2011 Fourth International Conference on Software Testing, Verification and Validation Workshops

IPEG: Utilizing Infeasibility

Mickaël Delahaye*
University of Orléans
LIFO
Orléans, France
Email: mickael.delahaye@gmail.com

***Abstract*—Infeasible paths are an hindrance to path-oriented test input generators. IPEG is a tool that takes a C program, and such an infeasible path of the program as input, and infers a possibly infinite family of infeasible paths. This paper gives a short description of the tool and the technique behind it.**

***Keywords*-Dynamic symbolic execution; test data generation; tool presentation**

I. Introduction

Path-oriented test input generators, notably dynamic symbolic execution tool such as PathCrawler [1], CUTE [2], or Pex [3], meet two hurdles on their way to practical use: the number of paths in the programs and the fact that among them a lot are in fact infeasible. This paper presents a tool that goes off the beaten track and is interested in the latter. Indeed, when a dynamic symbolic tool considers such a path to generate a test input, it proves that the path condition is actually inconsistent and consequently that the path is infeasible. This task is a waste of time, because the goal of a test input generator is not to report infeasible paths but to find test inputs for feasible paths.

This paper presents IPEG, Infeasible Path Explanation and Generalization, a tool that takes a program in a meaningful subset of C and an infeasible path as input and generalize from them infeasible paths. Because finding infeasible paths is as hard a task as generating test inputs, this tool takes opportunity of an already found infeasible path to generalize a possibly infinite family of infeasible paths. IPEG uses an original method to generalize infeasible paths. It first extracts a cause of the infeasibility and then finds paths that are infeasible for the same reason.

Please consider for the moment Fig. 1 that gives a simple C program, an infeasible path, and an infeasible path automaton generated by IPEG. The program contains an infinite number of infeasible paths to the statement `REACHME`. That is why, a typical dynamic symbolic execution tool using depth first strategy may have to unwind the loop a great number of times before considering the only feasible path reaching `REACHME`. Given one of the infeasible paths, IPEG generates an infeasible path automaton that recognizes them all. Similar cases exist for every strategy and do occur in real programs.

* This work was done in part while the author was still affiliated with the CEA, LIST, 91191 Gif sur Yvette Cedex, France.

```
1  int i=0;
2  while (i < n)
3    i++;
4  if (n==0)
5    REACHME;
```

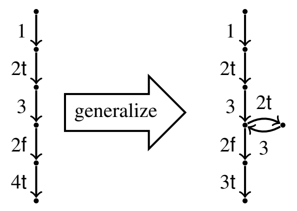


Fig. 1. Example of Infeasible Path Generalization

This short paper describes briefly the generalization method in a first section and the tool's characterics in a second section.

II. A Generalization Method

IPEG generalizes infeasible paths based on an infeasible path using three steps. First, from the path condition of the input infeasible path, it extracts the root of the infeasibility as an explanation. Second, it computes dependencies on the path in order to discriminate a family of infeasible paths. Finally, it builds an automaton capturing the family of infeasible paths.

A. Explaining the infeasibility

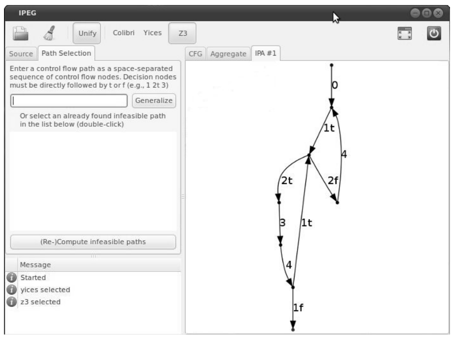
Using symbolic execution, we first compute a *path condition*, a constraint set that characterizes a path. Every solution of a path condition is an input activating the path. Conversely, a path is infeasible if the path condition is inconsistent.

From the path condition of the input infeasible path, it is possible to explain the infeasibility by extracting, using constraint programming techniques, an *explanation*, that is, a smaller set of constraints that actually causes the inconsistency. At this step, every explanation extraction technique can be used. However, the smaller the explanation the better the expected result.

By keeping track of which constraint(s) correspond to which statement, it is possible to know the statements that causes the infeasibility. But, in general, those statements are not enough to characterize the infeasiblity, because statements can be interpreted in different constraints depending on the context.

B. Discriminating Infeasible Paths

To ensure statements are interpreted the same way, this method observes *flow dependencies* on the path. Variables that may affect the interpretation of statements pointed by the explanation are identified. It is important to note that this step



Enter a control flow path as a space-separated sequence of control flow nodes. Decision nodes must be directly followed by t or f (e.g., 1 2t 3)

Or select an already found infeasible path in the list below (double-click)

Started
yices selected
z3 selected

Fig. 2. IPEG's graphical user interface

of the method introduces some overapproximation but only in presence of indirections (arrays, pointers).

The family of paths, called (π, K)-general infeasible paths, that contains the statements pointed by the explanation and respects the observed flow dependencies are necessarily infeasible. Indeed, the path condition of each of these paths contains the same explanation. However, it remains to actually find the family.

C. Building an Infeasible Path Automaton

Using the statements pointed by the explanation and the flow dependencies the third and last step builds an infeasible path automaton that matches only (π, K)-general infeasible paths. Considering the statements of interest in their original order, this step tries to find all the paths between pairs of successive statements that do not contradict the flow dependencies. These paths are in fact *definition-clear paths* with respect to the variables subject to path flow dependencies. By doing so for each and every pair of successive statements, the method builds an automaton that recognizes a subapproximation of the (π, K)-general infeasible paths.

III. A Tool

A. Input language

IPEG accepts programs in a meaningful subset of the C language:

- Integer operations (except bitwise operation)
- Arrays, structures
- Pointers (except function pointers)
- All control statements

But it does not handle yet floating point numbers, bitwise operation and integer overflows.

B. Software Requirements

IPEG is a tool developed mostly in OCaml for Linux-based systems. It may use various solvers:

- Colibri, a propagation-based constraint solver developed at the CEA used in at least three test generation tools (GATeL [4], PathCrawler [1], Osmose [5]),
- IC, another propagation-based constraint solver that comes with ECLiPSe Prolog,
- Z3, an SMT solver of Microsoft Research [6] used notably in Pex [3] and Sage [7],
- Yices, an SMT solver of SRI [8],
- and any SMT-LIB compatible solver [9].

Besides the solver, IPEG relies on Frama-C, an open-source analysis framework for C. Indeed, definition-clear paths are computed based on the value analysis [10].

C. Features

Given a program and an input infeasible, IPEG generalizes an infeasible path automaton. Also, IPEG is able to aggregate generalized infeasible path automata, by union. But the most import feature is its ability to tell if a path is or is not already proved infeasible by the generalization.

As well as a command line interface, IPEG's users have at their disposal a graphical user interface. This interface lets you choose a program, select a path, test the feasibility of a path, generalize an infeasible path automaton from the path, and aggregate multiple automata. As shown in Fig. 2, it also allows you to visualize the automaton, and its building steps, as graphs.

IV. Conclusion

IPEG is a tool that brings new perspectives of optimization to path-oriented test generation. Indeed, IPEG not only generalizes infeasible paths but it also aggregates generalized infeasible paths into a unique structure allowing us to check paths against this structure for a low cost.

Therefore, even if IPEG is not publicly available yet, its integration into test generation tools seems very interesting as suggested in experimental results presented in [11], and later experiments.

References

[1] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: Automatic generation of path tests by combining static and dynamic analysis," in *EDCC'05*, 2005, pp. 281–292.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE-13*. ACM Press, 2005, pp. 263–272.

[3] N. Tillmann and J. de Halleux, "Pex: White box test generation for .NET," in *TAP'08*, ser. Lecture Notes in Computer Science, vol. 4966, 2008, pp. 134–153.

[4] B. Marre and A. Arnould, "Test sequences generation from lustre descriptions: Gatel," in *ASE'00*. IEEE Computer Science, Sep. 2000.

[5] S. Bardin and P. Herrmann, "Structural testing of executables," in *ICST'08*, 2008, pp. 22–31.

[6] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS'08*, ser. Lecture Notes in Computer Science, vol. 4963, 2008, pp. 337–340.

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS'08: Network and Distributed System Security Symposium*. The Internet Society, 2008.

[8] B. Dutertre and L. de Moura, "The Yices SMT solver," 2006, tool paper available on line at `http://yices.csl.sri.com/tool-paper.pdf`.

[9] C. Barrett, S. Ranise, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," 2008. [Online]. Available: http://www.smt-lib.org/

[10] G. Canet, P. Cuoq, and B. Monate, "A value analysis for C programs," in *SCAM'09: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. Los Alamitos, California, USA: IEEE Computer Science, 2009, pp. 123–124.

[11] M. Delahaye, B. Botella, and A. Gotlieb, "Explanation-based generalization of infeasible path," in *ICST'10: International Conference on Software Testing, Verification, and Validation 2010*. Los Alamitos, California, USA: IEEE Computer Society, 2010, pp. 215–224.