A Survey of Infeasible Path Detection

Sun Ding, Hee Beng Kuan Tan and Kai Ping Liu
Division of Information Engineering, Block S2, School of Electrical & Electronic Engineering,
Nanyang Technological University, Nanyang Avenue, 639798, Singapore, Singapore

Keywords: Survey, Path Infeasibility, Symbolic Evaluation, Program Analysis, Software Testing.

Abstract: A program has many and usually an infinite number of logic paths from its entry point to its exit point. Each execution of the program follows one of its logic paths. Regardless of the quality of the program and the programming language used to develop it, in general, a sizable number of these paths are infeasible — that is no input can exercise them. Detection of these infeasible paths has a key impact in many software engineering activities including code optimization, testing and even software security. This article reviews methods for detecting infeasible paths and proposes to revisit this important problem by considering also empirical aspect in conjunction to program analysis.

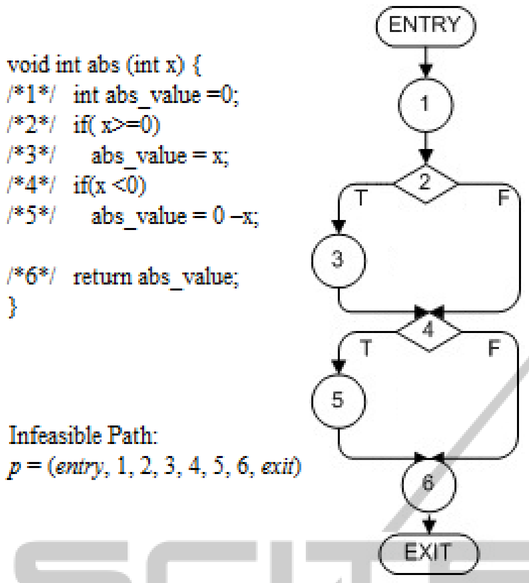
1 INTRODUCTION

Control flow graph (CFG) is the standard model to represent the execution flow between statements in a program. In the CFG of a program, each statement is represented by a node and each execution flow from one node to another is represented by a directed edge, where this edge is out-edge of the former node and the in-edge of the latter node. Each path through the CFG from the entry node to the exit node is a logic path in the program. In order for an execution to follow a path in the CFG, the input submitted to the program must satisfy the constraint imposed by all the branches that the path follows. An **infeasible path** is a path in the CFG of a program that cannot be exercised by any input values. Figure 1 shows an infeasible path *p = (entry, 1, 2, 3, 4, 5, 6, exit)* in a CFG. This is because we cannot find any input x satisfying $x \geq 0$ and $x < 0$ jointly.

The existence of infeasible paths has major impact to many software engineering activities. Code can certainly be optimized further if more infeasible paths can be detected during the process of optimization. In software testing, the structural test coverage can be much accurately computed if infeasible paths can be detected more accurately. In the automated generation of structured test cases, much time can be saved if more infeasible paths can be detected. In code protection, it can also help in code deobfuscation to identify spurious paths inserted during obfuscation. In software verification, detecting and eliminating infeasible paths will help to enhance the verification precision and speed. There are many more areas like security analysis (Padmanabhuni and Tan, 2011), web application verification (Liu and Tan, 2008, 2009), database application design (Ngo and Tan, 2008) that can be helped by the detection of infeasible paths.

To detect infeasible paths in real programs, one needs to deal with complex data structures and dependency. Additional effort is required to formally present them in symbolic expressions or constraints for further verification by heuristics, predefined rules or even standard theorem provers. If the verification returns negative results (e.g.: "Invalid" answer from theorem provers), the path is then considered as infeasible. Such verification model is undecidable in general. But it is still possible to have practical approaches that are not theoretically complete to detect infeasible paths.

The purpose of this article is to familiarize the reader with the recent advances in infeasible paths detections and its related applications. Concepts and approaches will be introduced informally, with citations to original papers for those readers who preferring more details. Information about tools and implementation is also introduced. The paper is organized as below: the literals for infeasible paths detection is reviewed in section2. Information of tool implementation is introduced in section 3. We discussed remaining problems and future challenges in section4. Section 5 summarizes the entire paper.



```
void int abs (int x) {
/*1*/  int abs_value =0;
/*2*/  if( x>=0)
/*3*/     abs_value = x;
/*4*/  if(x <0)
/*5*/     abs_value = 0 –x;

/*6*/  return abs_value;
}
```

Infeasible Path:
p = (*entry*, 1, 2, 3, 4, 5, 6, *exit*)

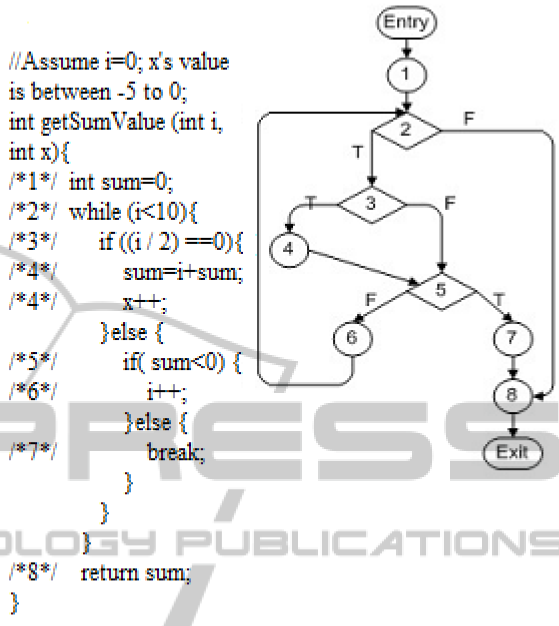
Figure 1: An infeasible path.

2 DETECTION OF INFEASIBLE PATH

During software developing or testing, infeasible path detection usually appears as an essential step for the final project goal. A variety of methods have been proposed. Based on the ways that they detect infeasible paths, we classify them into six types: (1) data flow analysis; (2) path-based constraint propagation; (3) property sensitive data flow analysis; (4) syntax-based approach; (5) infeasibility estimation; (6) generalization of infeasible paths. These methods differ in their detection precision, computational cost and relevantly suitable applications. We review these methods by introducing their main features, strength and weaknesses and the related applications.

2.1 Data Flow Analysis

Classic Data flow analysis is a technique over CFG to calculate and gather a list of mappings, which maps program variables to values at required locations in CFG. Such list of mappings is called flow fact. A node with multiple in-edges is defined as a merge location, where flow facts from all of its predecessor nodes are joined together. Due to the joining operation, a variable may be mapped to a set of values instead of a single value. If a node has multiple out-edges (predicate node), each of these out-edges is defined as a control location. Flow fact is split and traversed to the successor nodes at control locations. Due to the splitting operation, a variable may be mapped to an empty set of values (Khedker et al., 2009).



```
//Assume i=0; x's value
is between -5 to 0;
int getSumValue (int i,
int x){
/*1*/  int sum=0;
/*2*/  while (i<10){
/*3*/     if ((i / 2) ==0){
/*4*/        sum=i+sum;
/*4*/        x++;
          }else {
/*5*/        if( sum<0) {
/*6*/           i++;
             }else {
/*7*/           break;
             }
          }
       }
/*8*/  return sum;
}
```

Figure 2: Infeasible path detection with data flow analysis.

Data flow analysis is a common approach for detecting infeasible paths. In this type of approach, each control location would be checked when they are traversed. An infeasible path is detected when any variable is mapped to an empty set of values at a control location. In Figure 2, suppose we only consider the flow fact about variable *sum*. Here *sum* is an integer variable initialized as 0. Therefore the flow fact is initialized as *[0, 0]* after *node1*. The flow fact is traversed transparently through *node2*, *node3* and reaching *node5*, after which it is split as two: one as *[0, 0]* flowing to *node6* and the other as an empty set flowing to *node7*. It is then concluded that any path passing through (1, 2, 3, 5, 7) is infeasible. Approaches based on data flow analysis are often useful for finding a wide variety of infeasible paths. In the above example, the checking at *node5* can detect a family of infeasible paths, which all containing the sub part (1, 2, 3, 5, 7).

However, classic data flow analysis scarifies the detection precision, which causes some infeasible paths wrongly identified as feasible. It is important to note that the flow fact computed at a control location *L* is essentially an invariant property — a property that holds for every visit to *L*. Therefore two things will cause the loss of the detection precision: First, the correlated branches are ignored and flow facts are propagated across infeasible paths. Second, by the joining operation at merging location, flow facts from different paths are joined, leading to further over-approximation (Fischer et al., 2005). To explain this point, consider the example program shown in Figure 2. The flow fact of variable *i* is initialized as *[0, 0]*. After passing *node3*, it is split as two: one as *[0, 0]* on the *TRUE* branch and the other as an empty set on the *FALSE* branch. However by simply keeping track of all possible variable values at *node5*, the two different flow facts are joined. The flow fact from *node5* flowing to *node6* is over-approximated as *[0, 0]*. Hence we cannot directly infer that *node4* cannot be executed in consecutive iterations of the loop. Therefore path such as (*entry*, 1, 2, 3, 4, 5, 6, 2, 3, 4) cannot be inferred as infeasible, which actually is. The most typical and well cited method for detecting infeasible paths based on data flow analysis is from Gustafsson et al. (2000, 2002, and 2006). Approaches based on data flow analysis are path insensitive.

Other similar methods include work from Altenbernd (1996), detecting infeasible paths by searching for predicates with conflict value range while traversing CFG in an up-bottom style. This method depends on knowing execution time of each basic block in advance. The basic block refers to a piece of continuing nodes except predicate nodes. At each merge location, only the in-edge with longest execution time will be remained for further consideration. All flow facts will be checked at each control location. Those branches that with variables mapped to an empty value set will be detected as infeasible and excluded for further consideration. Dwyer et al. (2004) proposed to adopt data flow analysis approach to check consistency violation in concurrent systems. They construct a Trace-Flow Graph (TFG), which is a modified version of CFG for concurrent systems. Variables in TFG are mapping to a set of possible system properties like sequence of event calling, synchronization of critical section. A consistency violation is found when a corresponding path is identified as infeasible in the TFG.

Approaches based on data flow analysis do not require providing a prepared set of paths. It searches for infeasible paths directly based on CFG. So they are often applied to estimate the maximum execution time for a procedure, called WCET: worst-case execution time (Ermedahl, 2003) which is essential in designing real time systems. Firstly they help tighten the estimated result of WCET analysis by removing the influences from infeasible paths in the case that these paths are the longest ones. Secondly, they are useful in deriving loop upper bound in WCET analysis.

2.2 Path-based Constraint Propagation

Path-based propagation approaches apply symbolic evaluation to a path to determine its feasibility. These methods carry a path sensitive analysis for each individual path in a given path set by comparing with approaches based on data flow analysis. Through symbolic evaluation, they propagate the constraint that a path must satisfy and apply theorem prover to determine the solvability of the constraint. If the constraint is unsolvable, the path is then concluded as infeasible. These methods have high precision of detection but with heavy overhead. They are usually applied in code optimization and test case generation in which accuracy is essential. Figure 3 gives a general overview of these methods.

In propagating constraint along a given path, either backward propagation (Balakrishnan, 2008) or forward propagation strategy (Ball and Rajamani, 2002) could be adopted to extract the path constraint under the supported data types. Forward propagation traverses the path from entry to exit and performs symbolic execution on every executable statement. Intermediate symbolic values are stored for subsequent use. It can detect infeasible paths early by detecting contradicting constraints early. It is also more straight forward and thus easier to implement, especially in the case of dealing with arrays or containers like List and HashMap. (Tahbildar and Kalita, 2011). However the storage of intermediate values may grow very fast and cause this strategy not scalable for large program. Backward strategy applies a bottom-up traversing manner by collect path constraint first and later only search for values correlated with path constraint. The space of the intermediate storage is largely reduced.

Based on the underlying domain theories of the constraint solver or theorem prover (Robinson and Voronkov, 2001), constraint solving determines the solvability of the propagated constraint. The power and precision of path-based constraint propagation methods depend on the power of constraint solver. Hence, they are sound except on those cases in which existing constraint solvers have problems (e.g., floating point problems).

As mentioned by Williams (2010) recently, though constraint resolution is very efficient most of the time, it is actually NP-complete and it is undecidable to know which kind of constraint will

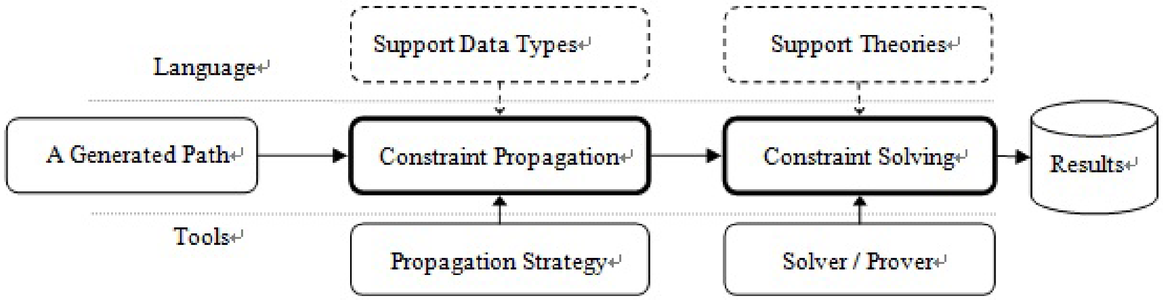
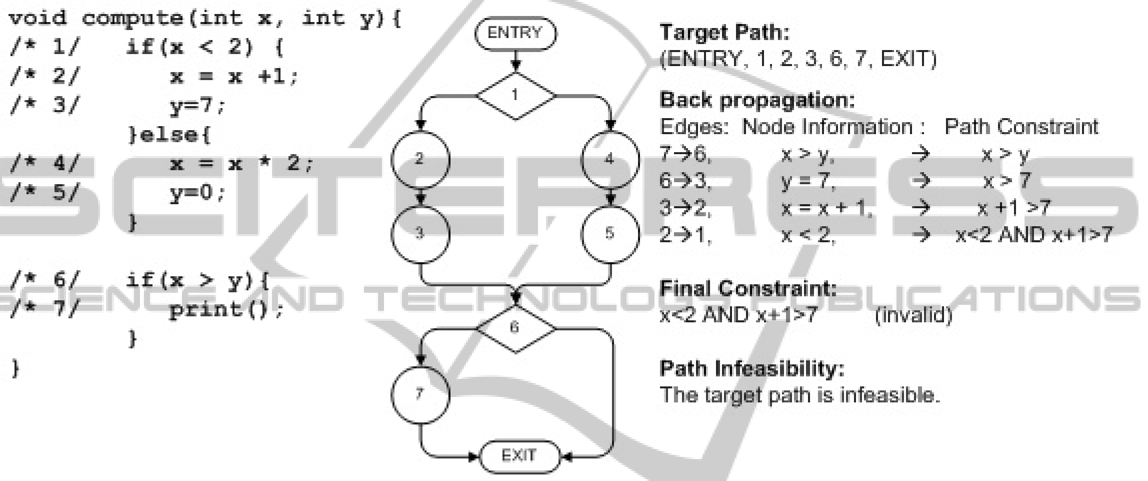


Figure 3: An overview of path-based constraint propagation method.


```
void compute(int x, int y){
/* 1/   if(x < 2) {
/* 2/       x = x +1;
/* 3/       y=7;
        }else{
/* 4/       x = x * 2;
/* 5/       y=0;
        }

/* 6/   if(x > y){
/* 7/       print();
        }
}
```

Target Path:
(ENTRY, 1, 2, 3, 6, 7, EXIT)

Back propagation:

Edges:	Node Information :		Path Constraint
7→6,	x > y,	→	x > y
6→3,	y = 7,	→	x > 7
3→2,	x = x + 1,	→	x +1 >7
2→1,	x < 2,	→	x<2 AND x+1>7

Final Constraint:
x<2 AND x+1>7 (invalid)

Path Infeasibility:
The target path is infeasible.

Figure 4: An illustration of the path-based constraint propagation method.

take "too long" to be solved. Therefore it is not always possible for these methods to determine the feasibility of a path automatically. Furthermore every time execution the constraint solver, there is a risk of causing the timeout exception.

Figure 4 illustrates an example of path-based constraint propagation methods in general. Consider the target path p = (*entry*, 1, 2, 3, 6, 7, *exits*). After propagation along the path using backward propagation strategy, the constraint finally becomes ((x<2) AND (x+1)>7). The resulting constraint is submitted to a constraint solver for evaluation. As the result is unsolvable, therefore, this path is identified as infeasible.

Among the path-based constraint propagation methods, Bodik et al. (1997) observed that many infeasible paths were caused by branch correlation and data flow analysis based approaches are overly conservative to handle them. Starting from a predicate node, they address this problem by backward propagation along a path to accumulate path constraint and consecutively evaluating the constraint with predefined rules. The path traversed is identified as infeasible if the associated constraint is determined by the predefined rules as unsolvable. If the constraint is determined by the predefined rules as solvable, the path is identified as feasible. If the solvability of the constraint cannot be determined by the predefined rules, the feasibility of the path is therefore undetermined. Goldbeg et al. (1994) approached the problem with proposed a more general model. For each targeting path, its infeasibility is determined by the corresponding path constraint. The path constraint is the conjunction of each branch condition along the path after substituting every referenced variable's definition. An independent constraint solver named KITP is invoked to evaluate the path constraint. If the path constraint is evaluated as unsolvable, then the path is identified as infeasible. Goldbeg et al. explicitly specified a domain, on which that constraint solver KITP could work. The domain limits the considered data types as Integer and Real and limits the constraint type as linear.

By equipping different constraint solvers, other approaches are able to detect infeasible paths with constraints over other domains. Ball and Rajamani (2001) used a binary decision diagram (BDD) as the prover to identify infeasible paths for Boolean programs. Zhang and Wang (2001) used an ad-hoc SMT solver to detect infeasible paths with path constraints over both Boolean and numerical linear domains. Bjørner et al. (2009) detected infeasible paths over the domain of String by using a SMT solver called Z3 (2008) which is able to manipulate and process those string functions in path constraints.

The advantage of the path-based constraint propagation approaches is the precision with which we can detect infeasible program paths. The difficulty in using full-fledged path-sensitive approaches to detect infeasible paths is the huge number of program paths to consider. In summary, even though path-based constraint propagation methods are more accurate for infeasible path detection, they suffer from a huge complexity.

Constraint propagation methods are often used in areas requiring high accuracy like test case generation and code optimization. In test case generation, paths will be firstly evaluated their infeasibility. Infeasible paths will be filtered away from test data generation to save resources and time. For example, Korel (1996) checked the path infeasibility before generating test case at the last predicate to avoid unnecessary computation. Other similar examples are like work from Botella et al. (2009), and work from Prather and Myers (1987). In code optimization, def-use pairs along those infeasible paths are eliminated for enhancing the efficiency of code (Bodik, 1997).

2.3 Property Sensitive Data Flow Analysis Approach

Both data flow analysis and constraint propagation approach have strength and weakness. This section introduces the hybrid approach that combine both of them together under the framework of partial verification. The latter refers to the verification against a list of given properties to check instead of verifying all system properties. Property is an abstract term covering variables, functions or special data structures like pointer in C/C++.

With a given list of properties, methods of this type have similar routine with approaches using classic data flow analysis except two modifications. First the flow fact is updated at location *L* only when *L* contains properties correlated operations. Second at merge locations, equal values for the same property from different flow facts will be merged as one; but different values for the same property from different flow facts will be separately recorded instead of joining them together. Same with approaches using classic data flow analysis, infeasible paths would be detected if any property is mapped to an empty value set at a control location. To illustrate this, let us go back to the example in Figure 2. Suppose variable *sum* and *i* are specified as the two properties. At the *TRUE* branch of *node5*, *sum* is mapped to an empty set. A family of infeasible paths containing (1, 2, 3, 5, 7) are detected as efficient as using classic data flow analysis. However at *node5*, the flow facts for *i* will be recorded separately: f_{node3} and f_{node4} .Therefore, this time, we are able to detect infeasible paths such as (*entry*, 1, 2, 3, 4, 5, 6, 2, 3, 4) because only f_{node4} will be considered in the consecutive iterations of the loop.

Among this type of approaches, one well cited work is from Das et al. (2002). They proposed a method called ESP, whose main idea is introduced in the last paragraph, to enhance the precision of data flow analysis while also guaranteeing the verification could be controlled in polynomial time. They later extended ESP to a more abstract and general model. Work from Dor et al. (2004) extended ESP for better performance over C/C++, especially for better cooperating with pointer and heap. Other similar work may include work from Fischer et al. (2005) and Cobleigh et al. (2001).

The advantage here is that this type of methods achieves a good balance between precision and scalability. However, it brings difficulty in specifying properties accurately. There is also a risk of detection failure because of losing a precise tracking of some properties, which having complex data structures.

2.4 Syntax-based Approach

Many infeasible paths are caused by the conflicts that can be identified from using solely syntax analysis. Syntax-based approaches take advantage from these characteristics. They define syntax for such conflicts as patterns or rules. Syntax analysis is applied to detect infeasible paths through using rules or recognizing patterns.

The more noticeable recent method is proposed by Ngo and Tan (2007, 2008). They identified the four syntactic patterns, identical/complement-decision pattern, mutually-exclusive-decision pattern, check-then-do pattern, looping-by-flag pattern. These patterns model the obvious conflicts between the value of a variable set and the value of the same variable asserted by the predicate of a branch or between predicates of branches, in a path.

For example, the predicates at branches (2, 3) and (4, 5) in Figure 1 are $x \geq 0$ and $x < 0$ respectively. These two predicates have obvious conflict and can be detected from syntax analysis. Hence, the path p = (*entry*, 1, 2, 3, 4, 5, 6, *exit*) in Figure 1 is clearly infeasible. Based on the four patterns identified, they developed a method to detect infeasible paths from any given set of paths. Through the use of these patterns, the method can avoid the expensive symbolic evaluation by just using solely syntax analysis to detect infeasible paths through recognizing these patterns. In opposing to other methods, their methods were proposed as an independent method. They have also conducted an experiment on code from open-source systems and found their method can detect a large proportion of infeasible paths.

Among those earlier syntax-based approaches, one well cited work is from Hedley and Hennell (1985). They proposed to use heuristics rules to detect four types of infeasible paths: infeasible paths caused by consecutive conditions, infeasible paths caused by inconsistency between test and definition, infeasible paths caused by constant loop control variable, and infeasible paths caused by constant loop times. These rules are quite efficient as they solely based on syntax analysis. Later experiment from Vergilio et al. (1996) showed that by a fast scan, Hedley and Hennell's rules were able to correctly identify nearly half paths as infeasible in a path set with 880 sample paths.

The advantage of syntax-based approaches is that they can avoid the expensive symbolic evaluation by applying solely syntax analysis to achieve some efficiency. However, these methods may report a small number of infeasible paths that are actually feasible as they just based on syntax analysis alone. That is, they suffer from the possibility of reporting false-positive results.

Syntax-based approaches detect infeasible paths from a set of paths, so they rely on well-constructed paths set. They are often used for a fast scan during the early testing stage. They are also used as the first step for code optimization or coverage estimation to avoid the influence of infeasible paths during the later analysis.

2.5 Infeasibility Estimation Approach

Early researchers have found the problems caused by infeasible paths and it was very hard to achieve a satisfied detecting result. Therefore they managed to build statistical metrics to estimate the number of infeasible paths in a given procedure based on certain static code attributes. The most famous work is from Malevris et al. (1990). They stated that "the number of predicates involved in a path being a good heuristic for assessing a path's feasibility". The greater the number of predicates exist in a path, the greater the probability for the path being infeasible. They further concluded a regression function $f_q = Ke^{-\lambda q}$ to represent the above relationship, in which K and λ are two constants, q stands for the number of predicate nodes involved in a given path, while f_q stands for the possibility of this path being feasible. Later, Vergilio et al. (1996) validated the above results over a broader selection of programs and extended the work to involve in more static code attributes, such as: number of nodes, number of variables and number of definitions.

The advantage of such metrics is that they are easy to implement and provide a fast way to predict path infeasibility within a confidence level (Vergilio et al., 1996). However, it is a method of rough estimation rather than accurate detection. The accuracy of the regression function also biased over different test programs and different programming language.

2.6 Generalization of Infeasible Paths Approach

When a path is infeasible in a CFG, all other paths that contain the path are also clearly infeasible. Based on this simple concept, Delahaye proposed to generate all infeasible paths from a given set of seed infeasible paths (Delahaye et al, 2010). It provides a convenient way to generate error seeded models for further testing, especially for legacy programs or combined as a component in regression testing.

3 TOOLS IMPLEMENTATION

In most program optimization, software analysis and testing tools, infeasible path detection usually appears as an important component. Best to our knowledge, there is no independent tool particularly designed for it. In this section, we introduce related existing tools based on above approaches. We also brief the implementation details of the methods described in last section to help those who want to detect infeasible paths in their own applications.

We select 14 relevant tools and analysze them in this section. These tools are summarized in Table 1.

Table 1: Useful Tools for Implementation.

Name	Description	Link	Supported Language
Code analysis and optimization			
Soot	A Java Optimization Framework.	http://www.sable.mcgill.ca/soot/	Java
CodeSufer	Code analyser	www.grammatech.com/products/codesurfer/	C/C++
Pixy	Code and security analyser for PHP	http://pixybox.seclab.tuwien.ac.at/pixy/	PHP
Automated Test Case Generation			
CUTE	Test case generation for C/C++	http://cute-test.com/wiki/cute/	C/C++
jCute	Test case generation for JAVA	http://cute-test.com/wiki/jcute/	JAVA
CREST	Test case generation for C	http://code.google.com/p/crest/	C
Pex	Structural testing framework for .Net	http://research.microsoft.com/en-us/projects/pex/	C#, C++, VB.net
eToc	Path selection with genetic algorithm and test case generation for C/C++, Java	http://www.informatik.hu-berlin.de/forschung/gebiete/sam/Forschung%20und%20Projekte/aktuelle-forschung-und-projekte/softwarekomponenten-entwicklung/eodl-projects/etoc/etoc	Java, C/C++
Theorem Prover			
Z3	SMT prover	http://research.microsoft.com/en-us/um/redmond/projects/z3/	C/C++
Lp_Solver	Linear constraint solver	http://lpsolve.sourceforge.net/5.5/	C/C++, JAVA, PHP, Matlab
BLAST	Lazy abstraction software verification	http://mtc.epfl.ch/software-tools/blast/index-epfl.php	C
Verification and Error detection			
ESC-Java	Error checking for annotated Java program	http://secure.ucd.ie/products/opensource/ESCJava2/	Java
SLAM	Verify critical properties for C/C++	http://research.microsoft.com/en-us/projects/slam/	C/C++
LCLint	Static analysis for C/C++	http://www.splint.org/guide/sec1.html	C/C++

3.1 Data Flow Analysis Approach

The most completed work is from Gustafsson et al. (2000, 2002, and 2006). In order to detect infeasible paths over complex programs, they decompose a program into several linked scopes. The latter is a set of program statements within a call of one procedure or an execution of a loop iteration. It is statically created so that calls to a function or a loop at different call sites will be marked and analyzed separately. This brings in higher precision but more expensive computation cost. The scope graph is hierarchical representation of the structure of a program which is used to describe the interaction relationships between scopes. Data flow analysis based on abstract interpretation will be performed over each scope separately to compute the live variables set. A recorder is created to store the infeasible paths detected within each scope. Among the work of Gustafsson et al., only primary data types are mentioned. There has been no corresponding open source toolkit published. For readers planning to code based on this type of approach, they could utilize existing data flow analysis framework to find out variables mapped to an empty value set at certain control locations and detect infeasible paths accordingly. For example, the sub package Spark in Soot can perform intra or inter data flow analysis over Java procedures. Other available toolkits are, for example, CodeSufer for C/C++, Pixy for PHP.

3.2 Path-based Constraint Propagation Approach

Approaches based on constraint propagation and solving often appear in tools of automated test case generation. The typical example is concolic testing (Sen et al., 2005). Before test data is generated, the target path will be tested its infeasibility by submitting the path constraint to theorem prover. If it is infeasible, the last predicate condition will be reversed to stand for a new path containing the opposite branch. The details could be found in the following tools: CUTE and jCUTE which are available as binaries under a research-use only license by Urbana-Champaign for C and JAVA;

CREST, which is an open-source solution for C comparable to CUTE; Microsoft Pex, which is publicly available as a Microsoft Visual Studio 2010 Power Tool for the NET Framework.

Readers, who are interested in implementing this type of approaches in their own application, can first apply those data flow analysis tools to propagate along a path and generate path constraint in symbolic expressions. Later the constraint could be submitted to theorem provers. Available tools of the latter include Z3, which is a SMT solver from Microsoft; Lp_Solver, which is an open source linear constraint solver under GNU license; BLAST, which is a prover often used to verify abstract program.

3.3 Property Sensitive Data Flow Analysis Approach

Das et al. (2002) proposed a tool called ESP, which is a research project for partial verification under Microsoft. ESP is able to construct CFG and perform property sensitive analysis in either intra-procedure or inter-procedure mode. The verification for given properties at control location is handled by its build-in rules for primary data types in C/C++. But the tool provides interface to replace the build-in rules with standard theorem provers for more complex analysis. There are also several other tools based on this type of approaches, which are like ESC-Java, SLAM, LCLint.

3.4 Syntax-based Approach

Syntax-based approach is easy to implement because the heuristics are concluded from code and expressed in a straight forward style for implementation. Ngo and Tan (2007) implemented an automated test case generation system called jTGEN for automated test data generation for Java. The system consists of an infeasible path detector that based on heuristic code-patterns, a code parser based on Soot, a path selector and a test case generator based on eToc. The system uses a genetic algorithm to select a set of paths. These paths are checked against heuristic code patterns and only feasible paths will be remained for test case generation.

4 LIMITATION OF CURRENT SOLUTIONS

Software grows fast in both size and complexity in current trend, more paths and constraints are encountered in the detection of infeasible paths, therefore using traditional symbolic evaluation based approaches for infeasible path detection encounter scalability issue. For methods of path sensitive analysis, there is a need to limit the number of the targeting paths. The simplest way is to set an upper bound to limit the paths number. Possible effort could be applying intelligent method like genetic algorithm to guide the path selection (Xie et al., 2009): by choosing proper fitness function, only paths with high suspicion of infeasibility would be remained for further processing. Another attempt is from Forgács and Bertolino (1997) who utilized program slicing: By reducing a program to a slice of the variables and statement concerned, the detection of infeasible paths is therefore made simpler.

Theoretically, it is believed that program complexity will highly raise the difficulty of detecting infeasible path. Because the path may contains long data dependency, complex data types, side-effect functions, and non-linear operators. It will be with high cost to develop a general model to cover them. It is also not possible to determine the infeasibility of all cases. As infeasible path detection could be viewed as a type of model abstraction and verification. Snifakis recently suggested (Edmund et al. 2009) that general verification theory would be of theoretical interest only. By contrast, a compositional study for particular classes of properties or systems would be highly attractable.

5 POTENTIAL PRATICAL SOLUTION

Detection of infeasible paths remains an important problem in software engineering. Current methods are still far to serve this important need effectively. Most of the current methods do not put much emphasis on the characteristics of infeasible paths in real system code. We propose a revisiting of this problem by examining, identifying and taking advantages of these characteristics as much as possible.

Theoretically, constraints imposed by branches that a path follows can be in any form. Therefore, it is unsolvable to determine the infeasibility of a path in general. However, theoretical limitation does not always imply practical limitation. Despite the theoretical limitation, one might still develop a good practical solution if there are useful practical characteristics one can take advantage.

More specifically, we propose to investigate the characteristics of constraints imposed by the branches that may lead a path in real system code to be infeasible. If one can empirically identify interesting characteristics that majority of these constraints are possessing, one might be able to take advantage from them to establish good practical methods to improve on both the precision and the proportion of infeasible paths that can be detected by current methods. Clearly, systematic experiment instrumented with both automated and manual evaluation to examine the proportion of infeasible paths that a method can detect is very difficult. However, researchers should still consider spending effort on this to provide the lacking empirical quantitative information on the proportion of infeasible paths that a method can detect.

If these practical methods can be invented and implemented to detect majority of the infeasible paths, it will provide major benefit to many related important applications such as code optimization, structural testing and coverage analysis.

6 CONCLUSIONS

We have reviewed existing methods for the detection of infeasible paths. We have also discussed the strengths and limitations of current methods. Noticeably, all the existing methods cannot detect majority of the infeasible paths efficiently. Most of the existing methods were proposed under other approaches to solve another problem such as code optimization or test case generation, in which the detection of infeasible paths has great impact.

REFERENCES

Altenbernd, P., 1996. On the False Path Problem in Hard Real-Time Programs. In *Real-Time Systems, Euromicro Conference*, pp. 0102-0102.

Balakrishnan, G., Sankaranarayanan, S., Ivančić, F., Wei, O. and A. Gupta, 2008. SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement. In *Static Analysis*, vol. 5079, pp. 238-254.

Ball, T. and Rajamani, S. K., 2002. The SLAM project: debugging system software via static analysis. *SIGPLAN Not.,* vol. 37, pp. 1-3.

Ball, T., and Rajamani, S. K., 2001. Bebop: a path-sensitive interprocedural dataflow engine. Presented at the *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Snowbird, Utah, United States.

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

Bjørner, N., Tillmann, N. and Voronkov, A., 2009. Path Feasibility Analysis for String-Manipulating Programs. Presented at the *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software*, York, UK.

Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M. and Williams, N., 2009. Automating structural testing of C programs: Experience with PathCrawler, in *Automation of Software Test*, *ICSE Workshop*, pp. 70-78.

Cobleigh, J. M., Clarke, L. A. and Ostenveil, L. J., 2001. The right algorithm at the right time: comparing data flow analysis algorithms for finite state verification. Presented at the *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada.

Concolic testing. Retrieved from: http://en.wikipedia.org/wiki/Concolic_testing

Edmund, M., Clarke, E., Allen, E. and Joseph, S., 2009. Model Checking: Algorithmic Verification and Debugging. In *Communications of the ACM*, Vol. 52, pp. 74-84.

Das, M., Lerner S., Seigle, M., 2002. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Not*., vol. 37, pp. 57-68.

Delahaye, M., Botella, B. and Gotlieb, A., 2010. Explanation-Based Generalization of Infeasible Path. Presented at the *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*.

Dor, N., Adams, S., Das M. and Yang. Z., 2004. Software validation via scalable path-sensitive value flow analysis. *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 12-22.

Dwyer, M. B., Clarke, L. A., Cobleigh, J. M. and Naumovich, G., 2004. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol*., vol. 13, pp. 359-430.

Ermedahl, A., June, 2003. A modular tool architecture for worst-Case execution time Analysis. *PHD thesis*, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden.

Fischer, J., Jhala, R. and Majumdar, R., 2005. Joining dataflow with predicates. *SIGSOFT Softw. Eng. Notes* 30, vol. 5, pp. 227-236.

Forgács, I. and Bertolino, A., 1997. Feasible test path selection by principal slicing. *SIGSOFT Softw. Eng. Notes*, vol. 22, pp. 378-394.

Goldberg, A., Wang, T. C. and Zimmerman, D., 1994. Applications of feasible path analysis to program testing, presented at the *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, Seattle, Washington, United States.

Gustafsson, J., 2002. Worst case execution time analysis of Object-Oriented programs. In the proceedings of *Proceedings of the Seventh International Workshop on*

Object-Oriented Real-Time Dependable Systems, San Diego, CA , USA, pp. 0071-0071.

Gustafsson, J., Ermedahl, A., Sandberg, C. and Lisper, B., 2006. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. Presented at the *Proceedings of the 27th IEEE International Real-Time Systems Symposium*.

Gustafsson, J., 2000. Analyzing execution-time of Object-Oriented programs using abstract interpretation. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University.

Gustafsson, J., Ermedahl, A. and Lisper, B., 2006. Algorithms for Infeasible Path Calculation. *Sixth International Workshop on Worst-Case Execution Time Analysis*, Dresden, Germany.

Hampapuram, H., Yang, Y. and Das, M., 2005. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 52-58.

Hedley, D. and Hennell, M. A., 1985. The cause and effects of infeasible paths in computer programs. *Presented at the Proceedings of the 8th International Conference on Software Engineering*, London, England.

Khedker, U., Sanyal, A., Karkare, B., 2009. Data flow analysis: theory and practice. *Taylor and Francis*.

Korel, B., 1996. Automated test data generation for programs with procedures, *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 209-215.

Liu, H. and Tan, H. B. K., 2009. Covering code behavior on input validation in functional testing. In *Information and Software Technology*, vol. 51(2), pp 546-553, 2009.

Liu, H. and Tan, H. B. K., 2008. Testing input validation in web applications through automated model recovery. In *Journal of Systems and Software*, vol. 81(2), pp. 222-233.

Malevris, N., Yates, D. F. and Veevers, A., 1990. Predictive metric for likely feasibility of program paths. *Information and Software Technology*, vol. 32, pp. 115-118.

McMinn, P., 2004. Search-based software test data generation: a survey: Research Articles. *Softw. Test. Verif. Reliab.*, vol. 14, pp. 105-156.

Moura, L. D. and Bjorner, N., 2008. Z3: an efficient SMT solver. Presented at the *Proceedings of the Theory and practice of software*, *14th international conference on Tools and algorithms for the construction and analysis of systems*, Budapest, Hungary.

Ngo, M. N. and Tan, H. B. K., 2008. Applying static analysis for automated extraction of database interactions in web applications. In *Information and Software Technology*, vol. 50(3), pp 160-175.

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

Ngo, M. N. and Tan, H. B. K., 2007. Detecting Large Number of Infeasible Paths through Recognizing their Patterns. In *Proceedings ESEC-FSE'07, Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, pp. 215-224.

Padmanabhuni, B. and Tan, H. B. K., 2011. Defending against Buffer-Overflow Vulnerabilities. *In IEEE Computer*, vol. 44 (11), pp 53-60.

Prather, R. E. and Myers, J. P., 1987. The Path Prefix Software Engineering. *IEEE Trans on Software Engineering*.

Robinson, A. J. A. and Voronkov, A., 2001. *Handbook of Automated Reasoning* vol. II: North Holland.

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

Tahbildar, H. and Kalita, B., 2011. Automated Software Test Data Generation: Direction of Research. *International Journal of Computer Science and Engineering Survey*, vol. 2, pp. 99-120.

Vergilio, S., Maldonado, J. and Jino, M., 1996. Infeasible paths within the context of data flow based criteria. In the *VI International Conference on Software Quality*, Ottawa, Canada, pp.310–321.

Williams, N., 2010. Abstract path testing with PathCrawler. Presented at the *Proceedings of the 5th Workshop on Automation of Software Test*, Cape Town, South Africa.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. and Stenstrom, P., 2008. The worst-case execution-time problem--overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 1-53.

Xie, T., Tillmann, N., Halleux, P. D. and Schulte, W., 2009. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. Presented at the *IEEE/IFIP International Conference on Dependable Systems & Networks*, Lisbon.

Zhang, J. and Wang, X., 2001. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, pp. 139-156.