constraints collected from the IDLCs to form very long constraints, which cause the constraint solver to run out of time in solving these constraints.

Guideline: *For applications that impose heavy validation on input values, a separate data generator that generates valid objects may be employed and only the constraints that lead to valid objects should be combined with the constraints collected from IDLCs.*

Path Explosion. As shown in Section III-C, bounded iteration may cause certain subsequent branches not to be

```
1  public bool IsSymmetric {
2      get {
3          if (noOfRows == noOfColumns) {
4              for (var i = 0; i < noOfRows; i++) {
5                  for (var j = 0; j < i; j++) {
6                      if (GetValue(i, j) != GetValue(j, i))
7                          return false;
8                  }
9              }
10             return true;
11         }
12         return false;
13     }
14 }
```

Fig. 3. An IDL that performs data validation from NGenerics

covered, and not bounding IDLs can cause symbolic execution to suffer from path explosion and constraint explosion. When a symbolic-execution tool negates a branch that decides the loop termination (i.e., a loop guard), the loop iteration increases by a certain number, depending on the output from the constraint solver. If each iteration of the loop contains many branches, such as the example in Figure 2, increasing the loop iterations introduces many more constraints and grows the search space exponentially. The current fitness-heuristic technique [54] does not treat loop guards differently, and thus computes the fitness values for loop guards using the same way as other branches. Such fitness values increase the probabilities to search paths in later iterations instead of sufficiently searching the paths in earlier iterations.

Guideline: *Search-guiding heuristics should assign lower probabilities on loop guards than other branches, and branches collected in later iterations should be given lower probabilities than branches collected in earlier iterations.*

Complex Loops. Loop summarization [20], [40] can alleviate the loop problems only when the IDLs can be summarized using induction variables. However, many loops are complex and cannot be summarized using induction variables. For the example in Figure 1, in the loop body, `dotCount++` is guarded by a condition `c == '.'`, and thus `dotCount` may be modified in only some of the iterations. In this case, most of these summary-based techniques would not consider `dotCount` as an induction variable and cannot generate useful summaries. Moreover, some complex IDLs that contain nested loops, such as the IDL in Figure 3, are not summarizable using induction variables.

Guideline: *For complex IDLs that interleave nested loops and branches, a test-generation tool may identify such loops and report to developers; developers can provide manually specified loop invariants or summaries to assist the tool in addressing the loop problems.*

V. Threats to Validity

Threats to External Validity. In our literature-survey study, we choose articles based on a comprehensive bibliography of articles on symbolic execution and its applications [1]; such bibliography has been collected by a third party. Although these collected articles may not be complete, the 159 studied articles can be considered as a representative set of articles on symbolic execution, since these articles are published in a wide range of venues, including conferences in software engineering, systems, security, software verification, programming languages, and database. We may further reduce the threat by including more articles from scientific publishers, such as ACM, IEEE, and Springer.

In our empirical study, 16 selected open-source applications are realistic and reasonable, within our affordable efforts besides our significant effort spent for the literature-survey study of 159 articles. In particular, we study these published papers carefully to classify loop techniques, and select subject applications of data structures/algorithms and parsers based on the study results from the literature. Applications of data structures/algorithms and parsers are often libraries and do not have many callers, and thus we further randomly sample applications from open-source project hostings. These randomly selected applications tend to be applications using libraries, reducing the bias of our subject selection. In addition, although previous test-generation studies [16] investigate the scale of 100 applications, these studies sample only 40 classes for manual inspection of problems. Note that we manually study about 2600 loops and provide in-depth analysis of loops that may compromise coverage. Such study is time-consuming to conduct, and thus constrains the scale of our studies.

Threats to Internal Validity. Subjectiveness in the failure classification is inevitable due to the large manual effort involved in both the literature-survey study and the empirical study. In addition, there also might be human errors in collecting application statistics (such as # LOC and # classes) and studying the exploration results of Pex. These threats are mitigated by double-checking all manual work. We ensure that the results are individually verified and agreed upon by at least two authors. These threats could be further reduced by involving third-party people who have experiences on software testing and symbolic execution to verify our results.

VI. Discussion and Future Work

Generalization to Other Test-Generation Techniques. Although our current study focuses on test-generation tools based on DSE, some of our findings can be generalized to other test-generation techniques. For example, the challenge of generating objects of specific data structures are applicable to both random and constraint-based test generation. Both of these techniques would take substantial time before they can produce valid objects, while employing a separate data generator that generates only valid objects but allows variants on some fields of the objects can improve the effectiveness of test generation. Similarly, the challenge of path exploration and our guideline are applicable to these techniques. For example, random test generation may give lower probabilities and constraint-based test generation may use fixed values or assign lower priorities to mutate variables that may increase loop iterations.

Future Directions to Improve Loop Techniques. Our studies show the effectiveness of existing techniques (bounded

iteration and search-guiding heuristics), and identify challenges faced by these techniques. Based on the findings of our studies, there are several directions to improve loop techniques. (1) **Identification of IDLs**. Symbolic analysis should be applied on external method calls [50], [52] to improve identification of IDLs, and modelling of these methods may be used to improve the precision of inferring the iteration counts of IDLs. (2) **Bounded heuristics**. Bounded iteration may be used to first bound the search space, and search-guiding heuristics are then used to guide the search of paths inside the bounded space. If the objective cannot be satisfied in the bounded space, the bound may be increased. (3) **Mixed techniques**. Bounded iteration and search-guiding heuristics are generally very effective in handling IDLs that require a few iterations to cover subsequent branches of the IDLs. after several iterations, loop summarization can be used to compute summaries for not-covered IDLs. (4) **Cooperative analysis**. For complex loops, such as nested loops or loops that cannot be summarized by loop summarization, tools may report such loops to users and present the related not-covered branches to obtain developers' guidance.

Object Generation and Mock Objects. In our empirical study, we observe that some complex objects have more than 10 fields that need to be set by symbolic values, and these fields may in turn be complex objects. The current mechanism of factory methods has limitations in supporting the creation of such objects, and often results in lots of constraints during the process of creating objects. In future work, we plan to investigate how to simplify the object-creation mechanism to better assist test-generation tools. In addition, we observe that there are many IDLCs that require mocking complex objects to simulate environment dependencies, such as file systems, IO streams, and GUI objects. We are able to provide mock objects for file systems, but could not mock IO streams or GUI objects due to their unique challenges. Mocking IO streams requires modelling the pointer for reading data and the pointer's movement, while mocking GUI objects requires modelling view objects organized in layers and their parent-children relationships. In future work, we plan to investigate how to construct *Parameterized Mock Objects*s [39], [44] to address these challenges.

VII. Related Work

Studies on Test Generation. Lakhotia et al. [37] conduct an empirical study on applying test-generation tools CUTE [41] (a symbolic-execution-based tool) and AUSTIN [36] (a search-based tool) to achieve branch coverage of C programs. Fraser et al. [16] present a study of applying a search-based tool EvoSuite [15] on a set of open-source applications. They identify that dependencies on the environment inhibit high coverage achieved by test-generation tools. Kim et al. [32] propose a distributed concolic algorithm [33], and present an empirical study to show that their technique achieves several orders-of-magnitude increase in speed of test generation compared to concolic testing. All these studies focus on the coverage or scalability of testing the whole applications, while our work provides in-depth studies to identify challenges on dealing with loops and presents guidelines on addressing the challenges.

Heuristics for Path Exploration. To address the path explosion caused by loops, some DSE tools bound the loop iterations [19], or use heuristics to guide path explorations [48], such as the fitness-heuristic technique [54]. The fitness-heuristic technique computes a fitness value to measure how close an already discovered feasible path is to a manually specified test target (e.g., a non-covered branch), and guide symbolic execution to take branches that have better fitness values. In our studies, we apply these two techniques on open-source applications, compare their effectiveness, and identify challenges faced by these two techniques.

Loop Summarization. Loop-summary techniques [20], [40] define an extra symbolic value for the iteration count of the executed loop, and collect constraints on program variables that have relationship to the iteration count of the loop. These constraints are added to path conditions for steering further path explorations, addressing the issue of unrolling the loop without achieving higher coverage. More recent research [42] extends the summarization of iteration counts by using symbolic variables to represent different paths taken inside the loop, enabling summarization of the relationship between program variables and the branches inside the loop. These techniques require detection of induction variables whose values are modified by a constant value or constant times for each loop iteration. However, as shown in our empirical study, there are also many other complex situations that interleave nested loops or conditional branches, not summarizable by using induction variables.

There also exist static-analysis techniques [6], [8], [24] for automatic loop-invariant generation and summarization. These static-analysis techniques may not be scalable for real-world complex applications, and face challenges in handling many runtime features of real-world applications, such as external method calls and indirect method calls via function pointers.

VIII. Conclusion

This paper has presented the first characteristic studies on loop problems for structural test generation, with the focus on test-generation tools based on DSE. Our characteristic studies consist of two parts: a literature-survey study of 159 published articles and an empirical study on 16 open-source applications. Our two-phase characteristic studies focus on bounded iteration and search-guiding heuristics, two most widely adopted techniques to deal with loop problems. Our studies find that these two loop techniques can address about 65% of IDLs that have side effects on variables used to decide subsequent branches. Our studies further identify challenges that compromise the effectiveness of these loop techniques and provide guidelines on how to address these challenges. Our findings and implications provide valuable guidelines for future research on loop problems for structural test generation.

ACKNOWLEDGMENT

This work is supported in part by NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603, a NIST grant, a Microsoft Research Software Engineering Innovation Foundation Award, and NSF of China No. 61228203. We thank Patrice Godefroid for his valuable feedback on an early version of the work described in this paper.

REFERENCES

[1] A bibliography of papers on symbolic execution technique and its applications. https://sites.google.com/site/symexbib/.
[2] CodePlex. http://www.codeplex.com/.
[3] GitHub. https://github.com/.
[4] Project website. https://sites.google.com/site/asergrp/projects/loopstudy.
[5] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
[6] T. Ball, O. Kupferman, and M. Sagiv. Leaping loops in the presence of abstraction. In *Proc. CAV*, pages 491–503, 2007.
[7] C. Barrett and C. Tinelli. CVC3. In *Proc. CAV*, pages 298–302, 2007.
[8] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds (extended abstract). In *Proc. CAV*, pages 1–12, 1996.
[9] J. Caballero, P. Poosankam, S. McCamant, D. Babić, and D. Song. Input generation via decomposition and re-stitching: finding bugs in malware. In *Proc. CCS*, pages 413–425, 2010.
[10] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224, 2008.
[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proc. CCS*, pages 322–335, 2006.
[12] A. Chaudhuri and J. S. Foster. Symbolic security analysis of Ruby-on-Rails web applications. In *Proc. CCS*, pages 585–594, 2010.
[13] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, pages 168–176, 2004.
[14] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.
[15] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proc. ESEC/FSE*, pages 416–419, 2011.
[16] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proc. ICSE*, pages 178–188, 2012.
[17] P. Godefroid and J. Kinder. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *Proc. ISSTA*, pages 1–12, 2010.
[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.
[19] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*, 2008.
[20] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proc. ISSTA*, pages 23–33, 2011.
[21] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proc. POPL*, pages 43–56, 2010.
[22] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proc. ICSE*, pages 55–64, 2010.
[23] S. Gulwani. Dimensions in program synthesis. In *Proc. PPDP*, pages 13–24, 2010.
[24] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proc. POPL*, pages 127–139, 2009.
[25] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.
[26] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: a MapReduce query optimizer. In *Proc. EuroSys*, pages 251–264, 2010.
[27] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *Proc. CAV*, pages 758–766, 2012.
[28] J. Jaffar, J. Navas, and A. Santosa. Unbounded symbolic execution for program verification. In *Proc. RV*, pages 396–411, 2012.
[29] J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *Proc. CP*, pages 454–469, 2009.
[30] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: object capture-based automated testing. In *Proc. ISSTA*, pages 159–170, 2010.
[31] W. Jin and A. Orso. BugRedux: reproducing field failures for in-house debugging. In *Proc. ICSE*, pages 474–484, 2012.
[32] M. Kim, Y. Kim, and G. Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *Proc. ICST*, pages 340–349, 2012.
[33] Y. Kim and M. Kim. SCORE: a scalable concolic testing tool for reliable embedded software. In *Proc. ESEC/FSE*, pages 420–423, 2011.
[34] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
[35] D. Kroening, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via program execution. In *Proc. ICFEM*, pages 224–238, 2004.
[36] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *Proc. GECCO*, pages 1759–1766, 2008.
[37] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *J. Syst. Softw.*, 83(12):2379–2391, 2010.
[38] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Proc. ASE*, pages 515–519, 2009.
[39] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proc. AST*, pages 149–153, 2009.
[40] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. ISSTA*, pages 225–236, 2009.
[41] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.
[42] J. Strejček and M. Trtík. Abstracting path conditions. In *Proc. ISSTA*, pages 155–165, 2012.
[43] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: Guided path exploration for efficient regression test generation. In *Proc. ISSTA*, pages 1–11, 2011.
[44] K. Taneja, Y. Zhang, and T. Xie. MODA: Automated test generation for database applications via mock objects. In *Proc. ASE*, pages 289–292, 2010.
[45] R. Tarjan. Testing flow graph reducibility. In *Proc. STOC*, pages 96–107, 1973.
[46] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proc. OOPSLA*, pages 189–206, 2011.
[47] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, pages 193–202, 2009.
[48] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.
[49] R. Wang, P. Ning, T. Xie, and Q. Chen. MetaSymploit: Day-one defense against script-based attacks with security-enhanced symbolic analysis. In *Proc. USENIX Security*, 2013.
[50] X. Xiao. Problem identification for structural test generation: first step towards cooperative developer testing. In *Proc. ICSE*, pages 1179–1181, 2011. SRC.
[51] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Covana: Precise identification of problems in Pex. In *Proc. ICSE*, pages 1004–1006, 2011. Demo.
[52] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proc. ICSE*, pages 611–620, 2011.
[53] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS*, pages 365–381, 2005.
[54] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. DSN*, pages 359–368, 2009.