HAL
open science

Explanation-based generalization of infeasible path

Mickaël Delahaye, Bernard Botella, Arnaud Gotlieb

To cite this version:

Mickaël Delahaye, Bernard Botella, Arnaud Gotlieb. Explanation-based generalization of infeasible path. 3rd IEEE International Conference on Software Testing, Validation and Verification (ICST'10), Apr 2010, Paris, France. hal-00699240

HAL Id: hal-00699240

https://inria.hal.science/hal-00699240v1

Submitted on 21 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Explanation-based generalization of infeasible path

Mickaël Delahaye
CEA LIST
Software Safety Laboratory
Point courrier 94
91191 Gif-sur-Yvette, France
mickael.delahaye@cea.fr

Bernard Botella
CEA LIST
Software Safety Laboratory
Point courrier 94
91191 Gif-sur-Yvette, France
bernard.botella@cea.fr

Arnaud Gotlieb
INRIA Rennes
Project Celtique
35042 Rennes Cedex, France
arnaud.gotlieb@inria.fr

***Abstract*—Recent code-based test input generators based on *dynamic symbolic execution* increase path coverage by solving path condition with a constraint or an SMT solver. When the solver considers path condition produced from an infeasible path, it tries to show unsatisfiability, which is a useless time-consuming process. In this paper, we propose a new method that takes opportunity of the detection of a single infeasible path to generalize to a (possibly infinite) family of infeasible paths, which will not have to be considered in further path conditions solving. The method exploits non-intrusive constraint-based explanations, a technique developed in Constraint Programming to explain unsatisfiability. Experimental results obtained with our prototype tool IPEG show that, whatever is the underlying constraint solving procedure (IC, Colibri and the SMT solver Z3), this approach can save considerable computational time.**

***Index Terms*—Dynamic symbolic execution; constraint-based explanation; test input generation**

I. Introduction

Recent development in automatic test input generation have seen the emergence of *dynamic symbolic execution* tools such as DART [1], PathCrawler [2], CUTE [3] or Pex [4]. These tools dynamically select a feasible path by picking up a test input and observing which instructions are activated. Then, they report *path condition* by symbolically evaluating the instructions along this path. By refuting a decision (usually the last decision of path condition) and submitting the corresponding system to a constraint or SMT[1] solver, they try to infer another test input covering a distinct path in order to increase path coverage. Whenever the path is infeasible, the tools face the problem of proving unsatisfiability, which can be time-consuming. As the goal of dynamic symbolic execution tools is to find new test inputs instead of reporting path infeasibility, this process corresponds to a useless waste of time. One could argue that detecting all the infeasible paths of a program is impossible anyway[2] but studies have pointed out that although infeasible paths are ubiquitous in program [6], finding ways to avoid useless infeasible path exploration is highly desirable [7].

In this paper, we introduce a new technique that takes opportunity of the detection of a single infeasible path to generalize to a (possibly infinite) family of infeasible paths. The method exploits non-intrusive constraint-based explanations, a technique developed in Constraint Programming. The underlying idea consists first in finding a minimal set of constraints explaining the unsatisfiability and second in generalizing to all the infeasible paths whose infeasibility has the same explanation. This generalizing is based on finite automata operations and approximate data flow computations that are able to capture the "essence" of infeasible paths. As a simple introductory example, consider the C program and the infeasible path of Fig. 1. Explaining the unsatisfiability leads to determine that pair $(2 < x, x < 2)$ is a minimal (but not unique) explanation of this infeasibility. Then, by generalizing this constraint-based explanation with our method, we obtain the automaton of Fig. 2 that recognizes an infinite family of infeasible paths. Hence, any further path condition solving will benefit from this automaton to avoid trying to determine the feasibility of these paths. Our method is correct, meaning that it only generalizes to provable infeasible paths, but it is incomplete meaning that it usually cannot find all the infeasible paths. We implemented our approach in a tool called IPEG (*Infeasible Path Explanation and Generalization*), which can be parameterized by any satisfiability test. In our experiments, we used three constraint solving procedures that are used in symbolic-execution based test input generators: IC, Colibri and the SMT solver Z3[8]. We compared IPEG against an approach that would not benefit from the automatic detection of infeasible paths on several benchmark programs. These experiments show that our approach can save considerable computational time.

The rest of the paper is organized as follows. Section II presents the indispensable background on constraint-based explanation to understand our generalization method. Section III details the explanation and generalization algorithms of the method, while Section IV evaluates the method through experiments. Section V places this work into the state of the art. Finally, Section VI concludes the paper.

II. Background

This section first gives the notations and definitions about infeasible program paths and dependencies. Then, it introduces and reviews the notion of constraint-based explanations.

[1] Satisfiability Modulo Theory

[2] As this problem was proved undecidable in the general case [5]

```
1  int f2(int x,int y){
2    int a, res, i=2;
3    if (x >= 0)
4      a = x;
5    else
6      a = -x;
7    if (y)
8      res = 1;
9    else
10     res = -1;
11   while (i < a)
12     res *= i++;
13   if (x < 2)
14     res += 5;
15   return res;
16 }
```

Infeasible path

$1.2.3_t.4.7_t.8.11_t.12.$
$11_t.12.11_f.13_t$

Corresponding constraints

Origin	Constraint
3_t	$x \geq 0$
7_t	$y \neq 0$
11_t (1)	$2 < x$
11_t (2)	$3 < x$
11_f (3)	$4 \geq x$
13_t	$x < 2$

Fig. 1: An example function and an infeasible path

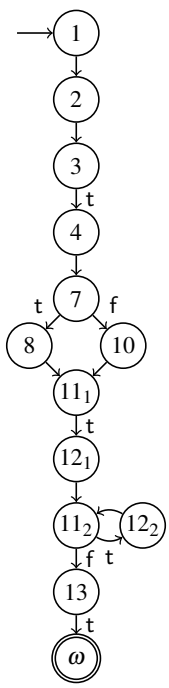

Fig. 2: An infeasible path acceptor

A. Notations and definitions

A *program path* π is a sequence of successive statements $(\pi_i)_{i\in\{1,\ldots,n\}}$ that can flow throughout the program. For simplicity's sake, we suppose each element of the sequence is unique by numbering multiple occurrences of the same statement when necessary. $\mathsf{node}(\pi_i)$ gives the program statement at π_i without numbering. We also explicitly write for each branching statement the decision (t or f). $\mathsf{decision}(\pi_i)$ indicates the decision taken at π_i if appropriate, ε otherwise. At a program statement n, $\mathsf{maywrite}(n)$ denotes the set of variables that may be written in n. This set is usually an over-approximation of the variable actually written in n as it is statically computed on all the program paths that led to n. A program path π is said to be *infeasible* if there is no program input able to execute the statements along π. Conversely, π is *feasible* if there is an input that can drive the computation along π .

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

We introduce here the notion of *consistency check* which is a predicate that can check the consistency, or satisfiability, of a given constraint system. It is noted $\Gamma(C)$ for a constraint system C, and it can return either **true**, **false** or **inconclusive**. If $\Gamma(C) = \textbf{true}$, C is consistent, or satisfiable. If $\Gamma(C) = \textbf{false}$, C is inconsistent, or unsatisfiable. The latter case, $\Gamma(C) = \textbf{inconclusive}$, occurs when the underlying computation theory is undecidable (e.g., non-linear integer arithmetic) or when the consistency check takes too long time and has to be interrupted. For shortness' sake, in the paper, Γ-*consistent* (resp. Γ-*inconsistent*) means consistent (resp. inconsistent) according to the consistency check Γ.

The *path condition* of a program path π is a conjunction of constraints on the input symbolic variables of the program that characterizes π's execution. Let us note C_π the path condition associated to program path π. Given a consistency check Γ, we have the following property: if C_π is Γ-inconsistent, then π is infeasible and conversely if C_π is Γ-consistent then π is feasible. But, there is no equivalence as the consistency check Γ can be partial.

B. Explanations

An important part in the method presented here 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*, or *irreducible*, if no subsystem of the explanation is also an explanation, that is, if all parts of the explanation are essential to the inconsistency.

Formally, given a partial consistency check Γ, a Γ-*minimal explanation* K of a constraint set C is a subset of C such that, for all constraint c of K, $\Gamma(K \setminus \{c\})$ is *true* or *inconclusive*. Consequently, a Γ-minimal explanation is not necessarily minimal when Γ is partial. Also, two distinct consistency checks can lead to distinct Γ-minimal explanations.

There is 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.

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*, or *elimination explanation*, 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. [9], adapted the recording of such elimination explanation 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 $x \neq y \neq z \neq t$ for x, y, z, and t in $\{1,2\}$. Its inconsistency can be proved by the solver with the following case-based reasoning:

- $x = 1 \overset{x\neq y}{\Rightarrow} y = 2 \overset{y\neq z}{\Rightarrow} z = 1 \overset{x\neq z}{\Rightarrow} \bot$
- $x = 2 \overset{x\neq y}{\Rightarrow} y = 1 \overset{y\neq z}{\Rightarrow} z = 2 \overset{x\neq z}{\Rightarrow} \bot$

As the domain of x has been entirely explored, we get an explanation of the inconsistency of the overall constraint system. Note however that explanations obtained this way will not usually be minimal, or even Γ-minimal, because an automatic reasoning can be terribly convoluted.

2) Non-intrusive methods: The main idea of non-intrusive methods 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 $x \neq y \neq z \neq t$, for x, y, z, and t in $\{1,2\}$, can be found by successively checking the consistency of its subsystems: $x \neq y$, $x \neq y \wedge x \neq z$, etc. Here $x,y,z \in \{1,2\}, x \neq y \neq z$ is a minimal explanation. Note that this explanation is not unique as one could replace z by t and obtain another minimal explanation.

Several non-intrusive algorithms have been proposed but the recursive dichotomic Junker's algorithm [10] is considered to be the most efficient as it runs in $O(n \log(k+1) + k^2)$ in the worst case instead of $O(nk)$ where is n is the size of the constraint system and k the size of the explanation to be found.

For a partial consistency check Γ, Junker's algorithm tries to find a Γ-minimal explanation to an input constraint set. At each recursion step, the algorithm tries to identify one constraint of the explanation in a given partition, initially the input constraint set. If it is possible, the other constraints that may be involved in the inconsitency are partitioned into two groups and fed to the same algorithm. If it is not, all the constraints in the input partition are skipped. Table I presents a run of Junker's algorithm on an hypothetic inconsistent constraint system $\{c_1, c_2, c_3, c_4, c_5, c_7, c_8\}$ with one Γ-minimal explanation $\{c_4, c_7, c_8\}$. In this table, **not false** actually means **true** or **inconclusive**.

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 have a small explanation. Note also that Junker's algorithm does not take into account the input system is indeed inconsistent. If it can be appreciable for the outer call, it causes some superfluous tests in the inner recursive calls (e.g., Steps 12 and 20 of Table I).

TABLE I: A run of Junker's algorithm

Step	Constraints	Γ	Explanation
1	c_1	**not false**	$\varnothing$
2	c_1 c_2	**not false**	$\varnothing$
	...		
8	c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8	**false**	$\{c_8\}$
9	c_1 c_2 c_3 c_4 c_8	**not false**	$\{c_8\}$
10	c_1 c_2 c_3 c_4 c_8 c_5	**not false**	$\{c_8\}$
	...		
12	c_1 c_2 c_3 c_4 c_8 c_5 c_6 c_7	**false**	$\{c_7, c_8\}$
13	c_1 c_2 c_3 c_4 c_8 c_5 c_7	**false**	$\{c_7, c_8\}$
14	c_1 c_2 c_3 c_4 c_8 c_7	**false**	$\{c_7, c_8\}$
15	c_7	**not false**	$\{c_7, c_8\}$
16	c_7 c_8	**not false**	$\{c_7, c_8\}$
17	c_7 c_8 c_1	**not false**	$\{c_7, c_8\}$
	...		
20	c_7 c_8 c_1 c_2 c_3 c_4	**false**	$\{c_4, c_7, c_8\}$
21	c_7 c_8 c_1 c_2 c_4	**false**	$\{c_4, c_7, c_8\}$
22	c_7 c_8 c_4	**false**	$\{c_4, c_7, c_8\}$

In this paper, we chose to explore a non-intrusive method to compute Γ-minimal explanations in order to be as independent as possible of the constraint or SMT solver. Indeed, intrusive methods usually require intimate knowledge of and heavy modifications of the selected underlying constraint solver.

III. Method

This section details a complete method of infeasible path generalization. The inputs of the method consists of:

- an imperative program under test,
- a consistency check Γ,
- an infeasible path π,
- and the corresponding Γ-inconsistent path condition C_π.

In addition, the method uses approximate dataflow informations with the set $\mathsf{maywrite}(n)$. Note that computing a tight over-approximate set $\mathsf{maywrite}(n)$ may be hard in the presence of arrays and pointers.

Infeasible path generalization tends to compute a set of infeasible paths defined as follows:

Definition 1. Given an explanation K of C_π, the *K-general infeasible path set*, noted I_K, is the set of every program path p such that their path condition C_p includes K.

Our method approaches this goal without requiring any additional symbolic execution by constructing a deterministic finite automaton A accepting only paths p such that the path condition C_p contains K, i.e., $\mathscr{L}(A) \subset I_K$.

Fig. 3 gives the general sketch of the method. It may be split into two steps:

1) Explaining the infeasibility by an Γ-minimal explanation
2) Determining a family of paths whose path condition contains this explanation

function generalize(π, C_π)
1 $K \leftarrow$ explain($C\pi, \varnothing$);
2 $N \leftarrow$ onpath$_\pi(K)$;
3 $D \leftarrow$ pathdependencies(π);
4 **return** extendpath(π, N, D);

Fig. 3: Method algorithm

Input: C, A two constraint sets such as $C \cup A$ is Γ-inconsistent
Output: a minimal subset X of C such that $X \cup A$ is Γ-inconsistent

function explain($C = \{c_1, \ldots, c_n\}, A$)
$i \leftarrow 0$; $b \leftarrow$ **true**;
while $b \wedge i < n-1$ **do**
  $i \leftarrow i+1$;
  $b \leftarrow \Gamma(A \cup \{c_1, \ldots, c_i\}) \neq$ **false**;
if $\neg b \wedge i = n-1$ **then**
  $i \leftarrow i+1$; *// Skips redondant check*
$X \leftarrow \{c_i\}$;
$m \leftarrow \lfloor (i-1)/2 \rfloor + 1$;
$U \leftarrow \{c_1, \ldots, c_{m-1}\}$; $V \leftarrow \{c_m, \ldots, c_{i-1}\}$;
if $V \neq \varnothing \wedge \Gamma(A \cup U \cup X) \neq$ **false then**
  $X \leftarrow X \cup$ explain($V, A \cup U \cup X$);
if $U \neq \varnothing \wedge \Gamma(A \cup X) \neq$ **false then**
  $X \leftarrow X \cup$ explain($U, A \cup X$);
return X;

Fig. 4: Dichotomic extraction of explanation

A. Explaining the infeasibility

The first phase consists in the line 1 of Fig. 3. The idea is that the smaller the explanation the more the chances are to find infeasible paths with the same constraints.

We propose here to use a variation of the dichotomic method of Junker [10] to find a Γ-minimal explanation. Fig. 4 gives the pseudocode of the function explain(C, A). This function takes two constraint sets, C and A, as inputs, such that $C \cup A$ is Γ-inconsistent. This function computes a minimal subset X of C such that $X \cup A$ is Γ-inconsistent. That is why explain($C_\pi, \varnothing$) returns a Γ-minimal explanation of the Γ-inconsistent constraint system C_π.

At each step, the method determines an additional constraint of the explanation. It checks iteratively the Γ-consistency of the union of the already detected constraints A and a growing subset C' of C. Consequently, when the consistency check fails, the last added constraint is a constraint of the Γ-minimal explanation of $C \cup A$. Then, the remaining constraints of C are discarded and the constraints added to C' except the last one, are split in two groups U and V. Finally, the method calls itself recursively on the two groups. The returned subset contains the last added constraint and the return values of the recursive calls.

Unlike Junker's, this algorithm also uses the fact $C \cup A$ is indeed inconsistent to skip redondant consistency checks.

In the example of Fig. 1, given a consistency check Γ, two Γ-minimal explanations can be found: ($2 < x \wedge x < 2$ and $3 < x \wedge x < 2$). By considering the constraints in the order they are computed along the initial path, the method selects the former Γ-minimal explanation: $2 < x \wedge x < 2$.

B. Determining the family

As each constraint comes from a given program statement, it is quite natural to translate the explanation expressed in terms of constraints to an expression in terms of statements.

However, it is not enough to characterize infeasible paths. Indeed, there may exist feasible paths that pass through all the statements corresponding to some explanation's constraints. Indeed, in two different paths the same instruction can be interpreted into two different ways. In the example of Fig. 1, the computed explanation $2 < x \wedge x < 2$ corresponds to the control points $11_\mathtt{t}$ and $13_\mathtt{t}$. Although the path $1.2.3_\mathtt{f}.6.7_\mathtt{t}.8.11_\mathtt{t}.12.11_\mathtt{f}.13_\mathtt{t}$ shared the same control points, it is not infeasible. For example, $x = -3$ activates this path. Indeed, the first of these two control points is interpreted in another way ($2 < -x$ instead of $2 < x$).

Translating to instructions: Function onpath$_\pi$, used at line 2 in Fig. 3, keeps track of the association between constraints and statements during the symbolic execution. This function computes the actual set N of path statements corresponding to the explanation K.

Ensuring the explanation: One way to ensure that the constraints in the explanation are actually posted "as is" is to ensure the read values at the selected statements originates from the same assignments as in the input paths. We introduce the following definition to keep track of this kind of dependencies.

Definition 2. On a program path π, a *path flow dependency* $\pi_i \xrightarrow{v} \pi_j$ occurs between two statements π_i and π_j for the variable v if:

- $i > j$,
- π_i reads v,
- π_j writes v,
- and there is no statement π_k for $i > k > j$ such that π_k writes v; in other words, the subpath $(\pi_{j+1}, .., \pi_{i-1})$ is a def-clear path for variable v.

This definition is special in two ways: first, the dependency is considered on a single path (that may be infeasible) and second, it is defined in the reverse order of the execution. This will simplify the presentation of the generalization algorithm. As the set of variables read or written at a given statement cannot be known exactly in the presence of indirection (arrays, pointers), the algorithm of Fig. 3 at line 3 only computes an (over-)approximation of the path flow dependencies D on π.

In the example of Fig. 1, the path statements 11_1 and 13 can be identified as the origin of the selected explanation. Fig. 5 gives a graphical representation of the path flow dependencies related to these statements.

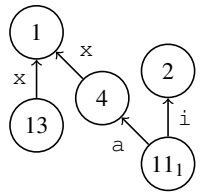

Fig. 5: Path flow dependencies linked to the path statements 11_1 and 13

Extending the possibilities around the path: After translating the explanation to a set N of path statements and finding the path flow dependencies, the method finds a (possibly infinite) family of paths containing N and enforcing the same path flow dependencies that the ones observed in the original infeasible path. The following definition helps clarifying this notion.

Definition 3. Given S the subset of path flow dependencies linked to N, a (K,D)-*general infeasible path* is a program path that contains every program statements of N and S in the same order than π and such that every subpath between these statements is def-clear w.r.t. the path flow dependencies in S.

This definition enforces that the statements N are actually interpreted exactly as they were in π. Consequently, (K,D)-general infeasible paths are guaranteed to include the explanation K in their path condition and to be infeasible.

Fig. 6 gives an algorithm to construct an automaton accepting (K,D)-general infeasible paths. This algorithm starts from the infeasible path π and adds all the possible variations that cannot affect the set of path statements N corresponding to the explanation. This algorithm uses the path dependencies D to identify these variations.

It maintains the last seen indispensable elements k, a set L of active path flow dependencies, i.e., a pair of a variable and an index in the path prefix, and δ a set of transitions. A path statement is indispensable if it belongs to N or is directly or indirectly linked to a path statement of N in the path dependency graph D. The active path dependencies L allow the algorithm to efficiently identify indispensable statements and variables whose values are important to the explanation.

Initially, k indicates the last element of the path prefix contained in N, L gives the path dependencies of this element and δ contains one transition from π_k to the final state ω.

Then, it considers each element of the path prefix from the next to last:

- First, adds a transition to δ corresponding to the transition observed on the path from the previous element to this element.
- Then, if the element is indispensable (that is, in N or in L for any variable), keeps it and update k and L accordingly. Otherwise, adds transitions corresponding to the alternative definition-clear paths between this node and the node at k w.r.t. the variables L, by calling extension.

Input: π an infeasible path, N the projection of the unsatisfiable core on π and D the path dependencies observed on π
Output: an infeasible path automaton $\langle \delta, i, f \rangle$ where δ is the transition set, i the initial state and f the unique accepting state

function extendpath(π, N, D)
$m \leftarrow \max\{i | \pi_i \in N\}$;
$\delta \leftarrow \{\pi_m \xrightarrow{\text{decision}(\pi_m)} \omega\}$;
$L \leftarrow \{(x,j) | (\pi_m \xrightarrow{x} \pi_j) \in D\}$;
$k \leftarrow m$;
for $i = m-1$ **à** 1 **do**
  $d \leftarrow \text{decision}(\pi_i)$;
  $\delta \leftarrow \delta \cup \{\pi_i \xrightarrow{d} \pi_{i+1}\}$;
  if $\pi_i \in N \vee \exists x, (x, \pi_i) \in L$ **then**
    $L \leftarrow \{(x,j) \in L | j < i\} \cup \{(x,j) | (\pi_i \xrightarrow{x} \pi_j) \in D\}$;
    $k \leftarrow i$;
  else if $d \neq \varepsilon$ **then**
    let n be the statement such that $\text{node}(\pi_i) \xrightarrow{\neg d} n \in$ **Program**,
    $A \leftarrow \text{extension}(n, \text{node}(\pi_k), \{x | \exists j, (x,j) \in L\})$;
    if $\mathscr{L}(A) \neq \varnothing$ **then**
      $\langle \delta', s, d \rangle \leftarrow \text{safeRenaming}(A)$;
      $\delta \leftarrow \delta \cup \delta' \cup \{\pi_i \xrightarrow{\neg d} s, d \rightarrow \pi_k\}$;
return $\langle \delta, \pi_1, \omega \rangle$;

Fig. 6: extendpath algorithm

Input: s and d two program statements 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 extension(s, d, V)
/* *Considering the program as a transition set,* $\langle$**Program**$, s, d\rangle$ *is an automaton accepting all program paths from s to d* */
$\delta \leftarrow$ **Program**;
/* *Removes statements writing any variable of V* */
foreach statement n **do**
  if $\text{maywrite}(n) \cap V \neq \varnothing$ **then**
    $\delta \leftarrow \delta \setminus \{a \xrightarrow{d} b \in \delta | a = n \vee b = n\}$;
/* *Keeps only accessible and coaccessible states and transitions* */
return $\text{trim}(\langle \delta, s, d \rangle)$

Fig. 7: extension algorithm

As the state names used in the automata returned by extension might collide with names already used, this algorithm uses a function safeRenaming to rename the states of the returned automata.

Computing the extensions: Function extension(s,d,V) computes an automaton recognizing a safe under-approximation of the paths between the source s and the destination d that do not modify any variable of the variable set V (i.e., def-clear paths). Fig. 7 gives a simple algorithm for extension(s,d,V). First, it considers the program as an automaton from s to d and removes all statements which may write V from this automaton. Second, it trims the automaton for efficency, keeping only useful states (e.g., accessible and coaccessible).

Fig. 8 shows the automaton generated for the infeasible path of Fig. 1. As shown on Fig. 8 in dotted stroke, the path has been extended from 11_3 to 13, adding any number of loop

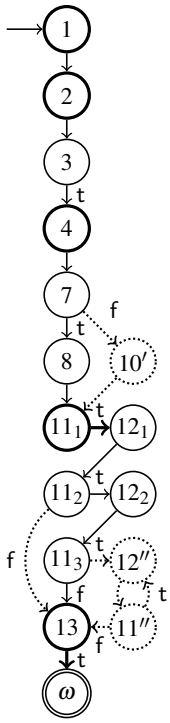

Fig. 8: A generated automaton

iterations, from 11_2 to 13, allowing a possible shortcuts and from 7 to 11_1, adding the other branch of the conditional construct. The earlier introduced automaton of Fig. 2 can easily be obtained by minimizing the automaton of Fig. 8.

The recognized infeasible path families can also be written as a regular expression. For instance, the automaton of Fig. 2 can be translated to the following regular expression:

$$1.2.3_{\mathsf{t}}.4.(7_{\mathsf{t}}.8|7_{\mathsf{f}}.10).11_{\mathsf{t}}.12.(11_{\mathsf{t}}.12)^{\star}.11_{\mathsf{f}}.13_{\mathsf{t}}.$$

C. Correctness and completness

This infeasible path generalization method is correct, that is, it finds only infeasible paths. Indeed, one can prove the following theorem:

Theorem (Correctness)**.** *For any path infeasible π (according to Γ), the method* $\mathsf{generalize}(\pi, C_\pi)$ *returns an automaton accepting only infeasible paths.*

Sketch of proof: After proving $\mathsf{explain}(C, \varnothing)$ computes an explanation of C (see [10]), it is equivalent to say that $\mathsf{extendpath}(\pi, N, D)$ returns an automaton accepting only K-general infeasible paths. First, $\mathsf{extension}(s, d, V)$ computes an automaton accepting only def-clear paths for all s, d and V, because the algorithm explicity removes statements that may modify any variable of V. Then, we can show $\mathsf{extendpath}(\pi, N, D)$ returns an automaton accepting only (K, D)-general infeasible path. Finally, it remains to formally prove (K, D)-general infeasible path are indeed K-general infeasible. ∎

However, the method presented here is not complete. Some infeasible paths may be not generalized from π, for at least the following reasons:

- It considers only one Γ-minimal explanation.
- Other statements than N may results in the same constraints.
- Syntax-based dependencies are overprotective. For instance, on two distinct paths, the execution may compute differently the same value for a statement involved in the explanation K.

IV. Experimental evaluation

A. Implementation

We implemented our method in a prototype tool called IPEG. The tool can be parameterized by any constraint or SMT solving procedure and we evaluated three distincts constraint solvers:

- **IC**, a constraint library provided by the ECLiPSe Prolog environment. IC implements constraint propagation and labelling over finite domains for solving integer constraint systems. This library is used in the tool TestGen [11] which generates test data for ADA programs using symbolic execution.
- **Colibri**, a home-made constraint solver developped at the CEA which is tuned to address dynamic symbolic execution requests from at least three test data generation tools: PathCrawler [2], OSMOSE [12] and GATEL [13].
- **Z3**, the SMT solver developped at Microsoft Research [8] and used in the Pex automatic test data generator for .NET [4].

As these solvers implement three distincts constraint solving procedures, the notion of *consistency check* used to compute explanation slightly differs:

- For **IC**, the test is based on interval-based consistency checking meaning that constraint propagation just considers the bounds of variation domains to establish partial consistency.
- For **Colibri**, the test is also based on interval-based consistency checking apart the fact that the solver maintains an internal structure of bounded unions of intervals. In addition, the solver implements two add-ons dedicated to consistency checking based on a closure test for distance constraints and consistent congruential properties.
- For **Z3**, the test is based on a combination of techniques from SAT-solving, closure algorithms for binary constraints, simplex-based consistency check and so on.

Our prototype IPEG handles a meaningful subset of the C language, including integers, arrays, structures, control statements (conditionals, loops, switch, continue, break and goto) and most C operators, but it has also some restrictions as it currently does not support function calls, pointer variables, floating-point computations, bit-wise operators and integer overflows.

B. Benchmarks

To evaluate our method, we applied it on about two thousand paths from eight small-sized C programs extracted from

classical software testing benchmarks. The function `erfill` takes an array and two integers as inputs and implements two related operations. First all occurences of the first integer input in the array is removed and second the free space is filled with the second integer. The program `tcas` is a small set of C functions that are part of the Traffic Alert and Collision Avoidance System embedded on commercial aircrafts. This program can be found in the Software-Artifact Infrastructure Repository [14]. The function `merge` takes two arrays of 5 integers as inputs and puts them in a third array of 10 integers. If both input arrays are sorted, the resulting array should be sorted as well. The function `selection` sorts an array of 5 integers using a classical selection algorithm. The function `tritype` gives the type of a triangle given the length of each side. The function `gcd` computes the greater common divisor of two integers. Finally, functions `f1` and `f2` are home-made examples based on an iterative factorial algorithm. The source code of some of these functions is given in appendix. Also, the maywrite sets were manually provided for each function.

Input infeasible paths: Our goal is to apply our method to every infeasible path that an exhaustive test input generation tool based on dynamic symbolic execution would stumble upon. We used a dedicated module of IPEG to incrementally search the program's execution tree for infeasible paths using the selected constraint solver. As these solvers implement partial consistency checks, the detected infeasible paths may vary from one solver to another. Because the execution tree may be infinite, and like most path-oriented test input generator, a generic limit is put on the number of instructions of the considered paths. For this experiment, the limit was 50 instructions, that is, no infeasible path containing more than 50 instructions was considered.

Comparison method: We compared our generalization method with an approach which would have proved the infeasibility of each path of the automaton. Let us call this method: *the exhaustive method*. To be fair, we used the same constraint solver in both cases. Also, as most dynamic symbolic execution tools (e.g., PathCrawler) adopt an incremental path verification, this exhaustive method also verifies path infeasibility incrementally. Consequently, if multiple paths of the automaton can be proved infeasible by a single path prefix, only the time needed to prove the infeasibibility of the prefix will be counted and that only once.

The experimental protocol was as follows. First, we seeded IPEG with the input infeasible paths. For each of them, an automaton is built that recognizes generalized infeasible paths. The time T_{gen} needed to generate the automaton was recorded. Second, for each automaton, we used the exhaustive method to prove the infeasibility of all paths recognized by the automaton that contains 50 instructions or less. In each case, the total time T_{exh} needed to prove their inconsistency was recorded. Although our automaton may accept infeasible paths of more than 50 instructions and in some case of any length, we chose to put this limit to stay fair with other methods and to be consistent with the considered input paths.

TABLE II: Path results on the example programs

Program	Solver	Input paths	Generalized paths *Average*	Generalized paths *Maximum*
`erfill`	Colibri	211	**15**	148
	IC	211	**15**	148
	Z3	211	**15**	148
`f1`	Colibri	440	**61**	240
	IC	440	**61**	240
	Z3	440	**61**	240
`f2`	Colibri	58	**16**	30
	IC	58	**16**	30
	Z3	58	**16**	30
`gcd`	Colibri	4	**2**	2
	IC	4	**2**	2
	Z3	4	**2**	2
`merge`	Colibri	504	**43**	121
	IC	504	**43**	121
	Z3	504	**43**	121
`selection`	Colibri	460	**157**	385
	IC	–	**–**	–
	Z3	460	**157**	385
`tcas`	Colibri	692	**230**	550
	IC	–	**–**	–
	Z3	532	**168**	446
`tritype`	Colibri	15	**2**	5
	IC	22	**2**	3
	Z3	15	**2**	5

All the computations were performed on a Linux machine equipped with an Intel Core 2 Duo P9600 processor at 2.53GHz and 2Go RAM. We used Z3 2.0, IC as provided with ECLiPSe 5.10, and the latest version of Colibri as of October 1st, 2009.

Results: Table II presents the results of our comparison in terms of number of paths explored during the experiment. For each program, the table reports the number of input infeasible paths seeded and the number of paths which have been generalized, i.e., paths from an automaton, in terms of average and maximum over the overall set of input paths.

Table III reports the results of our comparison in terms of CPU time and memory usage. The first two columns report the average CPU time required by IPEG to generalize infeasible paths for each automaton and the maximum time. The next two columns report average and maximum time to prove with the exhaustive method that each path is actually infeasible. Hence, by relating the two average values, we get in the third column the speedup our generalization method offers (> 1 is good). The four last columns of the table perform the same comparison in terms of memory usage.

Note that solver IC was incapable to prove the infeasibility of most paths in `selection` and `tcas` as the path conditions they contain lead the solver in a too long computation (they were allocated 5 seconds per path).

At first glance, Table II shows the method did generalize input paths to substantial families of infeasible paths, except for some programs (`gcd` and `tritype`). Table III gives the speedup that our generalization approach can reach in average.

TABLE III: Time and memory results on the example programs

Program	Solver	Gen. time (ms)		Exh. time (ms)		Speedup	Gen. MU (Kb)		Exh. MU (Kb)	
		Avg.	*Max.*	*Avg.*	*Max.*		*Avg.*	*Max.*	*Avg.*	*Max.*
`erfill`	Colibri	2	7	37	399	**14.8**	197	550	108	142
	IC	2	7	35	374	**15.0**	232	610	131	173
	Z3	8	18	48	524	**6.0**	564	942	431	473
`f1`	Colibri	3	7	229	844	**77.1**	199	586	92	114
	IC	3	7	224	851	**80.8**	229	645	112	138
	Z3	19	66	189	840	**9.8**	531	964	409	436
`f2`	Colibri	2	3	32	61	**17.2**	145	235	115	115
	IC	2	7	29	58	**17.0**	168	273	139	139
	Z3	9	13	38	70	**4.4**	465	571	439	440
`gcd`	Colibri	0	0	0	1	**0.7**	33	37	18	19
	IC	0	0	2	2	**0.2**	38	43	21	22
	Z3	8	9	5	8	**0.6**	336	340	313	315
`merge`	Colibri	5	10	163	523	**31.9**	402	703	135	142
	IC	5	10	171	510	**37.5**	449	770	165	173
	Z3	9	21	175	516	**18.8**	774	1095	463	472
`selection`	Colibri	4	7	520	1406	**141.7**	483	893	108	133
	IC	–	–	–	–	**–**	–	–	–	–
	Z3	11	66	459	1256	**43.6**	840	1278	429	467
`tcas`	Colibri	16	23	1159	2687	**74.3**	1478	2308	82	93
	IC	–	–	–	–	**–**	–	–	–	–
	Z3	46	303	763	2035	**16.6**	1808	2654	423	447
`tritype`	Colibri	1	3	1	4	**1.3**	50	213	26	37
	IC	1	3	2	5	**2.2**	48	82	31	41
	Z3	6	34	4	8	**0.7**	359	541	324	344

In most cases, the generalization is interesting, except for the few cases where the speedup slips below 1. Table III also shows the generalization uses more memory than the exhaustive approach but its memory need stays reasonable. As these results show the method is not always profitable, one important question is how do we determine when to use the method. The three following points face this problem from different angles.

First, we wanted to analyze the influence of the solver on the method. As shown in Table II, there is few differences between the input feasible paths for each solver. For instance, Z3 was able to detect shorter infeasible paths than the propagation of Colibri on `tcas`. Table III shows the observed speedups are better with Colibri or IC. Note however this study does not intend to compare solvers between each others. This is mostly due to the fact that the consistency check of Z3 is stronger than the propagation of Colibri and IC. Another more marginal reason is the implementation, namely a looser binding with Z3.

Second, we were interested in the type of programs that could benefit from this method. Generalization was poor in two cases (`gcd` and `tritype`) as these programs have been so studied that they do not contain any superfluous dependency anymore. Hence, generalization is harder in these cases. The results are particularly good for `merge`, `f1`, `f2`, `selection` and `tcas`. Indeed, `merge` and `selection` have tightly dependent loops for which the generalization finds several infeasible combinations of iterations. On the contrary, for `f1` and `f2` it is due to the fact that the loop does not intervene in most infeasible paths. In `tcas`, which is mostly a sequence of very related conditional statements, our method limits the combinatorial explosion of infeasible paths.

Third, we wanted to study the impact of the length of the input feasible path on the generalization. In fact, we remarked that the longer the input the more likely the method is advantageous. Fig. 9 shows six diagrams giving the speedup in function of the length of the input path for the three solvers and two programs (`merge` and `erfill`). These diagrams show the generalization is better in terms of CPU time when longer paths are seeded to the method. However, these diagrams also highlight the strong dispersion of the speepdup.

Finally, we tried to extend the use of the previously generated automata beyond the limit of 50 instructions. Given the previously computed automata for three programs `erfill`, `merge` and `tcas`, the chart of Fig. 10 gives for Colibri and Z3 the variation of the average speedup (first over the automata and then over the three programs) between the previous generalization (on input paths of less than 50 instructions) and the exhaustive method on all generalized paths of less than s instructions, in function of this bound s. This chart shows the generated automata are rarely useful to skip smaller paths, but can really be used for longer paths.

V. Related Work

Using unsatisfiable core to prune a search space is not a new idea in SAT solving. Under the generic terminology of *clause learning*, SAT solvers exploit unsatisfiable cores to speed up the unit propagation of the DPLL algorithm. For instance, Zhang and Malik in [15] shown that an explanation

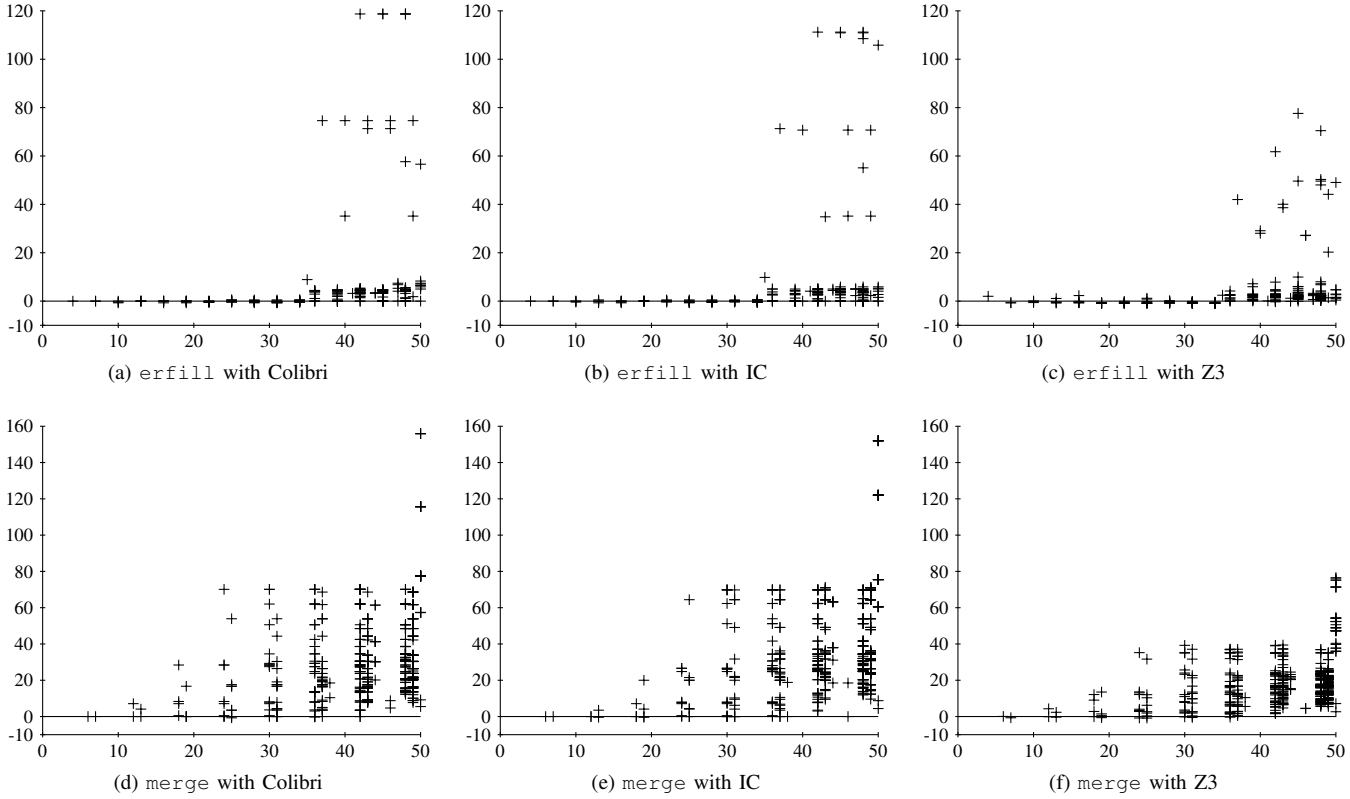

Fig. 9: Speedup in computation time in function of the input infeasible path's length

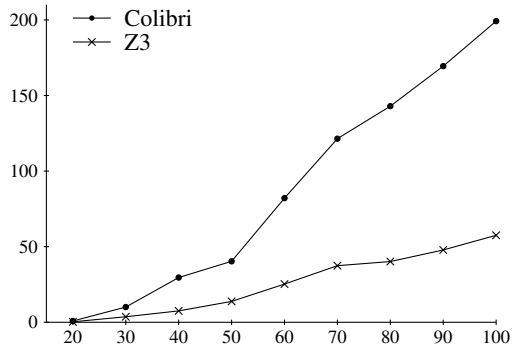

Fig. 10: Speedup in function of the generalized paths' maximum length

can be computed as a by-product of a DPLL-based procedure, where the unsatisfiable core consists of all the clauses used by the DPLL solver. Note just that, unlike our approach, *clause learning* can be considered as an intrusive method as it requires deep interactions with the SAT solver. In the context of the automatic test data generator Pex [4], the SMT solver Z3 [8] is used and apart from the work of Cimatti et al. [16] who proposed using external SAT-based unsatisfiable cores, we are not aware of any work consisting in computing unsatisfiable core on dedicated decision procedures of SMT solvers.

The only work in software testing that tries to generalize infeasible paths we are aware of, is the work of Ngo and Tan [17]. 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 ) ...; if( x > y) ...`). As the pattern recognition are sound, all the detected paths are indeed infeasible. Given 5963 paths (2276 of which are infeasible) on six Java programs (from 6 to 220 KLOC), their method recognizes 82.3% of all infeasible paths. In [7], Ngo and Tan extended their approach to empirical properties instead of patterns to automatically detect non-feasible paths. They implemented the approach in a tool called jTGEN that was used to generate test data for Java programs. Our approach distinguishes from these works on two main points. First, our method to generalize infeasible paths is based on a sound and minimal explanation detection as it computes Γ-*minimal explanation*. Hence, the explanation we compute is more general than the patterns of the work of Ngo and Tan. Second, unlike their approach which has no generalization part, our generalization algorithm is built on automata operations and approximate data flow computations. Experimental comparison between our approaches is not possible as our tool IPEG is dedicated to C programs, but we are confident in the capability of IPEG to detect more infeasible paths than jTGEN.

VI. Conclusion

In this paper, we developed a novel method to generalize infeasible paths from the detection of a single infeasible path. Our approach relies first on the computation of non-intrusive constraint-based explanation and second on automata operations and approximate data flow computations. The experimental results we obtain with a prototype tool implementation show that, whatever is the consistency check (IC, Colibri and the SMT solver Z3), our approach can save considerable time over an approach that does not make use of the generalization algorithm. Perspectives of this work include the improvement of the generalization algorithm with "semantics information" about parts of the program not related to the Γ-*minimal explanation*. We plan to exploit the underlying constraint or SMT solver to find improved ways to generalize the infeasibility. Another line of work concerns the usage of the automaton of infeasible paths in automatic test data generators and other applications. We plan first to integrate our tool IPEG into PathCrawler [2] in order to speed up its path-oriented test data generation process and then we will study its potential usage in static analysis approaches.

Acknowledgment

We thank Bruno Marre and Benjamin Blanc for providing the Colibri solver and useful discussions about explanation, Nicky Williams for preliminary discussions on this work and Phillipe Herrmann for pertinent inputs on SMT formula formats.

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proc. of PLDI'05*, 2005, pp. 213–223.

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

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

[4] N. Tillmann and J. de Halleux, "Pex: White box test generation for .net," in *Proc. of the 2nd Int. Conf. on Tests and Proofs*, ser. LNCS 4966, 2008, pp. 134–153.

[5] E. J. Weyuker, "The applicability of program schema results to program," *International Journal of Parallel Programming*, vol. 8, no. 5, pp. 387–403, Oct. 1979.

[6] D. Yates and N. Malevris, "Reducing the effects of infeasible paths in branch testing," in *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, 1989, pp. 48–54.

[7] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Inf. Softw. Technol.*, vol. 50, no. 7-8, pp. 641–655, 2008.

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

[9] N. Jussien, R. Debruyne, and P. Boizumault, "Maintaining arc-consistency within dynamic backtracking," in *Principles and Practice of Constraint Programming (CP 2000)*, ser. Lecture Notes in Computer Science, vol. 1894. Springer-Verlag, 2000, pp. 249–261.

[10] U. Junker, "QuickXplain: Conflict detection for arbitrary constraint propagation algorithms," in *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, August 2001.

[11] C. Meudec, "ATGen: automatic test data generation using constraint logic programming and symbolic execution," *Software Testing Verification and Reliability Journal (STVR)*, vol. 11, no. 2, pp. 81–96, June 2001.

[12] S. Bardin and P. Herrmann, "Structural testing of executables," in *1th Int. Conf. on Software Testing, Verification and Validation (ICST'08)*, 2008, pp. 22–31.

[13] B. Marre and A. Arnould, "Test sequences generation from lustre descriptions: Gatel." in *Proc. of the 15th IEEE Conference on Automated Software Engineering (ASE'00)*. IEEE CS Press, Septembre 2000.

[14] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[15] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formulas," *SAT'2003*, 2003.

[16] A. Cimatti, A. Griggio, and R. Sebastiani, "A simple and flexible way of computing small unsatisfiable cores in sat modulo theories," in *SAT'07*, ser. Lecture Notes in Computer Science, vol. 4501. Springer, 2007, p. 334.

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

Appendix

```
#define N 5
int erfill(int a[N],int e,int f) {
  int i, j, l;
  l = N; i = 0;
  while (i < l) {
    if (a[i] == e) {
      for (j = j+1; j < l; j++)
        a[j-1] = a[j];
      l = l-1;
    } else {
      i = i +1;
    }
  }
  while (l < N) {
    a[l] = f:
    l = l + 1;
  }
}
void f1(int x,int y,int z) {
  int a, p, i;
  a = (x != 0) ? 5 : 6;
  for (i=1; i < y; i++)
    if (z != i)
      p = i*p;
  if (z == 1) printf("ok\n");
}
#define N 5
#define M 5
void merge(int t1[N],int t2[M],int t3[N+M]) {
  int i = 0, j = 0, k = 0;
  while (i < N && j < M) {
    if (t1[i] < t2[j]) {
      t3[k] = t1[i];
      i++;
    } else {
      t3[k] = t2[j];
      j++;
    }
    k++;
  }
  while (i < N) {
    t3[k] = t1[i];
    i++; k++;
  }
  while (j < M) {
    t3[k] = t2[j];
    j++; k++;
  }
}
```