5-2015

Mining patterns of unsatisfiable constraints to detect infeasible paths

Sun DING

Hee Beng Kuan TAN

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Mining Patterns of Unsatisfiable Constraints to Detect Infeasible Paths

Sun Ding, Hee Beng Kuan Tan
School of Electrical & Electronic Engineering
Nanyang Technological University, Singapore
{dingsun, ibktan}@e.ntu.edu.sg

Lwin Khin Shar
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg, Luxembourg
lwinkhin.shar@uni.lu

***Abstract*— Detection of infeasible paths is required in many areas including test coverage analysis, test case generation, security vulnerability analysis, etc. Existing approaches typically use static analysis coupled with symbolic evaluation, heuristics, or path-pattern analysis. This paper is related to these approaches but with a different objective. It is to analyze code of real systems to build patterns of unsatisfiable constraints in infeasible paths. The resulting patterns can be used to detect infeasible paths without the use of constraint solver and evaluation of function calls involved, thus improving scalability. The patterns can be built gradually. Evaluation of the proposed approach shows promising results.**

***Keywords*— Infeasible paths; pattern mining; symbolic evaluation; static analysis; structural testing**

I. INTRODUCTION

An infeasible path is a path in a control flow graph (CFG) that cannot be exercised by any input values. The effectiveness and precision of structured testing techniques could be greatly improved if most of the infeasible paths are detected and automatically excluded. It has been widely discussed the importance of detecting infeasible paths when using structural testing criteria and its acute impact on definition-use dataflow testing [1]. A comprehensive survey of test coverage [2] also pointed out the adverse impact of the inability to detect infeasible statements, branches and paths on the precision of structural testing criteria. In modern test case generators, which generate test cases based on path-oriented criteria through combining static and dynamic analysis [3, 4], the detection of infeasible paths without extensive symbolic evaluation would help avoid significant waste of time spent on such paths. The detection of infeasible paths also plays an important role in other applications of program analysis such as vulnerability detection [5].

There are several ways to detect infeasible paths. Among them, the approach based on constraint propagation offers high precision and is commonly used [5]. This approach forms a path constraint by joining predicates along a path, and uses a constraint solver to validate the path constraint. A path's infeasibility is determined by the satisfiability of its path constraint [3, 4]. However these approaches are expensive. And due to widespread undecidable constraints [5], these approaches cannot determine the infeasibility of all paths fully automatically. There are also several approaches that apply heuristics or pre-defined code patterns [6] to enhance the efficiency of constraint propagation. But because their heuristics or patterns are all fixed, these approaches would bring in inaccuracy when dealing with complex programs.

In this paper, we propose an approach that analyzes code of real systems to mine patterns of infeasible path constraints. Our approach extracts minimal sets of unsatisfiable predicates from each infeasible path, generalizes the predicates into patterns. The resulting patterns are then used to detect infeasible paths without the use of constraint solver and evaluation of function calls involved. The major contributions in this paper include:

- A novel approach to mine patterns of infeasible paths such that these patterns can detect infeasible paths without the use of constraint solvers.
- The proposed approach mines patterns of infeasible path constraints that have not been explored before. As the mined patterns are not predefined, they can be built gradually based on input paths. Different patterns can be built for code developed using different programming language, executable code and different lifecycle maturity or different intention (e.g.: malware, obfuscated code).
- A prototype tool that implements the above approach.
- A case study that shows promising preliminary results.

II. PATTERNS IN INFEASIBLE PATH CONSTRAINTS

A. Basic Terms

An **infeasible path** is a path in a CFG that cannot be exercised by any external inputs. A path constraint refers to the constraint of a path that is expressed as a conjunction of predicates in which all the derived variables are substituted with their transitive definitions. Additionally, in this paper, **predicates** always refer to non-compounded predicates. Two predicates are mutually **dependent** if they share common variables. A set of predicates in a path is called **a set of dependent predicates** if each predicate is dependent on another predicate in the same set. If for a path, there is a set of dependent predicates which recursively includes all the dependent predicates, then we call this set as a **maximal set of dependent predicates.** There could be several mutual-exclusive maximal sets of dependent predicates along a path.

Recent studies [5, 6] discover that a path being infeasible is usually because its path constraint contains one unsatisfiable sub-constraint that is similar to an unsatisfiable real constraint over boolean/integer/real domains. Based on this finding, we

propose a novel approach to model such unsatisfiable sub-constraints. In the following, for convenience, we shall address these sub-constraints as **unsatisfiable constraints**.

Let P be a set of predicates in a path. An expression e is a **basic expression** if all the predicates in P contain e. A basic expression e' is the **maximal basic expression** (**max-basic expression**) if e' contains all of other basic expressions in P and there is no other basic expression e'' that contains e'. For example: $\{ p_1=(x_1+x_2 > 0), p_2=(x_1+x_2 < 0), p_3=(x_1+x_2+x_3 > 0)\}$ is a set of predicates; (x_1+x_2) is an basic expression as p_1, p_2 and p_3 all contain it. But $(x_1 + x_2 + x_3)$ is not a basic expression because p_1 and p_2 do not contain it.

B. Modelling

In order to model unsatisfiable constraints in infeasible paths as similar unsatisfiable boolean/integer/real constraints, we replace max-basic expressions that do not contain side-effect function-calls (functions that modify its execution context) depending on their types as follows:

1) Boolean types: by free Boolean variables
2) Integer types: by free integer variables
3) Other types: by free real variables.

We call the above replacement a **maximal basic expression to variable transformation** (**mBexpToVar transformation**). For example, applying mBexpToVar transformation to the set of predicates:

$\{ \quad p_1=(objA.f(objB.g(objC.h(z))) \equiv 1)$,
$p_2=(objA.f(objB.g(objC.h(z)))!= 1)\}$,

we replace $objA.f(objB.g(objC.h(z))$ by a real variable x and we get $\{ p_1=(x \equiv 1), p_2=(x != 1)\}$.

We model patterns of unsatisfiable constraints in an infeasible path constraint u as a set of jointly unsatisfiable predicates (if exists) through two relations. The first relation is called **unsatisfiabilty-based abstraction relation,** denoted by **Φ**. It maps a path constraint u to a maximal set of dependent predicates or a singleton of constant predicate, more specifically:

- $u\Phi v \Leftrightarrow u$ is a path constraint; v is a maximal set of dependent predicates or a singleton of constant predicate in u.

This abstraction relationship is defined to mine the potentiality jointly unsatisfiable set of predicates or potentially unstaisfiable predicate in path constraints. For example, if there is a path constraint $u = (n + m \equiv 1) \wedge (n - m \equiv 12) \wedge (7a + b \equiv 0)$, and a set of dependent predicates $v = \{ p_1=(n + m \equiv 1), p_2=(n - m \equiv 12)\}$, we shall have $u\Phi v$.

The second relation is called **unstaisfiabilty-based similarity relation,** denoted by Θ. It maps v which is a maximal set of dependent predicates or a singleton of constant predicate to a set of jointly unsatisfiable boolean/integer/real predicates (if exists) in two steps: (1) apply bMexpToVar transformation to v; (2) extract the minimal set of unsatisfiable predicates from the above result. More specifically:

- $v\Theta w \Leftrightarrow$ suppose s is a set of predicates returned from applying bMexpToVar transformation to v; s should be a set of jointly unsatisfiable boolean/integer/real predicates; then w is the minimal set of unsatisfiable predicates in s.

For example, $\{ p_1=(n + m > 1), p_2=(n + m <1), p_3=(z - y > 0)\}$ $\Theta\{ p_a=(x_1>1), p_b=(x_1 < 1)\}$.

Finally, the composition Ω of Φ and Θ, that is $\Omega = \Phi \circ \Theta$, represents the relation between path constraints and the minimal sets of jointly unsatisfiable predicates. For example, for path constraint $u = (objA.f(objX.g(objY.h(z))) \equiv 1) \wedge ((objA.f(objX.g(objY.h(z)))!= 1) \wedge (n + m \equiv 1)$ with all variables over real domains, and two sets of real predicates v and w, where:

$v=\{ p_1=(objA.f(objX.g(objY.h(z))) \equiv 1,$
$\quad p_2=(objA.f(objX.g(objY.h(z)))!=1)\}$,
$w=\{p_a=(x \equiv 1), p_b=(x != 1)\}$,

there are the relationships: $u\Phi v$ and $v\Theta w$. Hence, we conclude that $u\ \Omega\ w$. Therefore, $w=\{p_a=(x \equiv 1), p_b=(x != 1)\}$ is a set of jointly unsatisfiable real predicates extracted from path constraint u to model a pattern of unsatisfiable constraint in u.

III. MINING PATTERNS

The input of the mining process is a set of paths extracted from the control flow graphs of real systems, denoted as Set_{path}. The output of the mining process is a set of patterns of unsatisfiable constraints extracted from infeasible paths among Set_{path}, denoted as $Set_{pattern}$. For each path in Set_{path}, the path constraint u is extracted and processed with three steps that show in Figure 1. The resulting sets of predicates from the last step are then stored in the set $Set_{pattern}$.

Input: Set_{path}, a set of paths
Output: $Set_{pattern}$, a set of patterns
Step (1) Compute the set $s_1=\{v \mid u\Phi v\}$.
Step (2) Compute the set $s_2=\{w \mid u\Phi v$ and $v\Theta w\}$
Step (3) Generalize patterns from s_2, and store in $Set_{pattern}$

Fig. 1. Process of pattern mining

In Step 1, for each input path, its path constraint u is extracted out first. Then we calculate the corresponding unsatisfiabilty-based abstraction relation. More specifically, the set $s_1 = \{v| u\Phi v\}$ is constructed from the following two sub-steps: (i) $\forall p$: $p \in u$, if p is a constant predicate, include $\{p\}$ in s_1. (ii) $\forall v$: $v \subseteq u$, v is a maximal set of dependent predicates in u, if $|v| >1$, then include $\{v\}$ in s_1.

In Step 2, $\forall v$: $v \in s_1$, if $|v| \equiv 1$, we evaluate the predicate in v, if it is not satisfiable, we define $w = v$ and include $\{w\}$ in a set s_2; if $|v| > 1$, we apply bMexpToVar transformation to the predicates in v. Then, we submit the resulting set of predicates to a constraint solver (for example Microsoft Z3) that can find the core set w of unsatisfiable predicates if these predicates are jointly unsatisfiable. If $|w|>1$, we include $\{w\}$ in s_2.

In Step 3, $\forall w$: $w \in s_2$, we generalize v into a pattern with 3 sub-steps: (i) $\forall p$: $p \in w$, we first transform p by expressing it as: *exp relop c*, where *relop* is a relational operator, c is a constant and *exp* is an expression that cannot be expressed as another expression added with a nonzero constant; (ii) next, we replace each real variable in the predicates in w by an ordered real variable R_i ($i \geq 1$), such that R_{i+1} is only used for replacement

when R_i has been used. Likewise, we replace each integer/Boolean variable in the predicates in w by integer/Boolean variables in the same manner; (iii) at last, if w is a set of predicates with real variables, for each nonzero numerical constant in the predicates in w, we replace it with a constant symbol selected from the sequence (….., d_3 , d_2 , d_1, 0, c_1 , c_2 , c_3,….) such that:

1) Positive and negative constants are replaced by c_i and d_j $(i, j \geq 1)$ respectively.
2) c_{i+1} is selected only if $_{ci}$ has been selected, likewise, d_{j+1} is only selected only if dj has been selected used.
3) Among the constants in the predicates in w, the symbols that replace the original constants preserve the order of the original constants.

If w is a set of predicates with only integer variables, we replace each real variable in the predicates in w by an ordered real variable I_i $(i \geq 1)$, such that I_{i+1} is only used for replacement when I_i has been used. We replace the constants in the predicates in w with constant symbols selected from the sequence (….., n_3 , n_2 , n_1, 0, m_1 , m_2 , m_3,….) that stand for ordered integer constants in the same manner. If there are two original integer constants with the difference of one, the smaller one is replaced by a constant symbol taken from the above sequence, say n_1, and the bigger one is expressed as n_1+1. This is to deal with the fact that there is no integer in between of two integer constants with the difference of one. The latter may affect the satisfiability of the predicates in w jointly.

After the above 3 steps, we have mapped an unsatisfiable path constraint to a pattern. Finally, we include the resulting pattern in $Set_{pattern}$ if it is not included yet.

IV. The Application

The patterns of unsatisfiable constraints mined from the proposed approach can be used to detect infeasible paths efficiently. One needs to prepare patterns in advance and store them in $Set_{pattern.}$ Then to identify whether a given path is infeasible, we perform the following steps:

1) Extract the path constraint, denoted by u.
2) Apply the Step 1 of the pattern mining process on u to have s_1={$v \mid u\Phi v$}
3) Skip the Step 2 of the pattern mining process.
4) Apply Step 3 of the pattern mining process to have a set of generalized predicates ss_1={$v' \mid u\Phi v'$}.
5) If $ss_1 \in Set_{pattern}$, it is a successful. Then u is concluded as unsatisfiable and the corresponding path is concluded as infeasible.

It can be seen from the above steps, the use of expensive constraint solving is avoided, hence, improving the scalability in detecting infeasible paths.

Detection of infeasible paths is required in many areas and tools. The proposed approach can be applied in most of them in which perfect precision is not essential. These include test cases generation to avoid the time and effort spending on analyzing infeasible paths, coverage analysis to compute the coverage for a test suite by excluding infeasible paths, and software security vulnerability detection to exclude the consideration of infeasible paths.

V. Evaluation

We conducted a case study on a set of Java programs to assess the performance of the proposed approach. We implemented a prototype system called *InfiPatternMinerJ* and randomly selected Java systems from Sourceforge [7] for this case study.

A. Implementation

The prototype system *InfiPatternMinerJ* has two sub-systems: (i) *Pattern mining*: mine patterns of unstaisfiable constraints using the proposed approach. (ii) *Infeasible path detectio*n: Detect infeasible paths through matching with existing patterns. The second subsystem is to validate the accuracy of applying the proposed approach to detect infeasible paths.

Pattern mining: the prototype system implements the proposed mining process and mines patterns from training systems. More specifically, given a training system there are two sub-steps: (a) Prepare training paths: For each root procedure – methods without any caller, an inter-procedural CFG will be generated. *InfiPatternMinerJ* then extracts paths from each control flow graph. Based on recent studies [8, 9], the basis path criteria [10] could generate a limited-sized set of paths to exhibit a large portion of correlations among predicates. Therefore, to mine patterns as many as possible, and at the same time, to minimize the potential path explosion risk, *InfiPatternMineJ* samples a basis path set over each inter-procedural CFG for pattern mining. (b) Mining: The basis inter-procedural paths are stored in a set Set_{path}. *InfiPatternMineJ* performs the mining process to find out all the patterns from the input paths. The output from *InfiPatternMineJ* is a set of patterns stored in a set $Set_{pattern}$. Additionally, to minimize the inaccuracy due to the existence of external dependency (e.g. third party library functions, native functions), *InfiPatternMineJ* provides an interface for users to document the semantics of these functions on the fly.

Infeasible path detection: the prototype system implements the application in Section IV that uses patterns to detect infeasible paths without using constraining solvers. *InfiPatternMinerJ* later assesses the detection accuracy by validating each detected infeasible paths against a constraint solver—Z3.

In this case study, seven Java programs from different application domains are randomly picked as independent test cases: Dagger-0.9 (a dependency injection framework), Jgraph-5.8.1 (a graph visualization component), Jsmooth-0.9 (a executable wrapper), JasperReports-5.5 (a reporting engine), Xerces-Java-1.4 (an XML parse), JCM-1.0 (a math library) and JHotDraw-7.6 (a GUI framework). The first five systems are used as the training set, that is, *InfiPatternMinerJ* mines patterns from those systems. The remaining two systems are used as the test set, that is, using the discovered patterns from the training systems, *InfiPatternMinerJ* detects infeasible paths in those two systems.

B. Experiment results

For the 5 systems used for mining patterns, after excluding empty functions, there are 3317 inter-procedural CFGs and 27866 inter-procedural basis paths in total. *InfiPatternMinerJ* applied the proposed mining procedure on the input paths and gradually found 19 patterns: 4 patterns are cases that the corresponding paths containing a singleton of unsatisfiable constant predicate; 15 patterns are cases that the corresponding paths contain a set of unsatisfiably joint predicates. These 19 patterns are displayed by the 2nd column and the first 19 rows in Table IV. An example of pattern mining is given by Figure 2. This piece of code is simplified from a function in JHotDraw. We take a path as an example, that is *path*=(*entry*, 1,2,3,4,5,6,*end*). After applying the mining process, a pattern of unsatisfiable constraints is found: { $p_1=(I_1 > c_2)$, $p2=(I_1 < c_1)$, $p_3=(c_2-c_1<1)$}.

For the rest 2 systems used for infeasible path detection, 5000 paths are randomly selected from each. *InfiPatternMinerJ* concludes each path's infeasibility through pattern matching against the existing 19 patterns. The identified infeasible paths are re-evaluated with a full symbolic evaluation (assisted with manual effort) to assess the detection accuracy.

We applied a set of common measurement for the assessment: true positive (*tp*), false positive(*fp*), false negative (*fn*), true negative (*tn*), probability of detection (*pd*), probability of false alarm (*pf*), and precision (*pr*) [9]. Table III shows the result of this assessment. For the two systems, *InfiPatternMinerJ* has no false positives generated, but causes 76 false negatives. The corresponding value of *pr*, *pf*, *pd* is 1, 0, 94.99% respectively. This result shows our approach has a high accuracy in practice. Table 5 further shows the distribution of each pattern over the two test systems. We also investigate the 76 false negative cases, and found that these cases would form two new patterns: (a) $\{p_1=(x_1=0), p_2=(x_1<0)\}$; (b) $\{p_1=(x_1=c_1)$, $\{p_2=(x_1\geq c_2)\}$.

Interestingly, we also found that the training systems and the testing systems are from different application a domain, which helps prove that the proposed approach is a general reusable solution to detect infeasible paths with high coverage.

Figure 3 shows an example that uses the extracted patterns to detect an infeasible path. The target path is path=(a1,a2,a3,b1,b3,b4,b5,b8), which is extracted from a function in JCM. Based on the procedure in Section IV, *InfiPatternMinerJ* generalizes a set of predicates from the path. By comparing this set of predicates with all the existing 19 patterns, a successful match is found. Therefore, *InfiPatternMinerJ* concludes the target path as infeasible. Use our approach to detect infeasible path does not require using constraint solvers at all. This helps enhance the detection efficiency when comparing with other common approaches. For the same, if we use the Marple [5] to test the infeasibility of the target path, it will timeout because its constraint solver cannot handle the function Math.log() well.

C. Threats to validity

Firstly, in the implementation, we did not interpret the semantics of Java string functions. Therefore, *InfiPatternMinerJ* is not able to fully support systems with many string functions currently. Meanwhile, *InfiPatternMinerJ* provides an interface for users to define behaviors for native functions. The user definition may bring in inaccuracy. Secondly, in the experiment, we do not consider the inaccuracy caused by potential floating-point rounding errors. Therefore, *InfiPatternMinerJ* currently also cannot precisely handle systems with many floating-point computations. Thirdly, all infeasible paths detected in this paper refer to cases from single-threaded programs. We have not extended *InfiPatternMinerJ* to support programs with many multi-threading behaviors.

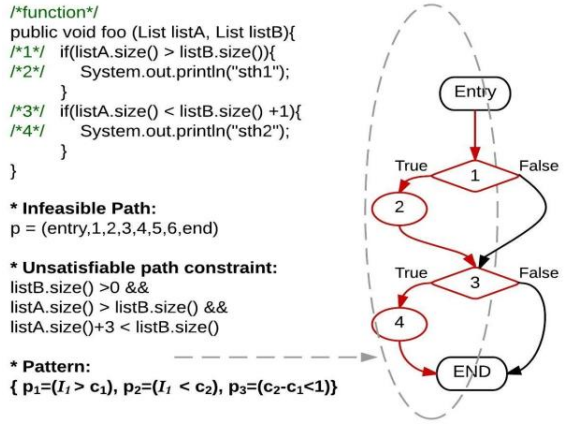

Fig. 2. An example of pattern mining

TABLE I. ACCURACY ASSESSMENT OF *INFIPATTERNMINERJ*

System	*tp*	*fp*	*fn*	*tn*	*Pd*	*Pf*	*Pr*
JCM	717	0	45	4238	94.10%	0	1
JHotdraw	814	0	31	4155	96.33%	0	1
total	1531	0	76	8393	95.27%	0	1

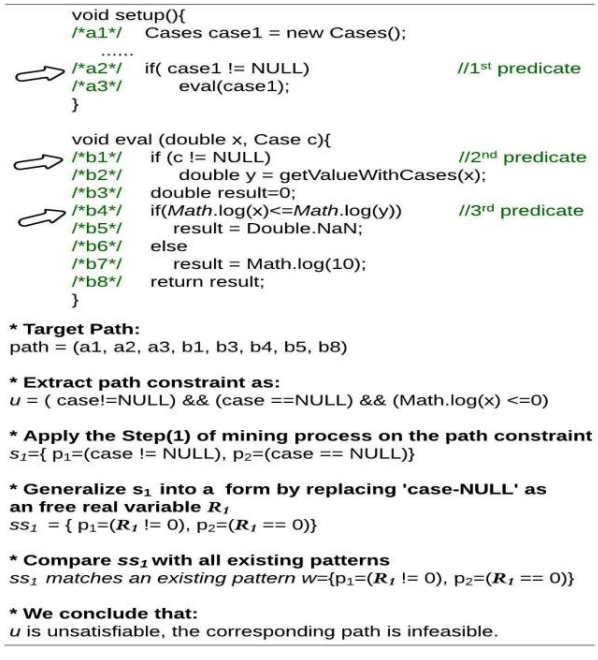

Fig. 3. An example of using patterns to detect infeasible path

TABLE II. DISTRIBUTION OF PATTERNS OVER THE 2 TEST SYSTEMS

Pattern ID	Patterns of unsatisfiable constraints	JCM	JHotDraw
1	$\{c_1 \neq c_2\}$	**150**	**211**
2	$\{c_1 > c_2\}$	**0**	**3**
3	{TRUE ≠ TRUE}	**0**	**0**
4	{FALSE =TRUE}	**51**	**47**
5	$\{R_1 \geq R_2\}$	**0**	**14**
6	$\{R_1 \geq c_1, R_1 < c_1\}$	**1**	**0**
7	$\{R_1 = c_1, R_1 > c_1\}$	**1**	**1**
8	$\{R_1 \geq c_1, R_1 < c_1\}$	**23**	**0**
9	$\{R_1 \geq c_1, R_1 \leq c_1, R_1 \neq 0\}$	**0**	**0**
10	$\{R_1 \geq c_2, R_1 \leq c_1\}$	**0**	**0**
11	$\{R_1 > c_1, R_1 < c_1\}$	**0**	**103**
12	$\{R_1 = c_1, R_1 \neq c_1\}$	**182**	**0**
13	$\{R_1 > c_2, R_1 \leq c_1\}$	**7**	**0**
14	$\{R_1 = c_2, R_1 \leq c_1\}$	**0**	**0**
15	$\{I_1 \geq c_2, I_1 \leq c_1\}$	**0**	**69**
16	$\{I_1 = c_2, I_1 = c_1\}$	**51**	**1**
17	$\{I_1 > c_1, I_1 < c_2, c_2 - c_1 < 1\}$	**4**	**175**
18	$\{I_1 = c_1, I_1 \neq c_1\}$	**247**	**190**
19	$\{R_1 = R_2, R_1 = R_3, R_2 \neq R_3\}$	**0**	**0**
20	**(new)** $\{R_1 = c_1, R_1 < c_1\}$	**0**	**31**
21	**(new)** $\{R_1 = c_1, R_1 > c_2\}$	**45**	**0**

VI. RELATED WORK

There are several ways to detect infeasible paths. But the most two common approaches are data flow analysis and constraint propagation. The first common approach applies data flow analysis [11]. These approaches are often useful for finding a wide variety of infeasible paths. However, due its path-insensitiveness, these approaches do not have high accuracy. The second common type of approaches is based on constraint propagation. This type of approaches usually detects infeasible paths through constraint propagation along paths via symbolic evaluation [1, 3, 4]. These approaches often extract and propagate the constraint that an input must satisfy for exercising a path as a symbolic expression. The path is infeasible if the constraint is unsatisfiable. This typically requires the help of constraint solvers. However, in general, it is intractable to solve a constraint. The advantage of such an approach is its higher precision. However, due to the large search space in both path exploration and constraint solving, it tends not to scale to large systems. This type of approaches is commonly used to generate path-based test cases [3, 4].

Other than above, one could also use light-weighted heuristics to identify infeasible paths among correlated branches [6, 8, 9]. However the heuristics or patterns in these works are all fixed. It easily yields many false-positive cases when applying these approaches on complex real systems.

The proposed approach in this paper has a different objective. It proposes an approach for mining patterns of infeasible path constraints from code such that these patterns can be used to detect infeasible paths without calling constraint solvers. Comparing with existing works, the novelties of the proposed approach are: (1) Applying the proposed patterns, infeasible paths are detected neither using constraint solvers nor evaluating function calls totally. Hence, detecting infeasible paths is scalable; (2) Existing heuristics and patterns used to detect infeasible paths are pre-defined through observation. This paper proposes a systematic approach for mining patterns of infeasible path constraints dynamically that has not been explored before; (3) the patterns for infeasible path constraint can vary but they can be collected gradually, and as the more patterns are collected, the accuracy is improved. Further, patterns can be collected from code developed using different programming language, executable code and different lifecycle maturity or different intention (e.g., malware, obfuscated code).

VII. CONCLUSION

In this paper, we propose a novel approach that mines patterns of infeasible paths. Each pattern is either a singleton of an unsatisfiable predicate or a set of unsatisfiably joint predicates. Based on the given paths, the proposed approach is able to find patterns gradually and automatically. The mined patterns can be used to detect infeasible paths without using any constraint solvers. Since the execution of constraint solvers is computationally expensive, the proposed mining procedure improves the scalability of infeasible paths detection and related applications. A case study is carried out to evaluate the proposed approach. With 5 training systems, the proposed approach found 19 patterns of infeasible paths. When using these patterns to detect infeasible paths on another two systems, the experiment results prove the accuracy and scalability of the proposed approach.

REFERENCE

[1] R. Bodic, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," *SIGSOFT Softw. Eng. Notes*, vol. 22, pp. 361-377, 1997.

[2] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, pp. 366-427, 1997.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263-272, 2005.

[4] N. Tillmann and J. D. Halleux, "Pex: white box test generation for .NET," presented at the Proceedings of the 2nd international conference on Tests and proofs, Prato, Italy, 2008.

[5] W. Le and M. L. Soffa, "Marple: Detecting faults in path segments using automatically generated analyses," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, p. 18, 2013.

[6] S. Ding and H. B. K. Tan, "Detection of Infeasible Paths: Approaches and Challenges," in *Evaluation of Novel Approaches to Software Engineering*, ed: Springer, 2013, pp. 64-78.

[7] (2015). *Sourceforge*. Available: http://sourceforge.net/

[8] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Inf. Softw. Technol.*, vol. 50, pp. 641-655, 2008.

[9] S. Ding, H. Zhang, and H. Beng Kuan Tan, "Detecting infeasible branches based on code patterns," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, 2014, pp. 74-83.

[10] J. Poole. (1995, A Method to Determine a Basis Set of Paths to Perform Program Testing. Available: http://hissa.nist.gov/publications/nistir5737/

[11] J. Gustafsson, "Eliminating annotations by automatic flow analysis of real-time programs," presented at the Proceedings of the Seventh International Conference on Real-Time Systems and Applications, 2000.