Infeasible Path Generalization in Dynamic Symbolic Execution

Mickaël Delahaye[a], Bernard Botella[a], Arnaud Gotlieb[b]

[a] *CEA LIST, Software Safety Laboratory, PC 174, 91191 Gif-sur-Yvette Cedex, France*
[b] *SIMULA Research Laboratory, Certus Software V&V Center, Lysaker, Norway*

Abstract

Context: Automatic code-based test input generation aims at generating a test suite ensuring good code coverage. Dynamic Symbolic Execution (DSE) recently emerged as a strong code-based testing technique to increase coverage by solving path conditions with a combination of symbolic constraint solving and concrete executions.

Objective: When selecting paths in DSE for generating test inputs, some paths are actually detected as being infeasible, meaning that no input can be found to exercize them. But, showing path infeasibility instead of generating test inputs is costly and most effort could be saved in DSE by reusing path infeasibility information.

Method: In this paper, we propose a method that takes opportunity of the detection of a single infeasible path to generalize to a possibly infinite family of infeasible paths. The method first extracts an explanation of path condition, that is, the reason of the path infeasibility. Then, it determines conditions, using data dependency information, that paths must respect to exhibit the same infeasibility. Finally, it constructs an automaton matching the generalized infeasible paths.

Results: We implemented our method in a prototype tool called IPEG (Infeasible Path Explanation and Generalization), for DSE of C programs. First experimental results obtained with IPEG show that our approach can save considerable effort in DSE, when generating test inputs for increasing code coverage.

Conclusion: Infeasible path generalization allows test generation to know of numerous infeasible paths ahead of time, and consequently to save the time needed to show their infeasibility.

Keywords: Dynamic symbolic execution, explanation, test input generation

1. Introduction

Software testing is an essential part of today's software engineering, as the principal and often the only mean to ensure software reliability. Among the techniques that permit one to improve the quality of a test set, code-based testing, also known as white-box testing, plays an important role. Code-based testing implies the usage of the source code to select test inputs, to measure code coverage, to localize faults and eventually to propose automatically bug repairs. These last years, code-based testing has become more and more appealing with the emergence of new techniques and powerful tools. However, modern effective code-based testing has also a main limitation: high code coverage is often difficult or costly to reach, without compromising the efficiency of the technique. This paper is concerned with this challenge and describes a new cross-cutting technique that contributes to handle this issue.

As said above, the field of code-based testing has seen the emergence of new techniques and powerful tools, most of them being based on Dynamic Symbolic Execution (e.g., PathCrawler (Williams et al., 2005), DART (Godefroid et al., 2005), CUTE (Sen et al., 2005), SAGE (Godefroid et al., 2008) or PEX (Tillmann and de Halleux, 2008) just to name the pioneering tools). Dynamic Symbolic Execution (DSE) is a software testing and analysis technique which starts by selecting and executing a (feasible) path, by picking up a test input at random. Then, it computes a path condition by symbolically evaluating the instructions along the activated path. By refuting one decision of that path and exploiting a constraint solver, DSE determines a new test input which, by construction, covers another path in the program under test. Said otherwise, DSE tries to uncover test inputs which cover distinct paths that the ones that already covered, in order to increase path coverage. An important observation concerns path selection: when a path is activated by a test input, it is necessarily feasible but when a path is selected by refuting one decision over a (feasible) path, then its feasibility is no

Email addresses: mickael.delahaye@cea.fr (Mickaël Delahaye), bernard.botella@cea.fr (Bernard Botella), arnaud@simula.fr (Arnaud Gotlieb)

```
/* Let x be the input,
   and res the output        */
[abs := x; i := 2]^a;
[res := 1]^b;
if [abs < 0]^c then
  [abs := -abs]^d;
while [i ≤ abs]^e do
  [res := res × i; i := i+1]^f;
if [x < 1]^g then
  [res := res + 5]^h;
```

Figure 1: A program with many infeasible paths

more guaranteed. Whenever DSE considers an infeasible path, then the constraint solver tries to prove the unsatisfiability of the path condition. Obviously, this task is not formally required and corresponds to a waste of time because the goal of DSE is to find new test inputs, and not to report path infeasibility. Even if detecting all the infeasible paths is impossible[1], studies have shown that they are ubiquitous in computer programs (Yates and Malevris, 1989) and that avoiding them when testing programs is highly desirable (Ngo and Tan, 2008).

As a simple motivating example, please consider the program of Fig. 1. On this program, a typical DSE tool might stumble onto a lot of infeasible paths, as shown on the run given on Fig. 2. In step (1), an input is arbitrarily chosen (e.g., $x = 2$), the program is executed, and the activated path is traced ($a\,b\,c^{\mathsf{f}}\,e^{\mathsf{t}}\,f\,e^{\mathsf{f}}\,g^{\mathsf{f}}$), as shown on the figure. In step (2), the tool chooses a new path to cover (indicated with dotted arrows) based on the current path using a depth-first strategy. The tool computes a path condition for this new path (indicated beside the path), and passes it on to a solver. The solver answers negatively (indicated by an X). Indeed, the path condition is inconsistent ($2 \leq x$ contradicts $x < 1$). In other words, the generation has met its first infeasible path $a\,b\,c^{\mathsf{f}}\,e^{\mathsf{t}}\,f\;e^{\mathsf{f}}\,g^{\mathsf{t}}\,h$. In (3), the tool tries to activate another path with the same method. This time, the solver gives a solution to the path condition. This solution ($x = 3$) is used as input to a concrete execution of the program to get a full activated path. This path iterates the loop one more time. In (4), the tool tries to activate the statement h, and for the very same reason as step (2), the attempt fails. And so on and so forth, going deeper and deeper in the loop. Hopefully, the tool bounces back either on an arbitrary limitation of the path length or the maximal value of the data type (only for finite data type). After covering a first path with no iteration in the loop, this hypothetical tool indeed finds an input that activates the statement h. During the test input generation, a lot of paths are proved infeasible. Indeed, a manual code review lets us confirm that, if the control flow passes through the "then" branch of the first conditional and through at least one iteration in the loop, it cannot go into the "then" branch of the second conditional. This family can be represented by an automata given of Fig. 3. Every path for which a prefix is recognized by the automata is necessarily infeasible.

Though one can argue another search strategy might perform better on this particular example, such traps exist for every strategy. Moreover, real programs do contain families of similar infeasible paths. Recommended programming practices, such as code reuse, modularity, and assertions, are often source of possibly redundant checks leading to numerous infeasible paths. As we will see later, even well known algorithms possess such families of infeasible paths.

Motivated by such cases, we propose in this paper a technique that allows test input generators to detect early and to skip numerous infeasible paths. This technique takes opportunity of the detection of an infeasible path by the test input generator to generalize to a possibly infinite family of infeasible paths. The method consists first of extracting the "essence" of the infeasibility from the path condition. Then, by combining data dependency information and finite state automaton operations, our approach constructs an infeasible path automaton, a representation of an infeasible path family of the program. Finally, this automaton can be used to detect paths belonging to the family for the cost of matching a regular expression.

To evaluate our approach, we developed a modular tool called IPEG (Infeasible Path Explanation and Generalization) for programs in C. It can be parametrized by any solving procedure. For our experiments, we used three constraint solvers (i.e., Colibri, Yices and Z3) that are currently used in dynamic symbolic execution tools. We evaluated our approach at two levels. First, the unitary evaluation checks the effectiveness of the generalization method to prove path infeasibility against an exhaustive symbolic execution. Second, an integrated evaluation checks how a naive integration of the method in a test input generator affects the performances. These experiments show that our approach can save considerable computation time during test generation.

Paper organization. Section 2 gives essential notations and background notions to understand the infeasible path generalization method. Section 3 presents the method in depth with a number of examples. Notions such as data dependencies and infeasible path automaton are introduced in this section. Section 4 discusses the integration of the proposed method within a dynamic symbolic execution procedure. Section 5 contains the results of our experimental evaluation of method. Section 6 positions our proposed method into state-of-the-art path infeasibility analyses. Finally, Section 7 concludes the paper and draws a couple of perspectives to this work.

[1]This problem was proved undecidable in general (Weyuker, 1979).

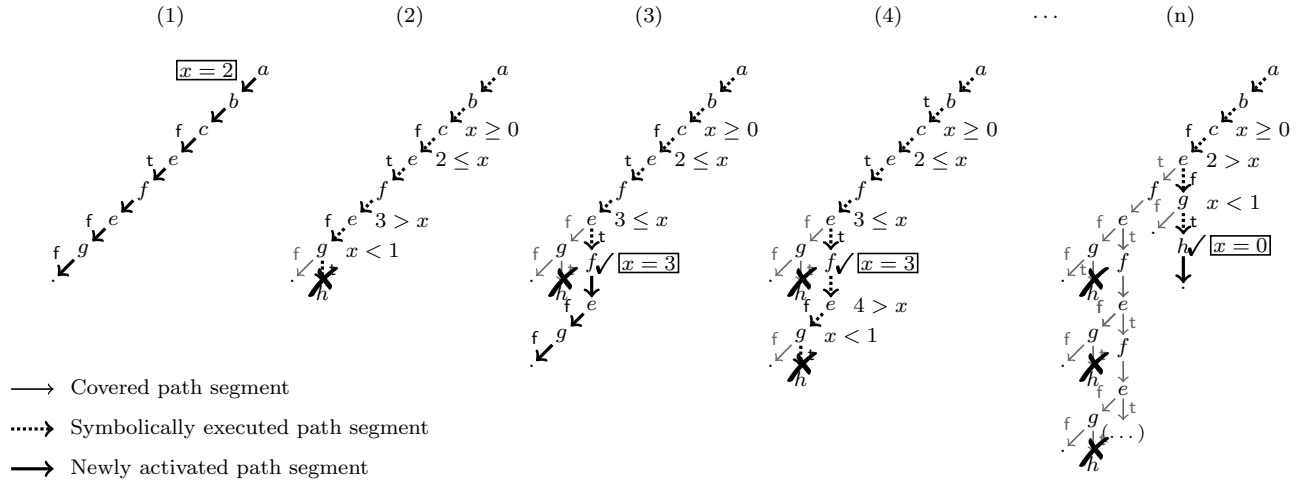

Figure 2: Steps of a typical dynamic symbolic execution test input generation

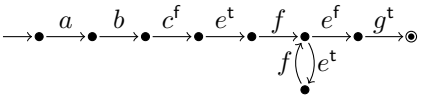

Figure 3: An infeasible path automaton

2. Background and Notations

This section first defines some notions and notations about programs, paths and feasibility. Then, it introduces and reviews the notion of constraint-based explanations.

2.1. Program and Path

For the sake of clarity, we will use a simple imperative language for representing programs. Fig. 1 gives a concrete example of the syntax used in the paper. It is important to note that simple statements (assignment or **skip** statement) and tests (that is, conditions on loops and conditional constructs) are labeled.

A *program path* is a sequence of program statements allowed by the flow relation defined on the studied language. In the paper, a path is noted by a sequence of augmented labels on a particular program. An *augmented label* is a label possibly followed by a letter, t or f, to explicit the truth value, respectively true or false, when the label points to a test. If x is an augmented label, $\mathsf{label}(x)$ denotes the simple label (without any letter).

For instance, on the program of Fig. 1, $a\, b^{\mathsf{t}}\, c\, e^{\mathsf{f}}\, g^{\mathsf{t}}\, h$ is a path that does not enter the loop and goes through the "then" branches of the two conditionals. Note however that the sequence $a\, e^{\mathsf{t}}\, i$ does not respect the program's control flow and as such is not to be considered a program path.

2.2. Feasibility and Path Condition

A program path is said to be *feasible* if there is at least one particular input that activates it. Conversely, a path is *infeasible* if there is no input that activates it.

A *path condition* of a program path given a symbolic input vector is a conjunction of constraints on the symbolic input vector (and possibly other logical variables) that characterizes the path's execution. Formally, given C a path condition of a path π, π is feasible (resp. infeasible) if and only if C is consistent (resp. inconsistent). Also, if C is consistent, any solution of C gives a test input vector activating π. Such a path condition can be computed using symbolic execution. *Symbolic execution* (King, 1975) concerns the execution of a program on symbols, in other words, logical variables, rather that concrete inputs. Sec. 3.1.1 describes in details our technique's need regarding the symbolic execution.

Given a set S of paths, a path p is a *shortest path* of S, if, for all $p' \in S$ there is no non-null s such that $p' \cdot s = p$. We can also see shortest paths are the maximal elements of the path set partially ordered by the prefix relation ($a \sqsubseteq b$ iff a is a prefix of b).

2.3. Data Dependencies

This section first defines two specialized variants of data dependencies before recalling the definition of definition-clear path.

A *DU (definition-use) path dependency* occurs on program path π between the indexes i and j with respect to the variable v if and only iff:

- $i > j$ (a conventional orientation),
- π_i reads v and π_j writes v,
- and there is no path index k with $i > k > j$ such that π_k writes v, in other words, the subpath $(\pi_k)_{i>k>j}$ is a def-clear paths w.r.t. the variable v.

A *UU (use-use) path dependency* occurs on program path π between the indexes i and j with respect to the variable v if and only if:

- $i > j$ (a conventional orientation),

- π_i reads v and π_j reads v,
- and there is no path index k with $i > k > j$ such that π_k writes v, in other words, the subpath $(\pi_k)_{i>k>j}$ is a def-clear paths w.r.t. the variable v.

It is important to note that these two kinds of dependencies are special because they are considered on a single path (possibly, an infeasible one). However, these dependencies can only be overapproximated in some cases. Indeed, the variables read or written at a given statement cannot be known exactly in presence of indirections (arrays, pointers).

A *definition-clear (def-clear) path* from label n to label m with respect to a set of variables V is a path that goes from n to m without modifying any variable of V.

2.4. Finite-State Automata

A *(finite-state) automaton* is a tuple $\langle Q, \Sigma, \Delta, I, F\rangle$, where Q is a set of states, I and F are subsets of Q indicating respectively the initial states and the final states, Σ a set of symbols, i.e, an *alphabet*, and Δ is a set of transitions $\mathcal{P}(Q \times \Sigma \times Q)$. A *deterministic (finite-state) automaton* is a finite-state automaton that has at most one initial state, and such that for each state there is at most one outgoing transition labeled with the same symbol.

Given an alphabet Σ, a *word* is a sequence of symbols of Σ. An automaton $\langle Q, \Sigma, \Delta, I, F\rangle$ *recognizes*, or *matches*, a word of Σ, $w = (\ell_1, \ell_l)$, if and only if there is a sequence of states in $(q_1, \ldots, q_{n+1})$, such that for all i, $q_i \in Q$, $q_1 \in I$, $q_{n+1} \in F$, and, for $i = 1 \ldots n$, $(q_i, \ell_i, q_{i+1}) \in \Delta$.

A *language* is a (possibly infinite) set of words. The *recognized language* of an automaton A is the set of words recognized by the automaton, noted $L(A)$. Two automata are equivalent if there recognized language are equal.

An *union* of two automata A and B, noted $A \sqcup B$, is an automaton C such that w is in $L(C)$ if and only if w is either in $L(B)$ or in $L(C)$, that is $w \in L(B) \cup L(C)$.

A *concatenation* of A and B, noted $A \circ B$, is an automaton C such that:

- If w is in $L(C)$, there is $u \in L(A)$ and $v \in L(C)$ such that $w = u \cdot v$, where "$\cdot$" indicates the concatenation of words.
- Conversely, if $u \in L(A)$ and $v \in L(C)$, then $u \cdot v$ is in $L(C)$.

In the rest of the paper, we suppose that techniques to compute the union and the concatenation are known.

2.5. Explanation

An important part of the method is to determine the reason why a constraint system is inconsistent. More precisely, the method needs to know what part of a constraint system is fundamentally false, that is, to find an *explanation.*

An *explanation*, or *unsatisfiable core*, is a subsystem of an inconsistent constraint system that is inconsistent by itself. An explanation is *minimal* if no subsystem of the explanation is also an explanation, that is, if all parts of the explanation are essential to the inconsistency.

There are at least two ways to find an explanation: first, the *intrusive* way, which consists in intrusively tracing back the solving process to the parts of the constraint system that lead to inconsistency; second, the *non-intrusive* techniques which consider the constraint system as a conjunction of individual constraints and tries to identify the ones that lead to inconsistency by successive external tests.

2.5.1. Intrusive methods

These methods extract an explanation from the constraints used by the solver's reasoning. Constraint programming tries to learn from failure since the late 70s by keeping track of *nogoods*. A *nogood* consists of a partial assignment of variables and a subset of the considered constraint system justifying this assignment is inconsistent. More recently, Jussien et al. (2000), adapted the recording of such nogoods to constraint propagation algorithms. Constraint propagation finds a constraint system unsatisfiable if any variable domain is empty. Therefore, when recorded, the conjunction of explanations for eliminating values from domain forms an explanation of the overall constraint system. For instance, consider the inconsistent constraint system $x \neq y \wedge x + y = 0 \wedge xy = 1$ for x, y in $\{0, 1\}$. Its inconsistency can be proved by the solver with the following reasoning:

$$x = 1 \overset{x \neq y}{\Rightarrow} y = 0 \overset{x+y=0}{\Rightarrow} \bot$$
$$x = 0 \overset{x \neq y}{\Rightarrow} y = 1 \overset{xy=1}{\Rightarrow} \bot$$

As the domain of x has been entirely explored, the constraints used in this reasoning form an explanation of the inconsistency of the constraint system. Note however that explanations obtained this way are not usually minimal because an automatic reasoning can be terribly convoluted. For instance, this particular reasoning uses every constraint of the system, and consequently the inferred explanation is equal to the whole constraint system.

Other kind of solvers compute and use such explanations. In fact, under the generic terminology of *clause learning*, SAT solvers exploit conflicts to speed up the unit propagation of the DPLL algorithm (Kroening and Strichman, 2008). Moreover, Zhang and Malik (2003) have shown that an explanation can be computed as a by-product of a proof generation in SAT solvers. As noted by Cimatti et al. (2007), this technique was adapted to SMT solvers, for instance, MathSAT. Other SMT solvers (e.g., Yices) introduce selector variables to identify each part of the formula, so that, if it is inconsistent, the conflict clause points directly to an explanation.

2.5.2. Non-intrusive methods

The idea is to iteratively test the consistency of subsystems of the constraint system until a minimal explanation is found. For instance, an explanation of the above constraint system $x \neq y \wedge x + y = 0 \wedge xy = 1$, for x, y in $\{0, 1\}$, can be found by successively checking the consistency of its subsystems: $x \neq y$, $x \neq y \wedge x + y = 0$, etc. Here $x, y \in \{0, 1\}, x \neq y \wedge x + y = 0$ is a minimal explanation. Note that this explanation is not unique as one could replace $x + y = 0$ by $xy = 1$ and obtain another minimal explanation.

Several non-intrusive algorithms have been proposed but the recursive dichotomic algorithm of Junker (2004), QuickXplain, is one of the most efficient as it runs in $O(k \log \frac{n}{k})$ in the worst case instead of $O(nk)$ where n is the size of the constraint system and k the size of the explanation to be found.

Junker's algorithm is adapted to constraint solver that may return an inconclusive verdict (undecidable theory, arbitrary timeout). As such, it computes one of the minimal subsets the constraint solver Γ can state inconsistent, noted here Γ-minimal explanation. Note that in the absence of inconclusive verdict, Junker's algorithm returns minimal explanations.

At each recursion step, the algorithm considers a background set of constraints (initially empty) and an active set of constraints (initially the whole constraint system). First, it checks if the background is not inconsistent and if the active set of constraints is not empty. If either condition is met, it returns an empty set. Otherwise, it continues by checking whether the active set of constraints is a singleton. If it is the case, this singleton is returned. Otherwise, the active set of constraints is partitioned into two parts, and the algorithm is called recursively on each part: first the second part, with the union of the current background and the first part as background, then the first part with the union of the current background and the result of the first recursive call as background. Finally, it returns the union of the results of both recursive calls.

Iterative methods are very susceptible to the iteration order. Partitioning allows Junker's method to counteract the phenomenon and to actually be more efficient under the hypothesis that the considered constraint set has a small explanation.

There are other approaches to this problem. Indeed, many methods exist to compute more or less precise unsatisfiable cores by combining techniques from both ways, but mostly for Boolean formulae (e.g., Dershowitz et al., 2006). That is why Cimatti et al. (2007) propose to use an external Boolean unsatisfiable core extractor to compute a small explanation of an SMT formula.

For our experiments, we choose to explore two ways to compute an explanation: the off-the-shelf unsatisfiable core provided by some SMT solvers, often imprecise but fast, and Junker's explanation algorithm, very precise, generic but demanding in terms of solver calls. We think these methods are representative of the trade-off existing between precision and time.

3. Infeasible Path Generalization

This section first describes the method of infeasible path generalization in three steps. Then, it sums up the method and applies the method to a full example. Finally, we proposes an integration to dynamic symbolic execution-based test generation.

3.1. Explaining the Infeasibility

The infeasible generalization method takes as input an infeasible path π of an imperative program. The first step of the method is concerned about finding why the path is infeasible.

A path can be infeasible for various reasons, but every infeasibility is captured by the path condition. But, a path condition can be as cluttered as the program code, that is, containing redundant constraints, or unrelated groups of constraints. Hence, it is interesting to extract from the path condition a more concise explanation using constraint techniques.

So, first, this section explains how the method computes a suitable path condition. Then, it discusses about methods to extract an explanation from the path condition. Finally, this section questions the relation between the constraint-level explanation and the program statements.

3.1.1. Symbolic Execution without Substitution

Explaining the infeasibility requires a precise path condition that contains as much details as possible about the execution. This method assumes a symbolic execution without substitution (constant/expression propagation) such, in the path condition C_π of a path π:

- Each statement or test met at π_i corresponds to at least one constraint of the path condition C_π tagged with the index i in order to know to what statement correspond a particular constraint
- Every variable use or definition corresponds to the use of a logical variable that specifically indicates the accessed instance of the variable (so as to know based on a constraint of the path condition which program variables was read or written and at which statement occurrence)

Sec. 4.1 further discusses this requirement and a way to alleviate it by trading off precision.

If we consider the example of Fig. 1 and the first infeasible path met during the test generation, that is, $a\, b\, c^{\mathsf{f}}\, e^{\mathsf{t}}\, f\, e^{\mathsf{f}}\, g^{\mathsf{t}}$, a suitable path condition is given on Table 1. Each constraint is labeled with the index on the path. For instance, $[abs_1 \geq 0]_3$ is computed at the third element of the path which corresponds to an occurrence of the test c with the truth value f. Also, the path condition

Table 1: Path Condition of $a\,b\,c^{\mathsf{f}}\,e^{\mathsf{t}}\,f\,e^{\mathsf{f}}\,g^{\mathsf{t}}$

i	π_i	Test/Statement	**Path Condition**
1	a	$[abs := x;\ i := 2]^a$	$[abs_1 = x_0]_1,\ [i_1 = 2]_1$
2	b	$[res := 1]^b$	$[res_1 = 1]_2$
3	c^{f}	$[abs < 0]^c$	$[abs_1 \geq 0]_3$
4	e^{t}	$[i \leq abs]^e$	$[i_1 \leq abs_1]_4$
5	f	$[res := res \times i;$ $i := i + 1]^f$	$[res_2 = res_1 \times i_1]_5,$ $[i_2 = i_1 + 1]_5$
6	e^{f}	$[i \leq abs]^e$	$[i_2 > abs_1]_6$
7	g^{t}	$[x < 1]^g$	$[x_0 < 1]_7$

uses a fresh logical variable at each program variable definition noted by the name of the original variable and an index starting to 1, zero being reserved for input variables (like x_0).

3.1.2. Extracting the Explanation

A path condition explains the infeasibility, but, as noted earlier, a more precise explanation is desirable. In fact, the constraints of the path condition are like common traits that the generalized paths must share: the less they are, the more likely they are to be shared. Consequently, even if any explanation extraction method may be used, a precise result is preferred.

Again on the example of Fig. 1, the inconsistency of the path condition of $a\,b\,c^{\mathsf{f}}\,e^{\mathsf{t}}\,f\,e^{\mathsf{f}}\,g^{\mathsf{t}}$, shown in Table 1, can be explained by four constraints $[abs_1 = x_0]_1$, $[i_1 = 2]_1$ $[i_1 \leq abs_1]_4$, and $[x_0 < 1]_7$. This constitutes a minimal explanation of the path condition and as such an ideal base to generalize infeasible paths.

3.1.3. From Constraints to Statements

Once the constraints causing the inconsistency have been identified, it is quite natural to trace them back to actual program statements. That is why each constraint is labeled with the path index from which it originates. Yet, this is not enough to characterize infeasible paths. Indeed, there may exist feasible paths that pass through all the statements corresponding to some explanation.

As shown previously, the infeasibility of $\pi = a\,b\,c^{\mathsf{f}}\,e^{\mathsf{t}}\,f\,e^{\mathsf{f}}\,g^{\mathsf{t}}$ can be explained by $[abs_1 = x_0]_1$, $[i_1 = 2]_1$, $[i_1 \leq abs_1]_4$, and $[x_0 < 1]_7$. These constraints correspond to path elements π_1, π_4, and π_7, that is, to the label a, e^{t}, and g^{t}. However, although the path $\pi' = a\,b\,c^{\mathsf{t}}\,d\,e^{\mathsf{t}}\,f\,e^{\mathsf{f}}\,g^{\mathsf{t}}$ shares these exact labels, this path is feasible (for $x = -2$). Indeed, statements interpreted in different contexts give different constraints. Here, the test e^{t} at index 5 on π' is interpreted as $[i_1 \leq abs_2]_5$ and not $[i_1 \leq abs_1]_4$.

3.2. Tracking Data Dependencies

After finding the culprit constraints of the path condition in the previous section, this section explains the data dependencies related to the statements corresponding to these constraints. The objective is to find data dependencies such that every path respecting them is necessarily infeasible.

On the path $a\,b\,c^{\mathsf{f}}\,e^{\mathsf{t}}\,f\,e^{\mathsf{f}}\,g^{\mathsf{t}}$ of the example program of Fig. 1, there are multiple path dependencies as defined in the Sec. 2.3. For instance, there is a UU path dependency between the indexes 7 and 1 with respect to the variable x, because $\pi_1 = a$ and $\pi_7 = g^{\mathsf{t}}$ reads the same x. Also, there is a DU path dependency between 4 and 1 with respect to abs, because $\pi_4 = e^{\mathsf{t}}$ reads the value of abs defined at $\pi_1 = a$.

For the purpose of infeasible path generalization, these two kinds of dependency must be treated without distinction. In fact, both dependencies are indicated with a unique notation: on a path π, $i \rightarrow_V j$ indicates a path dependency (UU and/or DU) between the indexes i and j w.r.t. a set of variables V.

Infeasible Chain

Path dependencies concerning a path's infeasibility can be stored in a specific structure, named $infeasiblechain$. Given a path π and an explanation K of its infeasibility, an *infeasible chain* of (π, K) is a pair $\langle I, D\rangle$ where I are the indexes corresponding to K and D the set of path dependencies between I.

The interesting thing about infeasible chains is that every path that "respects" an infeasible chain is also infeasible. Formally, a path π' is said to *respect* an infeasible chain $\langle I, D\rangle$ of (π, K), if there is a set I' of indexes on π' and a strictly increasing bijective mapping $f : I \rightarrow I'$, such that:

- For all i in I, π_i is the same augmented label as $\pi_{f(i)}$.
- For all $i \rightarrow_V j \in D$, there is no definition of any variable of V between $f(i)$ and $f(j)$ on π' (both excluded).

It is important to note that the infeasible chain of (π, K) discriminates a family of infeasible paths called the (π, K)-*general infeasible paths.*

On the example of Fig. 1, from the infeasible path $\pi = a\,b\,c^{\mathsf{f}}\,e^{\mathsf{t}}\,f\,e^{\mathsf{f}}\,g^{\mathsf{t}}$, the explanation $K = \{[abs_1 = x_0]_1, [i_1 = 2]_1, [i_1 \leq abs_1]_4, [x_0 < 1]_7\}$, it is possible to get the infeasible chain $\langle I, D\rangle$ of (π, K), where I is the set $\{1, 4, 7\}$ and D the path dependencies observed on π for the indexes I. This infeasible chain is represented on Fig. 4. On this figure, UU and DU path dependencies are represented differently only for the sake of precision and clarity.

Note that this infeasible chain can directly be extracted from the path condition. Path dependencies are present directly in the constraints of K. Indeed, two indexes i and j are dependent with respect to v if and only if there are some constraints $[c]_i$ and $[c']_j$ of K and k a variable index such that c and c' both uses the logical variable v_k.

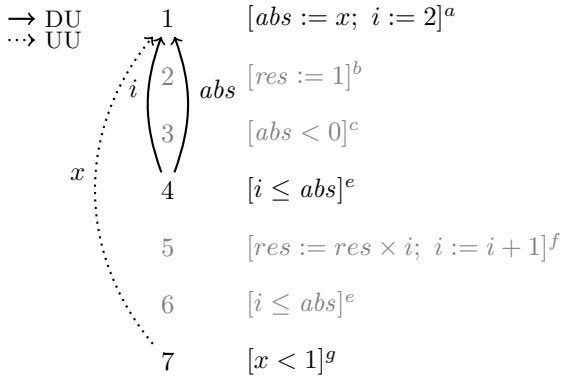

Figure 4: Infeasible Chain of (π, K)

Input: π an infeasible path, and D an infeasibility chain of π
Output: an infeasible path automaton

function buildAutomaton$(\pi, \langle I, D\rangle)$
/* *Initialize A with an automaton matching only (π_1)* */
$A := \langle\{0,1\}, \mathbf{Labels}, \{0 \xrightarrow{\pi_1} 1\}, \{0\}, \{1\}\rangle$;
$R := I \setminus \{1\}$;
$i := 1$;
while $R \neq \varnothing$ **do**
  $j := \min R$;
  /* *Computes the variables to protect between i and j* */
  $V := \bigcup\{U | \exists i', j' \in I, i' \leq i \wedge j' \geq j \wedge (i' \rightarrow_U j') \in D\}$;
  /* *Computes the def-clear paths from the successor of π_i (included) to π_j (excluded) w.r.t. the variables V* */
  $B :=$ defClearPaths(label(π_{i+1}), label(π_j), V);
  /* *Concatenate A, the def-clear paths B, and a mandatory passage by π_j* */
  $A := A \circ B \circ \langle\{0,1\}, \{0 \xrightarrow{\pi_j} 1\}, \{0\}, \{1\}\rangle$;
  $R := R \setminus \{j\}$;
  $i := j$;
return A;

Figure 5: buildAutomaton algorithm

3.3. Building an Infeasible Path Automaton

Now that we have a method to find an infeasible chain, it remains to actually find an automaton capturing (π, K)-general infeasible paths.The main idea is to fill in the gaps of the infeasible chain by every possibility offered by the program that respects the dependencies.

Fig. 5 gives an algorithm that takes an infeasible chain for a program path π and constructs an automaton that only recognizes paths respecting the chain. To anchor the generated automaton to the start of the program, this algorithm starts with an automaton matching A only the first path element π_1, that is, the program start, even if this path element is not present in the infeasible chain. It sets i to 1.

Then, the algorithm considers each path index j in the infeasible chain (except 1) in increasing order:

- First, it computes the def-clear paths between i and j of the infeasible chain with respect to the variables V as an automaton B. The variables V consists in all the variables for which there is a dependency from a path index less or equal to i and to a path index greater or equal to j.
- Second, it updates A by concatenating B and i takes the value of j.

Table 2: Building Steps of an Infeasible Path Automaton

Step	i	j	V	A
0	1	-	-	
1	1	4	$\{x, abs\}$	
2	4	7	$\{x\}$	

Table 2 details the steps to build an infeasible path automaton from the infeasible chain given on Fig. 4. At each step, this table indicates the value of i, j, V just before the call to defClearPaths and the value A after the concatenations. Initially, the automaton contains only one transition labeled a. Step 1 puts in A the concatenation of A, the only def-clear path from $\pi_i = \pi_1 = a$ to $\pi_j = \pi_4 = e^{\mathsf{t}}$ (both excluded) with respect to the variable $V = \{x, abs\}$, and a single-transition automaton corresponding to e^{t}. Finally, step 2 appends the def-clear paths from $\pi_4 = e^{\mathsf{t}}$ to $\pi_7 = g^{\mathsf{t}}$ with respect to the variable x and the single transition g^{t} to the automaton A. The method gives us an infeasible path automaton equivalent to the minimized automaton of Fig. 3.

Finding Definition-Clear Paths

Given a set of potentially written variables at each program statement, there is a simple way to compute an automaton matching every def-clear path with respect to particular set of variables, if we know the variables potentially modified by each statement on all paths. Such information can be computed beforehand by static analysis. Fig. 6 gives the pseudocode of this algorithm.

If *src*, *dst*, and V are respectively the source, the destination, and the set of variables, this algorithm starts from

Input: s and d two program labels and V a set of variables
Output: an automaton accepting only def-clear paths from s to d w.r.t. the variables V
function defClearPaths(s, d, V)
/* *Given the program labels considered as a set of states Q and the program transitions δ, $\langle Q, \delta, s, d\rangle$ is an automaton accepting all program paths from s to d* */
$Q :=$ labels(**Prog**); $\delta := \{a \xrightarrow{a^d} b | (a \xrightarrow{d} b) \in$ flow(**Prog**)$\}$;
/* *Removes statements writing any variable of V* */
foreach $n \in Q$ **do**
  if maywrite$(n) \cap V \neq \varnothing$ **then**
    $\delta := \delta \setminus \{a \xrightarrow{d} b \in \delta | a = n \vee b = n\}$;
    $Q := Q \setminus \{n\}$;
/* *Keeps only reachable states* */
return trim$(\langle Q, \mathbf{Labels}, \delta, s, d\rangle)$

Figure 6: defClearPaths algorithm

an automaton representing our program. Then, it removes transitions that may modify a variable of V. Finally, it removes states that are not reachable from the source or not co-reachable from the destination. Because the control flow is deterministic in this programming language, the obtained automaton is necessarily deterministic.

3.4. Method Rundown

To summarize, given a program P and an infeasible path π found in P, our proposed infeasible path generalization method consists in the three simple following steps:

1. Computing the so-called path condition of π, and automatically extracts from it, an *explanation* K (i.e., a smallest subset of inconsistent constraints) ;
2. Computing the *infeasible chain* $\langle I, D\rangle$ from this explanation K ;
3. Building an *infeasible path automaton*, by calling buildAutomaton$(\pi, \langle I, D\rangle)$, which generalizes π by finding other infeasible paths sharing the similar explanation K than π.

As said previously, this method finds applications in different areas including test case generation through symbolic execution and dynamic symbolic execution. Indeed, it can be used each time a single infeasible path is found with a proved constraint unsatisfiability, so that further analyses can avoid looking for infeasible paths sharing the same explanation of infeasibility.

3.5. Correctness and Completeness

This generalization method is *correct*, that is, (1) the method output contains only infeasible paths and (2) the algorithm does terminate under the hypothesis the solving procedure does terminate.

First, the generalization soundly detects infeasible paths. Indeed, it ensures that all paths recognized by the automaton share a common infeasibility explanation. Consider the following proof sketch. Based on an input infeasible path, the generalization first extracts an explanation from the inconsistent path condition of the input path and computes an infeasible chain. First, the algorithm ensures by construction that the automaton recognizes only paths that respect the infeasible chain. Then, given a correct symbolic execution without substitution, for each of the recognized paths, the path condition contains constraints similar to the identified explanation, because the statement responsible for the infeasibility are also executed (possibly with some variable changes). The respect of the dependencies implies that there is a substitution of variables such that the explanation with substituted variables is an exact subset of the path condition. That is why, the path condition is inconsistent. Finally, recognized paths are indeed infeasible.

Second, concerning termination, any preliminary static analysis (like abstract interpretation) ensures termination by design. Also, if the solver does terminate (possibly via a timeout mechanism), the explanation extraction terminates. Finally, because the other steps of the method consider a finite number of statements or constraints, the whole method terminates.

However, this method is *not complete*, in the sense that the generalization may miss some path whose path condition includes the same explanation K modulo a variable substitution.

First, the generalization is limited because the method takes its roots in the program syntax. For instance:

- Other statements or statement orders may result in constraints equivalent to K. For instance, a program can contain multiple copies of a statement leading to the same explanation K in distinct branches.
- Syntax-based dependencies are overprotective in presence of pointers or arrays.

Note however that rooting the method in the program syntax is also an advantage, because it allows us to test the infeasibility on simple objects at syntax level rather that on complex objects, such as constraint systems.

Second, another limit of the generalization comes from its dependency to the explanation. Actually, any additional constraint in the explanation may eliminate infeasible paths from the resulting automaton because they do not contain the program statement corresponding to this additional constraint.

Finally, one targeting the completeness of the generalization of an infeasible path might also want to address the multiplicity of the reasons causing this infeasible path. Indeed, in terms of constraint system, there may be multiple minimal explanations in a path condition. Here, this method chooses one explanation to generalize infeasible paths. By choosing another explanation, this method might find different infeasible paths. In fact, the outputs of the method for different input explanations might be

distinct as well as overlapping. Note that this particular limitation can be lifted by applying our method on every minimal explanation of the path condition.

3.6. Complexity Analysis

The computational cost of our approach is expressed in terms of the actual size n of the infeasible path (i.e., number of statements) used to feed the generalization process. This process is divided in the three following steps:

1. Assuming that a dichotomic extraction method is used, and that a constraint is associated to each statement of the path, the worst-case explanation process costs n^2 calls to the constraint solver. Depending on the cost of a satisfiability check by the constraint solver (usually a non-polynomial process in the worst case), we can end up with a very high computational costs. Hopefully, constraint solver calls are temperated with time-out mechanisms which compromise the result quality to preserve efficiency. So, let's assume that any call to the constraint call is bounded by a constant, we can safely consider that the explanation process takes $O(n^2)$ initial steps ;
2. Second, assuming that each constraint in the explanation is associated to no more than a small subset of program statements and that each data dependency is bounded by the finite number of variables of the program, constructing the infeasible chain is a linear process w.r.t. n. In fact, as every variable read-write operation is bounded in time, the infeasible chain can be computed in $O(n)$ operations.
3. Finally, building the automaton involves the computation of def-clear paths, i.e., an $O(m)$-procedure where m stands for the total number of statements in the program. As the def-clear path automaton contains at most m states and assuming nondeterministic automata, the concatenation operation takes $O(1)$ (only one accepting state). Overall the automaton construction takes $O(n.m)$ operations in the worst case.

So, to summarize, the complexity analysis of the overall process says that the infinite path generalization process takes $O(n(n+m))$ steps where n is the number of statements over the path (with possible repeated occurrences) and m is the number of statements of the program.

Note that there is two different ways of obtaining a deterministic infeasible path automaton: either to use deterministic operations at each step; or to determinize the result at the end. Both methods theoretically result in an exponential time and state complexity. However, in practice, the worst case does not happen: although the state space does increase considerably, the automaton construction takes a reasonable time, which is considerably less that the time required to extract an explanation. We explain this phenomenon by the nature of control-flow graph that are rather sparse and mostly linear and because the algorithm ensures that the concatenated automaton (def-clear paths) are deterministic.

```
/* Let u,v be the inputs,
   and v the output          */
while [u > 0]^a do
    if [v > u]^b then
        [t := u]^c;
        [u := v]^d;
        [v := t]^e;
    [u := u − v]^f;
```

Figure 7: GCD: Program Code

Table 3: GCD: Path condition of $\pi = 1^{\mathsf{t}}\,2^{\mathsf{t}}\,3\,4\,5\,7\,1^{\mathsf{f}}$

i	π_i	**Test/Statement**	**Constraint**
1	a^{t}	$[u > 0]^a$	$[u_0 > 0]_1$
2	b^{t}	$[v > u]^b$	$[v_0 > u_0]_2$
3	c	$[t := u]^c$	$[t_1 = u_0]_3$
4	d	$[u := v]^d$	$[u_1 = v_0]_4$
5	e	$[v := t]^e$	$[v_1 = t_1]_5$
6	f	$[u := u - v]^f$	$[u_2 = u_1 - v_1]_6$
7	a^{f}	$[u > 0]^a$	$[u_2 \leq 0]_7$

3.7. A Complete Example

Even well designed pieces of program like mathematical algorithms contain infeasible paths. This section concerns the applicability of our method to the Euclidean algorithm to compute the greater common divisor (GCD) of two integers. An implementation of this algorithm is given in Fig. 7. Given two input integers, in u and v, this implementation computes their greater common divisor as the output value of v, by a subtraction-based variant of the Euclidean algorithm.

This program contains a lot of infeasible paths, all very similar. Please consider applying the infeasible path generalization to the program path $\pi = a^{\mathsf{t}}\,b^{\mathsf{t}}\,c\,d\,e\,f\,a^{\mathsf{f}}$. Path π takes one iteration of the loop and passes through the "then" branch of the conditional statement.

First, the method computes a suitable path condition and extracts an explanation. Table 3 gives a path condition of π computed for infeasible path generalization. On this particular path condition, the unique minimal explanation is obtained by removing only one constraint. The minimal explanation of the path condition of π is indeed the following set of constraints $K = \{[v_0 > u_0]_2, [t_0 = u_0]_3, [u_1 = v_0]_4, [v_1 = t_0]_5, [u_2 = v_1 - v_2]_6, [u_2 \leq 0]_7\}$.

Second, the method computes the infeasible chain $\langle I, D\rangle$ of (π, K). Fig. 8 gives a graphical representation of the infeasible chain. Explanation K corresponds to the set of path indexes $I = \{2, \ldots, 7\}$, indicated in the left part of the figure. The right part of the figure recalls the corresponding statements and tests. Path dependencies D

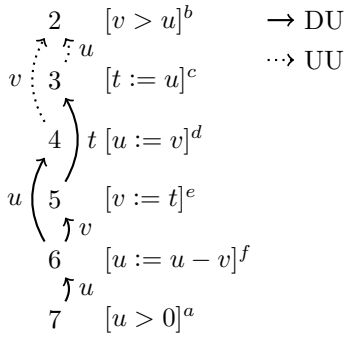

Figure 8: GCD: Infeasible Chain of (K, π)

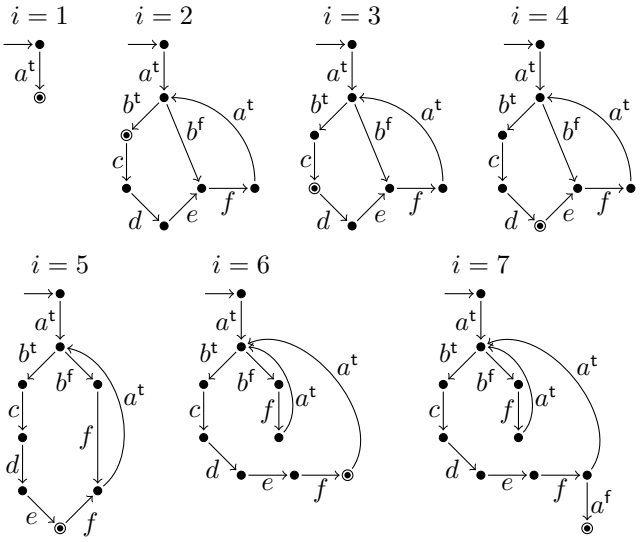

Figure 9: GCD: Construction steps of the Infeasible Path Automaton

are represented as arrows between the path indexes. Out of concern for precision, this figure distinguishes the two kinds of path dependency with different styles of stroke. This example particularly shows how tight the statements involved in the explanation can be bound together.

Finally, the method completes the infeasible chain in order to build an infeasible path automaton. Fig. 9 details the steps to build a deterministic automaton matching paths from the infeasible chain of (π, K) presented on Fig. 8. At $i = 1$, it starts with a single transition a^{t}. Then, for $i = 2$, it adds every path up to b^{t}, such paths can go through the loop an arbitrary number of times. It continues by adding 3 and 4, no alternative is available. Afterward it adds 5 and 7 for which there is also only one possible path. But these additions result in significant changes on the automaton in order to keep it deterministic. To finish, it adds a^{f}.

By using only one known infeasible path, this method is able to compute an infeasible path automaton that matches each and every infeasible path of this implementation of GCD.

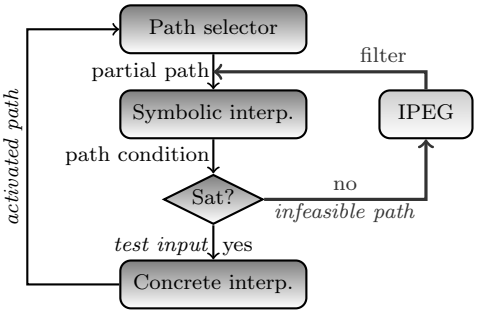

Figure 10: Dynamic Symbolic Execution with Infeasible Path Generalization

4. Integration to Dynamic Symbolic Execution

As noted in the introduction, infeasible paths is a hassle for Dynamic Symbolic Execution (DSE). In this section, we propose a basic yet realistic integration of the infeasible path generalization into a generic DSE implementation. By generic, we mean that this integration is independant of any specific DSE-based path selection strategy.

Generally speaking, any DSE-based process consists of four modules: 1) the *path selector*, 2) the *symbolic interpreter*, 3) the *constraint solver*, and 4) the *concrete interpreter*. Fig. 10 shows a typical DSE-tool architecture. It contains an additional module, called IPEG (Infeasible Path Explanation and Generalization), which corresponds to an implementation of the method presented in this paper.

1. The path selector module automatically selects a partial path (prefix) of the program, based on some inputs such as existing covered paths, on user data inputs or on various path selection strategies. Note that, typically, a path selector in DSE selects partial paths that are known to be feasible except for the last branch condition. This considerably reduces the number and length of the considered infeasible paths;
2. The symbolic interpreter computes the so-called *path conditions* for this partial path, meaning that it extracts a logical formulae from the program;
3. The constraints solver is called to evaluate the satisfiability of the formulae, and to produce a solution of the path condition. If the formulae is unsatisfiable ($SAT? = no$), then usually the process go back to the initial path selection process. On the contrary, if the formulae is satisfiable ($SAT? = yes$) and a solution is exhibited by the solver, then the process moves on the concrete interpretation;
4. The concrete interpreter converts the solution into a new test input that is submitted to the program. Executing the program with this new input produces a so-called *activated path*, which contains as a prefix the partial path selected by the first module.

Our proposed Infeasible Path Explanation and Generalization (IPEG) module is inserted in between item 3 and item 1 on the loop-back action. In Fig. 10, the IPEG

module is applied each time an unsatisfiable path condition is found ($SAT? = no$). The corresponding infeasible path represented by the unsatisfiable formulae, is analysed to extract a minimal explanation K. This minimal explanation is used to produce an infeasible path automaton, which is then combined with an internally maintained automaton. This combination is performed in the IPEG module by using the classical automata union operation. By enriching the current automaton with new infeasible path automata, the process increases its ability to discover infeasible paths. When a new partial path is selected by the path selector module, and just before being submitted to the symbolic executor, its feasibility is evaluated against the maintained infeasible path automaton. If the partial path is recognized by the automaton, then it is simply discarded and the path selector selects another partial path. By using the IPEG module, the process saves the testing of many unsatisfiable formula and avoids launching the constraint resolution to detect these unsatisfiable formula.

Optimization. Automata union is a classic operation which can be used to combine altogether infeasible path automata. However, it is noteworthy that a path is detected as infeasible as soon as one of its prefix is accepted by the automaton. As a consequence, one can *simplify* the maintained infeasible path automaton by pruning it from all outgoing transitions from final states. Any state which becomes unreacheable after this initial pruning can also be safely removed. As a safe optimization, the automaton only maintains shortest infeasible paths, so that its size (in terms of number of states and transitions) can be kept as low as possible.

4.1. Problems with Symbolic Expression Simplifications

As noted earlier, the symbolic execution must be selected with care. Sec. 3.1.1 prescribes that each instruction is to be translated into a constraint without any simplification (substitution, constant/expression propagation, unification, etc.). This is a limitation of the approach, because the size of the constraint system increases with the number of statements in the symbolically executed path. This section discusses this limitation and shows that infeasible path generalization is *still possible* with simplifications enabled, but that it reduces the power of generalization.

Let us explain the reason behind this recommendation. In this method, the explanation is able to point out the operations involved in the infeasibility because every operation is present in the path condition. However, it might not be the case if a simplification is done. For instance, on a path $a \ldots b^{\mathsf{t}}$, a statement a sets the variable i to the constant 10 and the test b read this value of i afterward, one can use directly the value 10 when translating b into constraints. If the test b is $i = 0$, the path element b^{t} translates to the inconsistent constraint $10 = 0$. This constraint forms a minimal explanation. This explanation corresponds only to the test b, which is not enough to obtain a sound generalization. In fact, one must also consider the statement a and the dependency between this statement a and the test b with respect the variable i as it occurred on the input path.

Note that it is possible to keep track of such a dependency. For instance, one can simply memorize for each variable both its value and the statements that the value depends on. Because the actual value propagated might not have been needed for the explanation, this technique leads to further losses of precision and consequently to a weaker generalization.

While the method and the symbolic execution is easily adapted to allow simplifications by taking into account such hidden dependencies, this adaptation is out of the scope of this paper. To summarize, infeasible path generalization works best with a symbolic execution without simplifications, but simplifications can be applied provided that hidden dependencies are tracked. Because simplifications may lead to a weaker generalization, it may be best to keep simplifications to a minimum.

4.2. Problems with Concrete Data

Another issue that may arise when integrating infeasible path generalization in DSE-based test generation, is related to the usage of concrete data. Dynamic symbolic execution does indeed allow to use data from previous concrete execution inside symbolic execution. Concrete values are mainly used to simplify expressions submitted to the constraint solver. In principle, using concrete data execution can compromise the soundness and completeness of the overall process as constraints can be over-simplified. In fact, constraint solving inconsistency may not only correspond to infeasible paths and may abusively recognize feasible paths (false positive). However, according to early observations, these cases are seldom in practice.

5. Experimental evaluation

This section presents the experimental results obtained to evaluate the infeasible path generalization process described above. The first subsection describes the prototype tool developed and our implementation choices. The second subsection presents the programs of our benchmark, while the third and fourth subsections give the experimental results. The fifth subsection introduces an additional experiment we made to evaluate the scalability of the approach. Finally, we conclude with a subsection dealing with the threats of validity of this experimental evaluation.

5.1. Prototype

The method presented in the paper has been implemented in a prototype tool called IPEG. The tool is parameterized by a constraint solver and an explanation method. In our experimental evaluation, we studied the impact of changing of constraint solver and explanation methods on the results.

Solvers

IPEG can use any constraint solver/decision procedure, provided that a proper interface is available. In our experiment we evaluated the three following solvers:

- *Colibri*, a propagation-based constraint solver developed at the CEA LIST, already used in the DSE-based test generation tools PathCrawler (Williams et al., 2005) for C, Osmose (Bardin and Herrmann, 2008) for binary code and GATeL (Marre and Arnould, 2000) for Lustre synchronous programs.

- *Yices*, an SMT solver developed at Stanford Research Institute (Dutertre and de Moura, 2006), notably used in the test input generator CREST (Burnim and Sen, 2008).

- *Z3*, an SMT solver developed at Microsoft Research (De Moura and Bjørner, 2008), used in testing tools, including Pex (Tillmann and de Halleux, 2008), a DSE-based test generation tool for .NET.

Methods for extracting an explanation

Similarly, IPEG allows the user to change of explanation extraction method. As described in Sec. 2.5, non-intrusive methods do exist to compute a minimal explanation, that is, a subset where each constraint contributes to the infeasibility. However, an acceptable trade-off between time and precision has to be found when extracting an explanation. Time issues can be managed by resorting to built-in intrusive explanation methods. This option leads to fast, but often non-minimal explanation. For propagation-based constraint solvers, it is possible to limit the explanation extraction to the propagation phase, that is, to prevent the costly labeling phase. Without labeling, such solvers cannot in general report solutions and may discern only few inconsistencies. Then, this approach is useful to find some, but not all, explanations. In the experiments, we evaluated three extraction methods:

- *Dichotomic method* (QuickXplain): a non-intrusive method, which extracts a minimal explanation by calling several times the solver (see Sec. 2.5) ;

- *Propagation-only* dichotomic method: the same method, but using a propagation-based constraint solver (Colibri) where the labeling is disabled ;

- Built-in *intrusive methods* of Yices and Z3.

Note also that the test generation process gives us the opportunity to know at least one constraint of the explanation (i.e., the last one). So, IPEG considers only the subset of constraints connected to this particular constraint, before extracting an explanation.

The considered subset of C

IPEG processes source code expressed in a meaningful subset of the C language. In fact, the main unsupported features are floating point numbers, unions, function pointers, and bitwise operations (including physical typecasting). Indeed, although infeasible path generalization can be adapted to these features, it requires that such complex features are accurately modeled into the form of constraints, which is a difficult task, that we considered being outside of the scope of this prototype.

Under some conditions, subroutine calls can be handled by inlining them. Note however that the def-clear path algorithm presented in Sec. 3.3 may not terminate. To prevent this issue, our implementation simply enforces every generalized path to strictly follow the same interprocedural control flow than the infeasible path used to feed the generalization process.

Dynamic Symbolic Execution module

The prototype IPEG also includes a dynamic symbolic execution module. It supports the same solvers than the infeasible path generalization module. Dynamic symbolic execution, as implemented in IPEG and most other tools, skips infeasible paths for which some prefix have been found infeasible. Indeed, IPEG embeds a depth-first prefix path selector for DSE. Initially, the path selector (see Sec. 4) selects the empty path. Then, given the last activated path, the path selector computes the next partial path by "flipping" the last branch of the path. If the selected partial path happens to be infeasible, the path selector simply "flips" the penultimate branch, and so and so forth. Except for a configurable limit on the selected path's length, this module do not provide optimizations for any specific coverage criterion.

Automata operations

The automata construction may use use deterministic automata operations, as well as non deterministic ones. A deterministic concatenation has a time complexity largely higher than a non deterministic concatenation. Indeed, a deterministic concatenation of two automata as an exponential state complexity, whereas the non deterministic concatenation is linear. However, as the worst case scenario seems unlikely in our case and deterministic finite state automaton are unequaled to match a string, the prototype IPEG uses deterministic automata operations.

Maywrite information

IPEG computes in advance maywrite information based on automated value analysis provided by Frama-C (Canet et al., 2009), a framework dedicated to the analysis of C source code.

5.2. Benchmark programs

- `gcd` computes the greater common divisor of two integers using Euclide's algorithm.

- `merge` merges two sorted arrays of 10 integers into a third of size 20.
- `bsearch` uses the bisection algorithm to pick a value in an ordered array of 20 integer elements.
- `selection_sort` applies the classical sort algorithm to an array of 10 integers.
- `tritype` gives the type of a triangle given the length of each side.
- `erfill` takes an array of 10 integers and two integers as inputs. First, it removes all occurrences of the first integer in the array by shifting to the left the remaining elements. Second, the free space is filled with the second integer.
- `tcas` consists of a small set of C functions that can be found in the Software-Artifact Infrastructure Repository (Do et al., 2005).
- `checkutf8` is a function that checks if a given array of bytes is a valid UTF-8 encoded character string, UTF-8 is an encoding for the Unicode character set.
- `git_config` is a program that parses configuration files in a particular format from the open-source distributed revision control system Git.

Except for array sizes, no precondition was imposed. In particular, input arrays of `merge` and `bsearch` could be non ordered.

For this evaluation, all the computations were performed on a Linux machine equipped with an Intel Core 2 Duo P9600 processor at 2.53GHz and 2GB RAM. Also, we used the latest release of Colibri (as of July 1st, 2010), Yices 1.0.28, and Z3 2.8. The benchmark programs' source code and detailed results are available online[2].

5.3. Unitary evaluation

First, we want to check the interest of our generalization method alone. This evaluation considers for each program a set of infeasible paths. For each of these paths, it applies the generalization. Then, it compares the time needed to generalize a path, to the time needed to prove by an exhaustive method based on symbolic execution that every generalized path is indeed infeasible.

5.3.1. Protocol

For each solver, program, and explanation method, we have computed the following data:

Input infeasible paths. We consider as *input infeasible paths* the shortest infeasible paths found by a breadth first search with at most 20 tests (branching condition). For each method, we allocate five seconds to the solver for proving infeasibility. The number of input infeasible paths is noted #IP.

Generalization. Each input infeasible path is generalized to an infeasible path automaton using IPEG. From the automaton, we extract the *generalized infeasible paths*, a set of infeasible paths containing at most 20 tests.

For each input infeasible path, we measure the *generalization time*, the *number of generalized infeasible paths*, and the explanation ratio, that is, the ratio between the number of constraints in the explanation and the number of constraints in the path condition. We average these measures over the input infeasible paths to obtain the *mean generalization time* (arithmetic), the *mean number of generalized infeasible paths* (arithmetic), the *mean explanation ratio* (harmonic).

Exhaustive method. Every *generalized infeasible path* has been proved infeasible using IPEG's symbolic execution. For each input infeasible path, the *exhaustive proof time* is the time needed to prove that every infeasible path generalized from this particular path (except the input infeasible path itself) is indeed infeasible. The *mean exhaustive proof time* is the arithmetic mean of the exhaustive proof time over all the input infeasible paths.

Speedup. Finally, we compute a speedup S for each infeasible path, defined as follows:

$$S = \frac{T_I + T_E}{T_I + T_G}$$

where T_I is the time needed to prove the input infeasible path infeasible, T_G the time needed for the generalization and T_E the time needed to prove that every path generalized are infeasible. We cannot compare T_G and T_E directly, because T_E can be null if the generalization does not infer any other path that the input infeasible path. Any value greater than one represent a gain in terms of performances. For instance, if for a path π, S is 2, the proof of the infeasibility of π and its generalization is two times faster than the exhaustive proof. The *mean speedup* is the geometric mean of these individual speedups.

5.3.2. Results

Table 4 shows these data measured on each program and each solver. The first two columns give the program and the solver (C for Colibri, Y for Yices, Z for Z3). The third column (E) indicates the explanation methods: a D for the dichotomic method, a P for the propagation-only variant of the dichotomic method and an I for the intrusive method built in the solver. The fourth column (#IP) contains the number of input infeasible paths considered for this configuration. Then, the four following columns give the mean explanation ration (ME%), the mean number of generalized infeasible paths (M#GP), the mean generalization time (MGT), and the mean exhaustive proof time (MET), both times expressed in milliseconds. Finally, the ninth column indicates the mean speedup (MS). Precise timing information was obtained through numerous repetitions.

[2] `http://micdel.fr/ipeg.en.html`

Table 4: Unitary Evaluation Results

Prog.	S	E	#IP	ME%	M#GP	MGT	MET	MS
tritype	C	D	21	22.43	0.0	3.5	0.0	0.25
		P	21	22.43	0.0	3.2	0.0	0.26
	Y	D	21	22.43	0.0	4.5	0.0	0.31
		I	21	20.59	0.0	2.9	0.0	0.40
	Z	D	21	22.43	0.0	3.9	0.0	0.32
		I	21	20.87	0.0	2.7	0.0	0.39
gcd	C	D	511	14.79	510.0	221.2	2021.1	10.67
		P	511	14.79	510.0	138.1	2021.1	16.21
	Y	D	511	14.79	510.0	17.0	2948.3	131.41
		I	511	14.79	510.0	7.2	2948.3	231.78
	Z	D	511	14.79	510.0	24.8	2542.5	86.84
		I	511	14.79	510.0	6.6	2542.5	223.43
bsearch	C	D	425	10.40	45.6	2710.6	24605.5	9.55
		P	425	11.56	35.1	3228.1	15145.7	5.90
	Y	D	425	10.39	45.6	355.8	1249.3	2.64
		I	425	11.53	37.4	27.9	869.5	5.06
	Z	D	425	10.39	45.6	116.7	1064.9	4.54
		I	425	11.96	35.1	24.8	668.0	5.63
merge	C	D	254	8.59	25.9	87.0	2521.8	12.31
		P	254	8.59	25.9	62.0	2521.8	14.49
	Y	D	254	8.59	25.9	42.6	348.4	5.69
		I	254	6.79	16.6	16.2	210.4	2.84
	Z	D	254	8.59	25.9	46.4	283.1	4.54
		I	254	5.68	29.3	13.4	312.3	12.04
selection	C	D	1024	22.59	340.3	145.6	103706.9	164.92
		P	1024	22.59	340.3	97.4	103706.9	188.00
	Y	D	1024	22.59	340.3	61.1	3830.7	38.19
		I	1024	22.59	340.3	14.2	3830.7	109.31
	Z	D	1024	22.59	340.3	79.8	3382.8	27.29
		I	1024	22.59	340.3	13.2	3382.8	105.94
tcas	C	D	288	5.37	11.5	10.5	37.8	2.63
		P	288	5.37	11.5	9.0	37.8	2.84
	Y	D	320	5.53	11.0	9.5	77.5	4.44
		I	320	5.73	11.1	9.2	77.7	4.46
	Z	D	320	5.53	11.0	9.6	62.4	4.02
		I	320	5.57	10.7	8.2	60.6	4.01
erfill	C	D	118	19.80	0.0	280.5	0.0	0.65
		P	118	19.80	0.0	244.8	0.0	0.71
	Y	D	118	19.80	0.0	80.2	0.0	0.22
		I	118	27.05	0.8	27.9	34.5	0.77
	Z	D	118	19.80	0.0	84.6	0.0	0.16
		I	118	19.80	0.0	19.3	0.0	0.47
checkutf8	C	D	227	7.34	3.7	16.7	23.0	1.02
		P	227	7.34	3.7	13.6	23.0	1.10
	Y	D	227	7.34	3.7	12.2	20.3	1.16
		I	227	8.22	4.9	8.8	27.4	1.63
	Z	D	227	7.34	3.7	12.8	18.1	1.08
		I	227	7.97	10.4	7.9	52.8	2.56
git_config	C	D	211	4.78	4.2	7.2	15.6	1.28
		P	211	4.78	4.2	6.7	15.6	1.30
	Y	D	211	4.78	4.2	7.7	22.4	1.56
		I	211	4.74	4.3	6.9	22.8	1.65
	Z	D	211	4.78	4.2	7.4	18.8	1.53
		I	211	4.74	4.2	6.4	18.9	1.52

Data shows that the infeasible path generalization is well compensated by the exhaustive proof, except for `erfill` and `tritype`. Indeed, with the different solvers and explanation methods, we observe good speedups (> 1) for most of the benchmark. For instance, the infeasible path generalization on the program `gcd` is on average from 10.67 up to 223.43 times faster than the exhaustive proof. This range can be explained by the strong differences separating the propagation-based solver Colibri to SMT solvers. Another result of this preliminary study is that searching a minimal explanation may be useless in general. Out-of-the-box intrusive methods sometimes find explanations of smaller cardinality (for instance, on `merge`).

Infeasible path generalization does not work well with two programs. First, `tritype` contains some infeasible paths, but they are very distinct, and any attempt of generalization ends up with a net loss. In contrast, some infeasible paths in `erfill` are infeasible for similar reasons but not identical reasons. Notably, a lot of these paths are infeasible because a loop counter has not increased to some value, giving rise to different explanations ($1 > 10$, $2 > 10$, etc.). Although such infeasible paths are present in programs, our technique does not generalize such paths efficiently. Also, `erfill` heavily uses arrays for which data dependencies greatly affect the generalization. In fact, with our prototype, the generalization was not able to generalize any new path (M#GP $= 0$), except with Yices/I. Indeed, another extraction method may find an unrelated explanation for the same infeasible path, and consequently it may lead to a better –or worse– generalization for some input paths. Although it may be interesting to find heuristics to obtain better explanation, one cannot ensure to find the best explanation to generalize except by investigating all minimal explanations, a task that will considerably hinder the performances of the generalization.

Limits of the unitary evaluation. The unitary experiment shows that the infeasible path generalization of a random infeasible path is in general more efficient that the exhaustive proof of the generalized paths. Still, a real use of this method might not consider any random infeasible path. In particular, for test generation, it appears natural to skip infeasible paths as soon as possible. But, this unitary experimentation does not take this into account. Indeed, input infeasible paths of this unitary study can – and sometimes do– belong to the same family of infeasible paths. That is, for a real use of the generalization, only one infeasible path will be necessary to generalize the family, while the computed speedup takes into account all the infeasible paths equally.

Fig. 11 illustrates by histograms the distribution of the speedup for each input infeasible path on four programs (`merge`, `tcas`, `bsearch`, and `git_config`) with Z3 and the dichotomic explanation extraction. These histograms show how the gain can vary in function of the input infeasible paths. For instance, the speedups on `merge` are

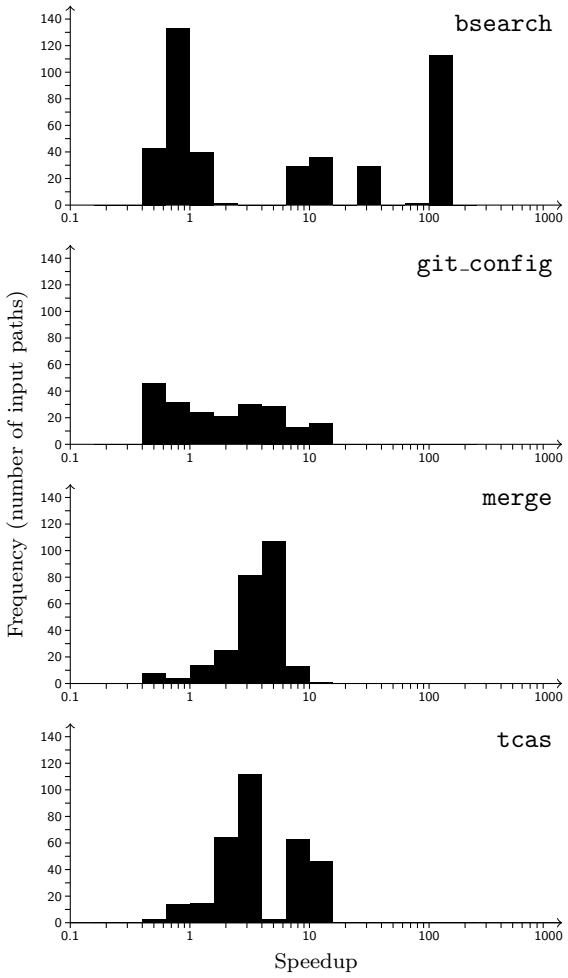

Figure 11: Histograms of speedup (Z3/D on four programs)

mostly greater than one and close together. In contrast, the speedup distribution on `bsearch` is scattered and reveals that the generalization does work well for an important group of infeasible paths (with speedup between 100 and 200) but does not work well with the other paths. In fact, `bsearch` contains a few big families of infeasible paths (> 20) and numerous small families (< 5). From a big family, the generalization computes infeasible path automata, which may differ slightly due to overapproximations, but gives in general very good speedups. As a result, big families weight on the average speedup. This shows a limit of the unitary experimentation and this motivates the need for another form of evaluation.

5.4. Evaluating the integration

This section is about verifying that the infeasible path generalization can be integrated in a dynamic symbolic execution test input generator. This evaluation is done with a dynamic symbolic execution module built in the prototype IPEG. This module, described in Sec. 5.1, provides a simple but realistic DSE. In particular, it skips any infeasible path for which some prefix have been found infeasible. Other DSE tools also embed heuristics to obtain a statement coverage while exploring less of the execution tree. In (Xie et al., 2009), Xie et al. draw up a list of state-of-the-art heuristics. IPEG does not propose such heuristics. Its goal is to demonstrate –in the worst case (the coverage of the execution tree)– the usefulness of infeasible path generalization in DSE, even though some infeasible paths are already skipped.

Fig. 12 sums up the results of our evaluation. It graphically gives the speedup for each program and variant (solver and explanation extraction method). The speedup is simply $\frac{T_1}{T_2}$ where T_1 is the time needed to generate tests without infeasible path generalization for every path of size below a certain limit and T_2 the time needed to generate the same tests with generalization enabled. The method is advantageous if the speedup is greater than one. Here, we pushed further the path size limit to explore a maximum of paths (full coverage reached on `tcas` and `tritype`) and stay with a test generation under four minutes (except with Colibri).

As predicted in the unitary evaluation, the infeasible path generalization does not benefit the test generation on `tritype` and `erfill`. But, it also confirms its good behavior on most of the programs. Results show for instance that the test generation of `gcd` and `git_config` with infeasible path generalization takes at least 30% less time than the test generation without it. Moreover, for `merge`, `selection` (except for Yices/I), and `tcas` the test generation is more than two times faster. However, this evaluation confirms our doubt about `bsearch`: infeasible path generalization indeed slows down the test generation (except for the intrusive variants). Still, as shown by the speedup histogram of `bsearch` on Fig. 11, our generalization is very beneficial for some paths. It seems interesting to find heuristics to apply our method with greater success.

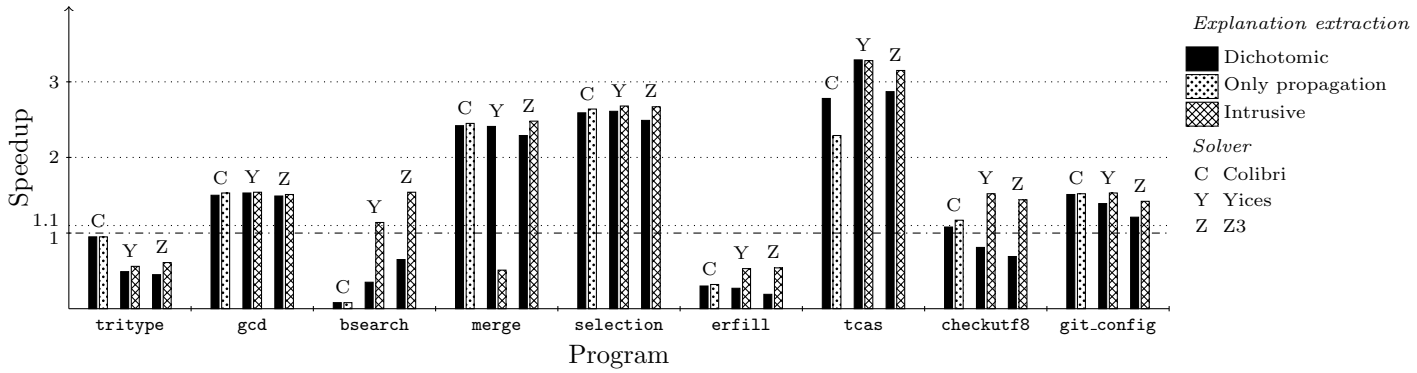

Figure 12: Test generation speedups

Table 5: Scaling up experiment results on `git_config2`

Optim.	#P	#S	ET	GT	UT	TT	S
DSE	-	-	-	-	-	236.4	1
+ General.	1695	11836	25.5	39.0	252.5	371.5	0.63
+ Intru.	1168	12363	11.1	20.6	200.2	295.4	0.80
+ Minim.	1168	12363	11.2	20.9	71.9	261.8	0.90
+ Heuris.	466	8638	4.6	8.2	17.2	155.2	1.52

Concerning explanations, if intrusive extractions bring worse results in few cases (e.g., on `merge` with Yices), its speed overcomes its possible imprecision in general. Also, for propagation-based constraint solver, the propagation-only variant of the dichotomic extraction seems precise enough for our purpose.

5.5. Scaling up the integration

In this section, we discuss experiments to apply our approach in more complex programs. Among our benchmark program, `git_config` is one of the more challenging because its consists of a relatively large number of functions, and nested loops which leads to longer paths. Here we are interested in applying our prototype in a second version of the program which parses a longer string of characters (10 instead of 5), as well as longer paths (from 20 to 40 branching conditions). We describe below multiple experiments and adjustment of the method in order to unlock real speedup from the test generation with infeasible path generalization.

Table 5 describes the results of our experiments. For each experiment, it provides detailed numbers: #P for the number of infeasible paths which were processed by infeasible path generalization, #S for the number of skipped infeasible paths, ET for the explanation time, GT for the generalization time (explanation excluded), UT for the time necessary to unify infeasible path automata, TT for the total generation time, as well as S for the speedup w.r.t. to the test generation without infeasible path generalization. Times are expressed in seconds.

The initial experiment consists in running a test generation with and without infeasible path generalization on `git_config2`. IPEG is used with a non-intrusive dichotomic explanation extraction, the solver Yices, and no other optimizations. We observe that infeasible path generalization slows down the test generation in this case. In particular, we observe that the union of automaton is very costly with longer paths.

In a second experiment, we choose to address the issue of the explanation time, which represents the biggest part of the generalization time. The previous experiments has shown that intrusive extraction does perform well with generalization. In addition to a lower explanation extraction time, we observe a better generalization. Indeed, the intrusive explanation explanation might lead to a different explanation that the first found by the dichotomic approach.

Our third experiment is aimed at reducing the time necessary to compute the union of infeasible generalization automata. A particular problem with deterministic automata is the explosion of their number of states which leads to always slower union. Longer paths in `git_config2` also contributes to this state exploration. To limit this phenomenon, IPEG was modified to minimize the automaton obtained at each individual path generalization and to minimize the union automaton every 20 generalizations. Brzozowski's algorithm was used to minimize the automata. We observe a sharp decrease in the computation time of the union. Another optimization which might largely decrease the number of states is to represent only branches in the infeasible path automaton rather that all statements. Heuristics may also be used to limit the union time such as reseting the global automaton regularly. Such optimizations and heuristics would have involved some major re-development in IPEG, but they seem very pertinent when considering bigger programs and longer paths.

Our fourth and final experiment is a response to the high number of generalizations. Indeed the number of generalization is too great with respect to the number of skipped paths. In fact, a lot of generalizations do not succeed in finding new infeasible paths or too few to cover the computation time. A simple but efficient heuristics is adopted. It consists to generalize only paths that ends with a "hot branch". A branch is considered hot if on only if n infeasi-

ble paths that ends with the branch have been considered by the test generation. In our experiments, the parameter n was 10.

To conclude, infeasible path generalization may be used with longer paths effectively. However, some points must be taken into consideration. First, as noted in Sec. 4, DSE test generator are often optimized and tuned for specific coverage criteria. One should take case to ensure the soundness of the generalization. Second, as pointed out by our experiments, although the automata union offers an elegant way to collect the knowledge of infeasible paths, it also collects knowledge about infeasible paths already considered and consequently not pertinent. In addition, it presents a real risk of state explosion. Those problems may be mitigated either through optimizations (such as minimization) or heuristics (such as resets). Second, all infeasible paths are not equal. Heuristics (such as the hot branch heuristics) are paramount to the approach in some cases.

5.6. Threats-to-validity

The three evaluation rounds we performed are complementary to show peculiar aspects of the infeasible path generalization process. However, they share common threats-to-validity.

As in any experimental study, there exists two major *threats to internal validity*: namely, the consequences of instrumenting the observed phenomenon and the measurement's precision. Here, the instrumentation is rather lightweight, as it consists only of adding incrementing counters and using interruptions to measure time. Even modest, the instrumentation overheads do exist. However, when comparing the approach proposed in the paper to the current existing approach, the instrumentation overheads penalize the proposed approach. In fact, the proposed approach requires to count paths and the number of generalization opportunities, as well as the multiple independent timers, while the existing approach requires only to count paths and contains only a single timer. Consequently, we consider that the instrumentation overheads can be disregarded. Regarding the precision of measurements, we observed significative differences between execution time of the same experiment. So, in order to mitigate the risk of reporting too-approximated results, we repeated all the experiments 20 times and reported only the average measurement. Additionaly, time variability was also greatly reduced by setting up a fixed CPU frequency, because modern CPU can adapt their own frequency depending on purpose.

The foremost *threat of external validity* is related to the choice of benchmark programs. Detailed in Sec. 5.2, our benchmark programs selection is composed h of programs of various origins and various (but moderated) sizes. Some of them are well-known benchmark from the software testing community (e.g., trityp, tcas), while some others come from the open-source community (e.g., git_congif). Noticing the moderated size of these programs, we believe that the results are currently generalizable to unit testing only. Even if some of the benchmark programs contains function calls, the number of such calls is too little to claim any result at the integration level or even system testing level. However, the methodology presented in the paper generalizes to functions with function calls, even if further experiments are needed to demonstrate the scalabilty of the approach to the case of integration testing. Note also that our implemented DSE-prototype is limited to simple search heuristics while exisiting powerful implementations provide optimized search and complex heuristics. It would be interesting to evaluate the infeasible path generalization process in these tools to reveal the true scalability of the approach. But, here again, this is part of further work.

6. Related Work

This paper presents an original method to generalize infeasible paths based on a single infeasible path. With respect to our earlier work (Delahaye et al., 2010), this method contains numerous improvements. First, the new method is completely formalized in terms of automata algebra, while the previous approach was based on graph manipulation. Second, the method allows for better generalization by allowing variable substitution in the explanation. Third, the prototype IPEG is also updated and automatically computes required static information on the program. Finally, IPEG is now capable to generate test inputs by dynamic symbolic execution, which allows us to evaluate our approach on test input generation.

To the best of our knowledge, there is no other use of explanations to generalize infeasible paths in the context of software testing. However, from a higher point of view, there are numerous approaches that combine static and dynamic knowledge of the code to improve performances of software verification.

For instance, in software model checking, given a program path to a particular target location, path slicing (Jhala and Majumdar, 2005) is an approach to find parts of the path that are relevant to reach a target location. This concept that works on a fixed program path but on free inputs is halfway between dynamic slicing (fixed path and test input) and static slicing (all paths and all inputs). Regarding infeasible paths, a path slice of an infeasible path is a subsequence of the path such that every path containing the subsequence is also infeasible. Although our method presents some common traits with path slicing, it is different because their goal is different. For instance, while path slicing starts from a unique target location, infeasible path generalization extracts a set of locations corresponding to an explanation. As a result, our method only uses data dependency between these locations, whereas path slicing considers all data dependencies that may affect the target. Another obvious difference is that a subsequence of the path (path slice) cannot capture as much infeasible paths as an infeasible path automaton.

That said, we think it is interesting to investigate the relationship between program slicing and path generalization. In the same manner that dynamic slicing was proved more close to static slicing than believed (Binkley et al., 2006), it might be possible to describe and define their relationship.

Dependencies have been used more directly in symbolic execution. Santelices and Harrold (2010) propose to consider the symbolic execution of path families instead of paths. In their work, a path family captures a set of paths that is determined to have the same behavior with respect to some output variable using control and data dependencies. Closely related to static slicing, the technique is based on static dependencies, and particularly over-approximated in the presence of loops. With change analysis in mind, rather than test generation, they propose to abstract loops and to overapproximate the path condition by trading precision for scalability. As noted earlier, infeasible path generalization is based on path dependencies rather than static dependencies. However, it might be interesting to extend our generalization to consider infeasible path families and under-constrained path conditions.

Also inspired by static analysis, RWSet (Boonstoppel et al., 2008) is a caching technique to reduce the number of explored paths in constraint-based test generation by discarding some paths with the same side-effects as some previously explored path. Even if it can discard some infeasible paths in the process, this technique is complementary to infeasible path generalization. In terms of performances, if the cost of maintaining such cache seems reasonable, identifying similar states involves constraint comparisons and is more costly than matching a path in a deterministic infeasible path automaton.

Caching may also be done at the constraint level. In (Visser et al., 2012), Visser et al. propose to exploit a constraint solution caching mechanisms to improve the performances of program analysis, and in particular symbolic execution. In the proposed system, the cost of the cache look-up is reduced using constraint canonization and slicing. This is for the most part complementary to infeasible path generalization. Concerning infeasible paths, the used path condition slicing offers very limited generalization because it only separates totally independent variables.

Infeasible paths have been pointed out as one of the major obstacles of test generation, notably in (Yates and Malevris, 1989). Yates and Malevris (1989) also show strong correlation between the number of branching conditions on a path and its potential infeasibility. They propose to generate tests for banch coverage based on the selection of the shortest paths of the program. In Papadakis and Malevris (2010), this path selection strategy has been extended to work on partial paths that reachs some branch to cover. In (Papadakis and Malevris, 2012), this technique is used to generate test cases that kills mutant efficiently by taking advantage of the equivalence between equivalent mutants (in weak mutation) and infeasible paths shown in (Offutt and Pan, 1996).

Offutt and Pan (1996) propose to detect equivalent mutants by detecting infeasible paths. The infeasible path detection is based on symbolic execution and by detecting a contradiction on the generated constraint systems. This technique syntactically detects contradiction given some known patterns of contradiction, designed with mutation operators in mind. It does not address the problem of loops in the program which was not handled correctly by their prototype.

Another related work that exploits constraint reasoning to speedup bug-oriented DSE-based test generation is (Jaffar et al., 2013). They propose to use Craig-interpolant to infer the feasibility of some path from the feasibility of a previously considered path. Interpolation is a concept closely related to explanation. By annotating the symbolic execution tree with interpolants, Jaffar et al. (2013) proposes to detect early a subtree that contains the same or more infeasible than a previously successfully explored subtree. This method allows the test generator to skip the whole subtree because the previous subtree did not meet any feasible bug. The two major differences between our two approaches are as follows. First, here no hypothesis on the objectives of test generation are made whereas (Jaffar et al., 2013) only targets finding bug. For instance, DSE generation with infeasible path generalization may be use to approximate worst-case execution time. Second, one important limitation of (Jaffar et al., 2013) is the subsumption check. It allows checking that the subtree does indeed entails the interpolant, whoever this checks assumes that variables are the same. It limits considerably the power of the technique in particular. In constrast, infeasible path generalization allows variable substitution.

Another work in software testing that tries to generalize infeasible paths is the work of Ngo and Tan (2007). They propose to detect the infeasibility of a path by statically recognizing four known code patterns leading to infeasible paths. For instance, their method detects infeasible paths where two conditional statements have the same condition (e.g., **if** $x > y$ **then** $\ldots$; **if** $x > y$ **then** $\ldots$). As the pattern recognition is sound, every detected path is indeed infeasible. We checked that our method does find all infeasible paths passing through the occurrence of one of those patterns, given an input path where the pattern occurrence is the only cause of infeasibility. In (Ngo and Tan, 2008), Ngo and Tan extend their approach to empirical properties instead of patterns to automatically detect non-feasible paths. Their approach relies on pairs of empirically correlated conditional statements, that is, conditional statements that reasons about the same variables and are not control dependent. Such statements are first shown to lead to many infeasible paths in programs. Then, the authors propose an integration into search-based test input generation, that ignores path when a branch violation occurs and the path contains a pair of empirically

correlated conditional statements. Our approach distinguishes from these approaches on two main points. First, our method dynamically finds patterns using known infeasible paths by computing explanations, whereas Ngo and Tan's methods try to recognize a few static patterns or a correlation between two statements. Both of their methods does not detect theoretically as much infeasibility as our method does, and the latter may lead to possible false positives. Second, unlike their approach, our method actually builds a representation of the infeasible paths by using automata operations and approximate data flow information.

7. Conclusion

This paper describes a new method to generalize infeasible paths from the detection of a single infeasible path and a way to exploit this infeasible path generalization technique in DSE-based automated test input generation. The method rests on three steps: First, it extracts one explanation of the infeasibility ; Second, it computes data dependencies associated to the proposed explanation; Third, it constructs an automaton which generalizes the feeded infeasible path and allows user to detect in advance other infeasible paths sharing the same explanation. The proposed method has been implemented and integrated into a generic DSE-based test input generation process. We selected a generic process in order to evaluate our method independently of any selection of constraint solver, or any explanantion method. The experimental results we got with this implementation show that, whatever the solver (i.e., Z3, Yices and Colibri), our infeasible path generalization method compares favourably with respect to an exhaustive infeasible path detection, and that it can speed up DSE when used for test input generation.

A notable possible improvement of the infeasible path generalisation method concerns the usage of semantics information about the program to obtain a larger generalization. By semantics information, we mean for example refining our data flow analysis with computated values. Additionaly, efficiently handling simplifications during DSE should also be investigated to limit the precision loss during the infeasible path generalization.

Regarding the perspectives of the proposed method, we believe that its usage could be beneficial to other applications than automatic test inputs generation. In particular, safe informations about infeasible paths usually release static analyzers from undesirable too large over-approximations obtained when computing some program properties. Our infeasible path generalisation method could efficiently capture and aggregate informations about infeasible paths. A challenge though would be to understand how our method can be used in practice, where most static analysis engines do not work incrementally. Additionaly, we also believe that security testing, when based on DSE, could benefit from incremental detection and generalisation of infeasible paths.

References

Bardin, S., Herrmann, P., 2008. Structural testing of executables, in: ICST'08, pp. 22–31.

Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., Korel, B., 2006. Theoretical foundations of dynamic program slicing. Theoretical Computer Science 360, 23–41.

Boonstoppel, P., Cadar, C., Engler, D., 2008. RWset: Attacking path explosion in constraint-based test generation, in: TACAS '08: 14th international conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer. pp. 351–366.

Burnim, J., Sen, K., 2008. Heuristics for scalable dynamic test generation, in: ASE'08: 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA. pp. 443–446.

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

Cimatti, A., Griggio, A., Sebastiani, R., 2007. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories, in: SAT'07, Springer. p. 334.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver, in: TACAS'08, pp. 337–340.

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

Dershowitz, N., Hanna, Z., Nadel, A., 2006. A scalable algorithm for minimal unsatisfiable core extraction, in: SAT'06, Springer. p. 36.

Do, H., Elbaum, S.G., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Eng. 10, 405–435.

Dutertre, B., de Moura, L., 2006. The Yices SMT solver. Tool paper available on line at http://yices.csl.sri.com/tool-paper.pdf.

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: directed automated random testing, in: PLDI'05: ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 213–223.

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

Jaffar, J., Murali, V., Navas, J.A., 2013. Boosting concolic testing via interpolation, in: ESEC/FSE'13: 9th Joint Meeting on Foundations of Software Engineering, ACM. pp. 48–58.

Jhala, R., Majumdar, R., 2005. Path slicing, in: PLDI'05: ACM SIGPLAN conference on Programming Language Design and Implementation, ACM. pp. 38–47.

Junker, U., 2004. QuickXplain: preferred explanations and relaxations for over-constrained problems, in: AAAI'04: 19th national conference on Artifical intelligence, AAAI Press. pp. 167–172.

Jussien, N., Debruyne, R., Boizumault, P., 2000. Maintaining arc-consistency within dynamic backtracking, in: CP'00, Springer. pp. 249–261.

King, J.C., 1975. A new approach to program testing, in: Proceedings of the international conference on Reliable software, ACM. pp. 228–233.

Kroening, D., Strichman, O., 2008. Decision Procedures: An Algorithmic Point of View. Springer Publishing Company, Incorporated.

Marre, B., Arnould, A., 2000. Test sequences generation from LUSTRE descriptions: GATEL, in: ASE'00, IEEE Computer Science.

Ngo, M.N., Tan, H.B.K., 2007. Detecting large number of infeasible paths through recognizing their patterns, in: ESEC-FSE'07, ACM. pp. 215–224.

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

Offutt, A.J., Pan, J., 1996. Detecting equivalent mutants and the feasible path problem, in: COMPASS'96: 11th Annual Conference on Computer Assurance, IEEE. pp. 224–236.

Papadakis, M., Malevris, N., 2010. A symbolic execution tool based on the elimination of infeasible paths, in: ICSEA'10: 5th International Conference on Software Engineering Advances, IEEE Computer Society. pp. 435–440.

Papadakis, M., Malevris, N., 2012. Mutation based test case generation via a path selection strategy. Information & Software Technology 54, 915–932.

Santelices, R., Harrold, M.J., 2010. Exploiting program dependencies for scalable multiple-path symbolic execution, in: ISSTA'10: 19th International Symposium on Software Testing and Analysis, ACM. pp. 195–206.

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

Tillmann, N., de Halleux, J., 2008. Pex: White box test generation for .NET, in: TAP'08, pp. 134–153.

Visser, W., Geldenhuys, J., Dwyer, M.B., 2012. Green: reducing, reusing and recycling constraints in program analysis, in: ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pp. 58:1–58:11.

Weyuker, E.J., 1979. The applicability of program schema results to program. Int. J. Parallel Program. 8, 387–403.

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

Xie, T., Tillmann, N., de Halleux, P., Schulte, W., 2009. Fitness-guided path exploration in dynamic symbolic execution, in: DSN'09: 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 359–368.

Yates, D., Malevris, N., 1989. Reducing the effects of infeasible paths in branch testing, in: TAV3, pp. 48–54.

Zhang, L., Malik, S., 2003. Extracting small unsatisfiable cores from unsatisfiable boolean formulas, in: SAT'03.