A General Approach for Expressing Infeasibility in Implicit Path Enumeration Technique [*]

Pascal Raymond
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France
Pascal.Raymond@imag.fr

ABSTRACT

Static timing analysis aims at computing a guaranteed upper bound to the Worst-Case Execution Time (WCET) of a program. It requires both an accurate modeling of the hardware, and a precise analysis of the program in order to reject infeasible executions (in particular, all infinite ones). For the actual computation of the worst-case execution, most of the existing tools and methods are based on the Implicit Path Enumeration Technique (IPET), which consist in encoding this search into a numerical optimization problem (Integer Linear Programming, ILP). An interest of this approach is that it naturally integrates the loop bounds. It also allows to implicitly prune infeasible paths, as far as they can be expressed using linear constraints. Several works on the subject are using this ability in order to enhance the WCET estimation: they identify specific property patterns (e.g., implications, exclusions) and propose ad hoc translation into numerical constraints.

The goal of this paper is to go further than ad hoc reasoning by proposing a general method for translating infeasibility in terms of numerical constraints. It does not address the problem of finding infeasible paths, only the one of characterizing them as precisely as possible. Moreover the paper aims at exploring the limits of the method, and thus, it does not try to enhance the result using additional methods (e.g., graph transformation).

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

WCET, infeasible path, Integer Linear Programming

[*]This work is supported by the french research fundation (ANR) as part of the W-SEPT project (ANR-12-INSE-0001).

ESWEEK'14, October 12 - 17 2014, New Delhi, India.
Copyright 2014 ACM 978-1-4503-3052-7/14/10 ...$15.00
http://dx.doi.org/10.1145/2656045.2656046

1. INTRODUCTION

Static timing analysis aims at computing, given a binary code and the description of the architecture, a proven upper bound to the Worst-Case Execution Time (WCET) of the code (see [8] for an overview of methods and tools). Existing methods and tools are mainly organized in 3 steps.

(1) *Data-flow Analysis* considers the semantics of the code (often requiring the availability of the source code, e.g., C) in order to identify the paths in the CFG that are actually feasible; this phase must at least find accurate loop bounds, in order to retain only finite executions.

(2) *Micro-architecture Analysis* builds the Control Flow Graph (CFG) of the code, made of Basic Blocks (BB) of sequential instructions connected by edges representing the possible control paths. It then computes, for each BB and/or edges, a local WCET estimation (aka local weight), taking into account as precisely as possible hardware features like memory cache, pipeline, branch predictor.

(3) *Worst-Case Path Search* comes at last, to identify a/the worst-case execution path according to the weights assigned to each BB and/or edges. For this purpose, the mostly used methods are based on Implicit Path Enumeration Technique (IPET), introduced in [6].

The key idea of this method is to encode the problem of finding a worst-case path into a numerical optimization problem, more precisely, an Integer Linear Programming problem (ILP). An integer variable is associated to each BB and/or edge, representing the number of time the block/edge is traversed during an execution. The structure of the graph can then be expressed as a set of linear constraints, according to the well-know principle that "incoming flow equals outgoing flow", and that the entry point of the program is executed once. Other constraints, coming from the data-flow analysis, are added to the numerical system, comprising at least the ones that are necessary to bound all loops in the graph. The objective function is then to maximize the weighted sum of all these counters, and it can be solved by a state-of-art Integer Linear Programming solver.

A main interest of the IPET method, pointed out since its first proposal [6], is its ability to implicitly prune lots of infeasible paths by stating simple linear constraints. The study of this ability is the subject of this paper. To complete this introduction, we present now some related works and some examples that illustrate the problem and motivate this work.

1.1 State of the art

To our knowledge, there is no previous work specifically dedicated to the expression of infeasible paths by means of ILP constraints. More precisely, almost all works based on the IPET method are considering this problem, but mainly as a complement to the one of finding the properties that make the paths infeasible. Papers on this subject are numerous, and a general characteristic is that they search for particular property patterns leading to infeasible path, that are translated into particular constraint patterns. These patterns are mainly based on the notion of conflicting edges: edges that cannot be all traversed during the same execution. Papers are considering pairwise conflicts (conflict-pair), or more generally n-ary conflicts (conflict-list).

For instance, [1, 4] are handling conflicting lists of edges, that can be expressed as bounded sums; it is similar to the notion of conflict we develop in this paper, but only in the case of programs without loops. Papers like [2, 3, 4] go further by considering properties holding in loop scopes, in the case of pairwise relations between edges (conflict pairs, exclusions, equalities etc.). Moreover, almost all works notice (often implicitly) that purely conjunctive linear programming is unable to express all kind of conflicting properties. The general solution is to define a mix of case-by-case analysis and pure ILP. This kind of methods can be referred to as splitting-based methods, in the sense that they consists in splitting the problem to make paths more and more explicit. Technically, they can be based on control-flow graph transformation, addition of extra variables and/or disjunctive linear systems (in order to express non convex domains [5, 6, 7]).

The work presented in this paper is somehow transversal to these related works. In particular it does not consider at all the problem of finding properties, but only focusses on the expression of these properties as ILP constraints. We aim at considering in a homogeneous manner programs with or without loops, and relations between two or more edges (conflicting pairs or lists). Moreover, we try to explore the limits of pure ILP constraints, and thus we do not consider splitting-based methods.

1.2 Examples

Program 1 shows an example of code written in pseudo-C (left). The actual CFG, handled by the timing analyzing tool, is shown in the middle. For reasoning about path feasibility, we use the simplified graph on the right: sequential blocks are abstracted to outline the branching possibilities. Each transition is identified by a letter (a,b,c, etc.) that helps to relate it with the source C code.

In the classical IPET approach, an integer variable is associated to each edge. This variable is a counter, whose value is the number of time the edge is traversed during one particular execution. We note these counters with the same letter than the corresponding edge. Let $E = \{a, b, c, ...\}$ this set of variables. The weights (local WCET) computed during the micro-architecture analysis are denoted w_a, w_b, etc. The IPET goal is to maximize, under a set of linear constraints, the objective function:

$$\sum_{x \in E} w_x \cdot x$$

The CFG of Program 1 can be literally translated into a set of structural constraints:

$$\mathcal{S} = \left\{ \begin{array}{ccc} 1 = a + d & a + d = g & g + k = h + \ell \\ h = b + e & b + e = c + f & c + f = k \end{array} \right\}$$

With these structural constraints only, there is trivially no bound to the objective function: the estimated WCET is infinite. This is why the control-flow analysis must at least produce a bound for each loop in the program. We suppose that such a constant bound n is given (e.g. 100), and a new constraint is added to obtain the basic set of constraints :

$$\mathcal{B} = \mathcal{S} \cup \{h \leq n\}$$

If we only consider the constraints set $\mathcal{B}$ (structural and loop bounds), the interest of the ILP encoding is not obvious. The WCET can be computed without the help of a numerical solver by applying a simple max/plus inductive algorithm:

1. The WCET of a sequence is the sum.
2. The WCET of a choice is the max.
3. The WCET of a loop bounded by n is n times the WCET of one iteration.

The ILP method becomes interesting if there exist additional information about edges that are not structural but rather semantical. Such an information is for instance that two (distant) edges are *incompatible* (or conflicting), in the sense that they cannot be both taken during the same execution. The purpose of this work is not at all to find these properties: we just make the statement that they exist, and our goal is to express them as precisely as possible in the IPET framework. Let us now precise this notion of incompatibility on the example.

Conflict within a loop. On the example program, one can observe that executing the branch e sets the variable `cond` to true, and, as a consequence, makes it impossible to take immediately after the branch f. This kind of property is very easy to express as an ILP constraint if the program has no loop: since each edge is executed at most once, the constraint $e + f \leq 1$ precisely express that at most one edge can be executed. Here the problem is slightly more complex because the edges belong to a loop. A simple ad hoc reasoning shows that the conflict holds at each iteration of the `for` statement, and that the number of iterations is precisely captured by the value of the edge counter h. The conflict can be expressed by:

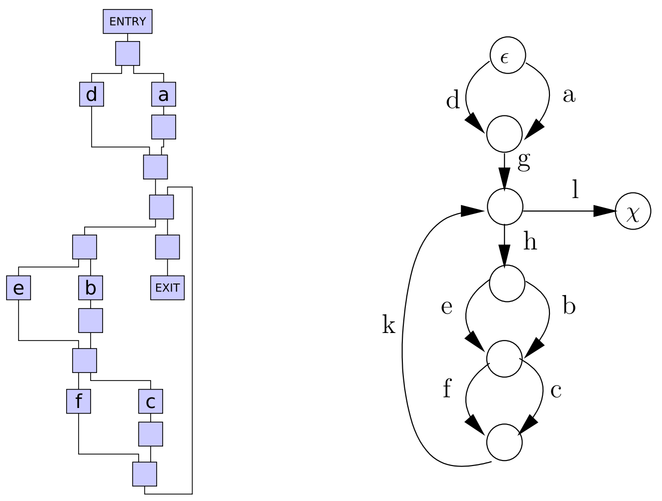
$$e + f \leq h$$

Since we know that h is bounded by the constant n, we may also directly write:

$$e + f \leq n$$

Most of the examples found in the literature are very similar to this case: authors search for particular property patterns that lead to particular constraint patterns, typically a sum bounded by a constant.

```
if (init) {
    /* a */
} else {
    /* d */
}
for(i=0;i<n;i++){
    if (Y[i]) {
        cond = not init and Z[i];
        /* b */
    } else {
        cond = true;
        /* e */
    }
    /* ... */
    if (cond){
        /* c */
    } else {
        /* f */
    }
}
```



Program 1: simplified C code, actual binary CFG and simplified CFG

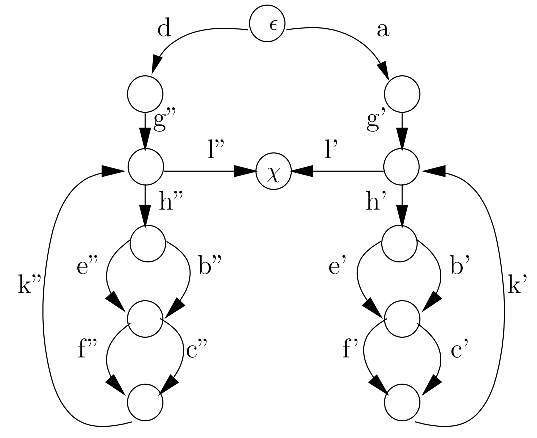


Figure 1: Unfolding the CFG of Program 1 to distinguish executions where a is taken or not.

Conflict across a loop. A less obvious semantic relation exists between the initial `if` statement and the two statements inside the loop: if the variable `init` is true, then in each loop, if `Y[i]` is true then `cond` is set to false and branch c becomes unreachable. This is yet another example of conflict, but involving three edges instead of two: a, b and c. Moreover, these edges do not belong to the same loop scope, making impossible to simply bound their sum.

A usual solution to treat this kind of property consists in splitting the problem according to several cases. This split can be made explicitly by unfolding the CFG as shown in Figure 1. We obtain a new graph, and thus a new ILP system which is equivalent with respect to the WCET computation. Note that the conflict previously discovered between e and f, simply applies to the new "versions" of the edges; from now on we use the more visual name of *avatars* to identify the several versions of an edge: $e' + f' \leq n$ and $e'' + f'' \leq n$. Moreover, we can now precisely express the new conflict, that holds only for the prime avatars of b and c (i.e., after the execution of a only):

$$b' + c' \leq n$$

CFG transformation (unfolding, loops unrolling) is a general solution for expressing any kind of infeasibility: the more detailed is the graph, the more explicit are the paths. However it contradicts the spirit of the IPET method, and it is technically limited by the combinatorial growing of the graphs.

A clever solution consists in keeping the splitting implicit, by introducing as few extra variables as possible. For instance, we can start with the original ILP system of Program 1 and introduce only the avatars of b and c. The following set of extra constraints, called $\mathcal{A}$, is sufficient to express the property:

$$\mathcal{A} = \left\{ \begin{array}{rcl} b' + c' & \leq & n \cdot a \\ b'' & \leq & n \cdot d \\ c'' & \leq & n \cdot d \\ b' + b'' & = & b \\ c' + c'' & = & c \end{array} \right\}$$

In this system, a behaves like a Boolean oracle whose value "chooses" between two different systems. If $a = 0$, then, because of the structural constraints, we have $d = 1 - a = 1$ and the set $\mathcal{A}$ can be reduced to $b \leq n$ and $c \leq n$, which is redundant with the structural constraints. If $a = 1$, $\mathcal{A}$ can be reduced to the expected conflicting constraint : $b+c \leq n$.

The solution of introducing extra variables is still not really in the spirit of the IPET, since extra variables are just a way to make paths more explicit.

In fact, for this particular case, ad hoc reasoning shows that the property can be expressed exactly without the help of a splitting method. The following constraint, involving only original variables, precisely captures the expected conflict:

$$n \cdot a + b + c \leq 2n$$

Conflict between iterations. Program 2 has the same control flow graph as Program 1, but the order of the if state-

```
if (init) {
   /* a */
} else {
   /* d */
}
cond = ... ;
for(i=0;i<n;i++){
   if (cond){
      /* b */
   } else {
      /* e */
   }
   /* ... */
   if (Y[i]) {
      cond = not init and Z[i];
      /* c */
   } else {
      cond = true;
      /* f */
   }
}
```

Program 2: Same CFG as Program 1, if statements are reversed

ments in the loop is reversed. As a consequence, whenever a is taken, the conflict between b and c does not occur during the same iteration, but between two consecutive iterations: if c is taken, `cond` is set to false, and in the next iteration (if it exists) b cannot be taken. Even if it is not so different from the previous example, this kind of inter-iteration property is hardly handled in existing work. Here again the problem can be solved by unfolding the graph or introducing extra variables. However, here again, it is not necessary to do so. Let n be the loop bound, the following constraint precisely captures the conflict:

$$(n-1)\cdot a+b+c\leq 2n$$

Here is an informal proof sketch:

- if $n=0$, then $b=c=0$ and the constraint becomes a tautology: $-1\leq 0$;
- if $n=1$ or $a=0$, the constraint becomes redundant: $b+c\leq 2n$;
- if $n\geq 2$ and $a=1$ the constraint becomes $b+c\leq 2n-(n-1)=n+1$. By doing a (virtual) unfolding of the n iterations, on can exhibit n avatars of b and c, denoted $b_1,\cdots b_n$, $c_1,\cdots,c_n$. Both b_1 and c_n are unconstrained, and then, their sum is bounded by 2. All others pairs (c_k,b_{k+1}) are conflicting and thus $c_k+b_{k+1}\leq 1$. Since there are $n-1$ such pairs, the sum of all the avatars is bounded by $2+(n-1)=n+1$, which is the bound given by the formula.

Limits of numerical constraints. Unfortunately, it is not possible to exactly reflect any conflict using only the existing variables. Program 3 is very similar to Program 1, except that the tests within the loop are not depending on the current iteration. For this program, if `init` is true, then depending on the value of `Y`, either $b=n$ and $c=0$ or $b=0$ and $c=n$. This property cannot be expressed in classical (conjunctive) linear programming since it requires to express a disjunction: $(b=n\wedge c=0)\vee(b=0\wedge c=n)$. Here again, splitting oriented methods (graph unfolding, extra variable,

```
if (init) {
   /* a */
} else {
   /* d */
}
for(i=0;i<n;i++){
   if (Y) {
      cond = not init and Z;
      /* b */
   } else {
      /* e */
      cond = true;
   }
   /* ... */
   if (cond){
      /* c */
   } else {
      /* f */
   }
}
```

Program 3: Same CFG as Program 1, conditions do not depend on the current iteration

disjunctive systems) can solve precisely the problem, but they are not considered in this paper.

Nevertheless, for this program, the simple linear constraint $n\cdot a+b+c\leq 2n$ still holds and, while not perfect, may prune some infeasible paths.

1.3 Motivation and paper organization

The goal of this work to go further than ad hoc reasoning by proposing a general method to produce linear constraints such as the ones presented in the examples ($e+f\leq n$, $n\cdot a+b+c\leq 2n$, etc.). The method should handle in a homogeneous manner programs with or without loops, intra or inter loop properties.

For this purpose, a first requirement is to precise the "nature" of an infeasibility property. Moreover, we want to explore the limits of the ILP expressiveness without the help of any split-oriented method: no graph transformation, no extra variables, no disjunction.

To summarize, a first definition of the goal is: given a CFG and information on the incompatibility between edges, find one or more linear constraint(s) that cut, as precisely as possible, the corresponding infeasible path.

The paper is organized as follow: section 2 gives the necessary formal definitions. We show that the notion of *conflicting sets* is atomic with respect to path pruning in the sense that any set of infeasible paths can be described as a union of conflicting sets.

Section 3.1 gives a first general solution for transforming conflicting sets into linear constraints. This solution requires very few information about the incompatibilities, but the counterpart is that the result may be rather imprecise. Section 3.2 presents a solution for capturing more precisely the incompatibilities.

2. DEFINITIONS AND NOTATIONS

2.1 Programs and unfoldings

A CFG (or simply a *program*) is a direct graph, possibly cyclic, made of a finite set of vertices (V), a finite set of edges ($E \subseteq V \times V$), a particular entry vertex (ϵ), and one or more exit vertex ($\mathcal{X}$):

$$P = (V, E \subseteq V \times V, \epsilon \in V, \emptyset \subset \mathcal{X} \subseteq V)$$

In the sequel we use lowercase letters to identify edges (e.g., a, b, c, etc., as in the CFG of Program 1).

A trace of a program P is a sequence of subsequent edges starting in ϵ and ending in $x \in \mathcal{X}$. The set of traces of a program P is denoted $\mathcal{T}(P)$.

In the WCET framework, we are only interested in programs that terminate in bounded time. It implies that we always suppose the existence of a finite set of *actually feasible executions*, which is, in general, a strict subset of the program traces. We do not claim that this finite set is precisely known, just that it exists. We call it the set of *actual executions* and note it $\mathcal{E} \subseteq \mathcal{T}(P)$.

We introduce now the notion of *unfolding* in order to formalize the notion of "more precise" program graph.

Definition 1.
An **unfolding** of a program $P = (V_P, E_P, \epsilon_P, \mathcal{X}_P)$, with respect to a set of executions $\mathcal{E} \subseteq \mathcal{T}(P)$ is a pair (U, δ) such that:

- $U = (V_U, E_U, \epsilon_U, \mathcal{X}_U)$ is a CFG,
- δ is a mapping from U edges to P edges (i.e., a *decoding*'):

  $$\delta : E_U \to E_P$$

  the induced trace mapping is denoted:

  $$\delta^* : \mathcal{T}(U) \to \mathcal{T}(P)$$

  and the set of decoded traces is denoted:

  $$\mathcal{T}^\delta(U) = \{\delta^*(t), t \in \mathcal{T}(U)\}$$

- such that:

  $$\mathcal{E} \ \subseteq \mathcal{T}^\delta(U) \subseteq \mathcal{T}(P)$$

For instance, Figure 1 shows an unfolding of Program 1, where the mapping consists in ignoring the prime symbols; in this case the unfolding is not more precise since $\mathcal{T}^\delta(U) = \mathcal{T}(P)$.

Note that the definition of unfolding can be generalized by considering neutral edges in U: in this case the mapping is a function $\delta : E_U \to E_P \cup \{\tau\}$ where $\tau \notin E_P$ is simply ignored when extending δ to sequences (i.e., $\delta^*(\tau) = \varepsilon$).

Among the unfoldings of P we are particularly interested in those that are acyclic (i.e., DAGs): they *structurally* reject the infinite traces of the original P. In particular, an

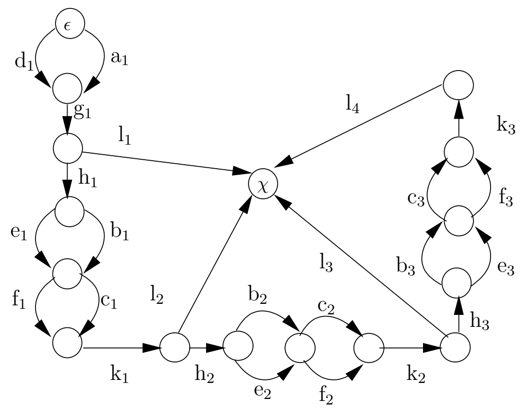


Figure 2: Acyclic unfolding ofProgram 1 (for $n = 3$).

acyclic unfolding is exactly what we get (implicitly) in classical WCET techniques when a bound is assigned to each loop of the program. Figure 2 shows an acyclic unfolding of Program 1, in the case that the number of iterations is bounded by $n = 3$. Note that it trivially exists, within the acyclic unfoldings of P, a *canonical* program that exactly captures the actual executions $\mathcal{E}$ (more precisely, a class of minimal graphs, equivalent modulo edge renaming).

In the sequel, we consider that we have an acyclic unfolding of the program P: we do not require that it must be precisely built or known, only that it virtually exists.

2.2 Avatars and implicit paths

Let P be a program, $\mathcal{E}$ the set of actual executions and (U, δ) an acyclic unfolding. From now on we call P the concrete program (and its edges the concrete edges). The edges in the reverse image of the concrete edge a are called the *avatars* of a. We note m_a the number of avatars of a ($m_a = |\delta^{-1}(a)|$), and, by convention, we note the avatars with subscript indices, like in Figure 2[1]:

$$\delta^{-1}(a) = \{a_1, a_2, \cdots, a_{m_a}\}$$

Let us now come back to the IPET framework. In IPET, sets of traces are characterized implicitly by giving the number of occurrences of the edges. We call these numbers the *edge counters*, and note, for instance, $|a|_t$ the number of occurrences of the edge a in the trace t. In order to simplify the notations, and whenever the context clearly concerns a trace t, we will simply note a for $|a|_t$.

The basic relation between the counters of an unfolded trace t and the corresponding concrete trace $t' = \delta^*(t)$ is trivially:

$$\sum_{i=1}^{m_a} a_i = a$$

Another trivial property of the unfolded counters is that they are either 0 or 1:

$$0 \leq a_i \leq 1$$

[1]This does not mean that we consider a "natural" ordering of the avatars, just think about them as a family of symbols.

2.3 Conflicting sets

Let P be a program, $\mathcal{E}$ the set of actual executions and (U, δ) an acyclic unfolding. U structurally rejects any infinite infeasible paths, however there is still a precision gap represented by $G = \mathcal{T}^{\delta}(U) \setminus \mathcal{E}$ which is the set of infeasible finite paths not rejected by U.

Definition 2.
Let $C \subseteq E_U$ be a set of edges, we note $\mathcal{T}_U^{\delta}(C) \subseteq \mathcal{T}^{\delta}(U)$ the set of traces of U that pass by all the edges in C:

- C is a **conflicting set** if $\mathcal{T}_U^{\delta}(C) \cap \mathcal{E} = \emptyset$ and $\mathcal{T}_U^{\delta}(C) \neq \emptyset$, i.e., it contains only infeasible paths, among them at least one not rejected by U;
- C is a **minimal conflicting set** iff any $C' \subset C$ is not a conflicting set;
- A set $\mathcal{C} = \{C_1, \cdots, C_k\}$ is a **conflict covering** iff:

$$\mathcal{T}^{\delta}(U) \subseteq \mathcal{E} \bigcup_{1 \leq i \leq k} \mathcal{T}^{\delta}{}_U(C_i)$$

  i.e., the conflicting sets of $\mathcal{C}$ are sufficient to reject any infeasible path from U;
- $\mathcal{C}$ is a **minimal conflict covering** iff it contains only minimal conflicting sets, and any $\mathcal{C}' \subset \mathcal{C}$ is not a conflict covering.

THEOREM 1. *For all $(P, (U, \delta), \mathcal{E})$, there exist a minimal conflict covering.*

Here is a sketch of a constructive proof, which is indeed related to the notion of prime implicant in Boolean algebra:

- there exist (at least) one conflict covering: the one obtained form $G = \mathcal{T}^{\delta}(U)) \setminus \mathcal{E}$ by interpreting the paths as (unordered) sets of edges,
- if a conflicting set C in a covering is not minimal, replace it by some other conflicting set $C' \subset C$,
- if a covering is not minimal, replace it by some covering $\mathcal{C}' \subset \mathcal{C}$.

This theorem is relatively trivial, but nevertheless important since it justifies the fact that the notion of conflicting sets is (implicitly) considered equivalent to the one "pruning properties" in the literature.

In an unfolding, the incompatibility due to a conflicting set can be expressed *exactly* in terms of ILP constraints. Consider a multiset of n concrete edges (we use superscript indexes to avoid confusion with avatar notation): a^x for $x = 1 \cdots n$. Consider a set of n avatars, one per each concrete edge a^x: $C = \{a^x_{i_x} | x = 1 \cdots n\}$. Note that we consider multisets of concrete edges for the sake of generality: it is possible that different avatars of a same concrete edge are conflicting; it arises for instance when a edge in a loop cannot be taken in two consecutive iterations. If C is a conflicting set, then, for any execution:

$$\sum_{x=1}^{n} a^x_{i_x} \leq n - 1$$

For the same multiset of concrete edges, it is likely that many others sets of avatars are also conflicting. This leads to the notion of "conflicting from time to time": a multiset of (concrete) edges $\{\{a^x\}\}$ is conflicting s times (i.e., is s-conflicting) if there exists s sets of their avatars that are conflicting.

3. COMPLETION OF s-CONFLICTING

3.1 Rough completion

3.1.1 Case of 3 edges

In order to make the presentation more clear, and to keep the notations readable, we consider here the case of 3 concrete edges, denoted a, b and c. Note that nothing in the following reasoning requires the these edges should be all different: $|a, b, c|$ must be understood as a multi-set containing 3 edge references (e.g. a and b may refer to the same concrete edge). The reasoning can be easily extended to the general case of any number of edge references.

Even if the reasoning is based on the existence of an acyclic unfolding, we try to keep the information about it as abstracted as possible; we suppose given:

- the number of avatars of each edge, denoted m_a, m_b and m_c;
- the number of conflicting sets of avatars, denoted s.

In order to avoid confusion, we use different index symbols for the avatars. Avatars are denoted a_i for $i = 1 \cdots m_a$, b_j for $j = 1 \cdots m_b$, and c_k for $k = 1 \cdots m_c$. The corresponding counters properties, holding for any execution are:

$$a = \sum_{i=1}^{m_a} a_i, \quad b = \sum_{j=1}^{m_b} b_j, \quad c = \sum_{k=1}^{m_c} c_k$$

We know that $\{\{a, b, c\}\}$ is s-conflicting. Thus, there exists a set S of s triples (i, j, k) such that each $\{a_i, b_j, c_k\}$ is a conflicting set. The sum of the corresponding linear constraints gives:

$$\sum_{(i,j,k) \in S} a_i + b_j + c_k \leq 2s$$

The idea is now to complete this constraint in order to get rid of the avatar details and come up with a relation between the full counters only. Without any other information than the number s, we can hardly do better than complete the relation with all the triples that do not belong to S. For these triples, such that $(i, j, k) \notin S$, the following trivial relation holds:

$$a_i + b_j + c_k \leq 3$$

Moreover, there are $m_a m_b m_c - s$ such triples. By summing all these trivial constraints with the conflicting ones, we obtain, in the left hand side, $m_b m_c$ times the sum of all a avatars, that is, the complete a counter. Similarly, the full b appears $m_a m_c$ times, and the full c $m_a m_b$ times. Finally, we obtain a relation free of any avatar details:

$$m_b m_c a + m_a m_c b + m_a m_b c \leq 2s + 3(m_a m_b m_c - s)$$

or equivalently:

Formula 1.

$$\frac{m}{m_a}\,a + \frac{m}{m_b}\,b + \frac{m}{m_c}\,c \leq 3m - s$$

$$\text{where } m = m_a m_b m_c$$

3.1.2 General case

The 3-edges formula can be easily generalized to the incompatibility between any number of (possibly redundant) edges: let X be a multiset of $|X|$ concrete edges. Each edge occurrence $x \in X$ is characterized by its number of avatars m_x, and the incompatibility is characterized globally by the number of conflicting sets of avatars s:

Formula 2.

$$\sum_{x \in X} \frac{m}{m_x}\,x \leq |X|m - s$$

$$\text{where } m = \prod_{x \in X} m_x$$

3.1.3 Inefficiency of rough completion

Let us consider the left hand part of Formula 2: each edge counter x is, by definition, bounded by m_x, thus the whole left hand sum is intrinsically bounded by $m * |X|$. The gain in precision of the Formula is then only s. Except for very special cases (big s, small number of edges and/or small number of avatars), the formula is unlikely to give a useful information.

One of the rare case where the Formula gives an exact information is when $m_a = m_b = m_c = s$, (i.e. a cycle-free program with 3 incompatible tests): the Formula

$$a + b + c \leq 2$$

precisely "cuts" the infeasible paths.

As soon as the m's are greater, the formula cuts very few infeasible paths, and thus gives very imprecise results. Consider 3 edges within the same loop (executed n times), conflicting at each iteration. The parameters are $m_a = m_b = m_c = s = n$ and the formula gives:

$$a + b + c \leq (3n^2 - 1)/n$$

which is equivalent (since a, b, and c are integers) to:

$$a + b + c \leq 3n - 1$$

The inefficiency is clearly a drawback of the completion, and not of the ILP approach, since we know that there exist, for this particular example, a constraint that precisely cuts the infeasible paths: $a + b + c \leq 2n$.

The goal of the next section is to propose a method for finding, when it exists, a precise ILP formulation of the conflict.

3.2 Precise completion

3.2.1 Multiplicity and lack

The problem of the rough completion is that it introduces a huge amount of useless information of the type $a_i \leq 1$. We will try here to introduce the minimal number of useless information. Consider the incompatibility relation:

$$\sum_{(i,j,k) \in s} a_i + b_j + c_k \leq 2s$$

and focus for instance on the term involving the a_i avatars. This term is of the form:

$$\sum_{i=1}^{m_a} \alpha_i a_i \quad \text{with} \quad \sum_{i=1}^{m_a} \alpha_i = s$$

The maximum of the α_i is called the *multiplicity of* a and denoted $p_a = MAX_{i=1\cdots m_a}(\alpha_i)$.
For each α_i, we define α'_i, its complement to p_a: $\alpha'_i + \alpha_i = p_a$.

Intuitively, the α'_i describe the avatars of a that are missing in the conflict constraint:

$$\sum_{i=1}^{m_a} \alpha_i a_i + \sum_{i=1}^{m_a} \alpha'_i a_i = p_a a$$

Moreover, the details of the α'_i have no importance, only their sum is important:

$$\sum_{i=1}^{m_a} \alpha'_i = p_a m_a - s$$

We call it the *lack of* a in the conflict, and note $\ell_a = p_a m_a - s$.

Consider for instance Program 1 in the case $n = 3$, and the corresponding acyclic unfolding in Figure 2. The numbers of avatars are $m_a = 1$ and $m_b = m_c = 3$. There are $s = n = 3$ conflicting sets $\{a_1, b_1, c_1\}$, $\{a_1, b_2, c_2\}$ and $\{a_1, b_3, c_3\}$. The multiplicity of a is 3 (a_1 appears 3 times in the conflicts), while the multiplicity of b and c is 1 (each avatar appears exactly once in the conflicts). It follows that the lack of a is $\ell_a = 3 \times 1 - 3 = 0$, and the lack of b and c is $\ell_b = \ell_c = 1 \times 3 - 3 = 0$. As a consequence, there is no lack at all in the constraints: no avatar is missing in the conflicts sets.

Consider now the example Program 2, with $n = 3$, for which the CFG in Figure 2 is also an acyclic unfolding. We still have $m_a = 1$ and $m_b = m_c = 3$, but there are now only $s = 2$ conflicting sets: $\{a_1, b_2, c_1\}$ and $\{a_1, b_3, c_2\}$. The multiplicity of a is 2, and thus the lack is $\ell_a = 2 \times 1 - 2 = 0$. The multiplicity of both b and c is 1, and thus the lack is $\ell_b = \ell_c = 1 \times 3 - 2 = 1$. Intuitively, there is a lack in the conflict sets: one avatar of b and one avatar of c are missing. Note that we do not care about what particular avatar is missing or not: only their numbers matter.

3.2.2 Lack completion

For each missing avatar, we can add a trivial constraint stating that is it less than 1, and by summing all these trivial constraints, we obtain:

$$\sum_{i=1}^{m_a} \alpha'_i a_i \leq \sum_{i=1}^{m_a} \alpha'_i = \ell_a$$

The same reasoning holds for the terms in b and c, and finally we can build a global sum of the conflict constraints and the three "lack" constraints that erases the avatar details:

$$\begin{array}{ccccccc}
\sum_{i=1}^{m_a} \alpha_i a_i & + & \sum_{j=1}^{m_b} \beta_j b_j & + & \sum_{k=1}^{m_c} \gamma_k c_k & \leq & 2s \\
\sum_{i=1}^{m_a} \alpha'_i a_i & & & & & \leq & \ell_a \\
 & & \sum_{j=1}^{m_b} \beta'_j b_j & & & \leq & \ell_b \\
 & & & & \sum_{k=1}^{m_c} \gamma'_k c_k & \leq & \ell_c \\
\hline
p_a a & + & p_b b & + & p_c c & \leq & 2s + \ell_a + \ell_b + \ell_c
\end{array}$$

3.2.3 Precise completion (3 edges)

To summarize, in order to obtain a precise translation in ILP of the incompatibility, we need:

- the numbers of avatars m_a, m_b and m_c,
- the number of "times" the incompatibility holds s (i.e., the number of avatar conflicting sets among the $m_a m_b m_c$ possible ones),
- for each edge, its multiplicity in the conflict, that is, the maximum occurrence of a particular avatar a_i in the set of conflicting sets: p_a, p_b and p_c,
- from these information, we compute the relative lacks of each edge, e.g., $\ell_a = p_a m_a - s$,
- and then we can state that the (ternary) s-conflicting formula holds:

Formula 3.

$$p_a a + p_b b + p_c c \leq 2s + \ell_a + \ell_b + \ell_c$$

3.2.4 Precise completion (general case)

This result can be easily generalized to the incompatibility between any number of (possibly redundant) edges. Let X be a multiset of $|X|$ concrete edges. Each edge occurrence $x \in X$ is characterized by its number of avatars m_x. The incompatibility is characterized globally by the number of conflicting sets of avatars s, and, for each edge occurrence x, its multiplicity p_x, and the corresponding lack $\ell_x = p_x m_x - s$. The generalized n-ary s-conflicting formula is:

Formula 4.

$$\sum_{x \in X} p_x x \leq (|X| - 1)s + \sum_{x \in X} \ell_x$$

3.3 Examples

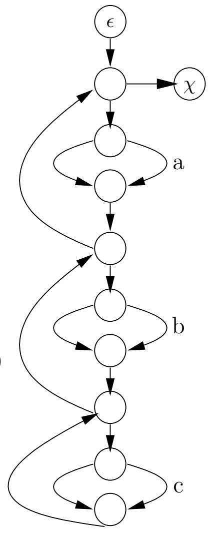
Across-loop conflict. In Program 1, the conflict between a, b and c holds for each of the n iterations, thus:

- $s = m_b = m_c = n$ and $m_a = 1$,
- $p_a = n$ and thus $\ell_a = p_a m_a - s = 0$,
- $p_b = p_c = 1$ and thus $\ell_b = \ell_c = p_b m_b - s = 0$,
- and finally:

  $$n \cdot a + b + c \leq 2n$$

  which is the precise translation of the incompatibility, as it was obtained by ad hoc reasoning in the introduction.

```
for(i=0;i<n1;i++){
 read(x);
 if (x)
 {
  /* a */
 }
 for(j=0;j<n2;j++){
  read(y);
  if(not x or y)
  {
   /* b */
  }
  for(k=1;k<n3;k++){
   read(z);
   if(not(x and y) and z)
   {
     /* c */
   }
  }
 }
}
```



Program 4: Example of ternary conflict in nested loops.

An even simpler example is when the 3 edges belong to the same loop, and the incompatibility holds for each of the n iterations:

- $n = s = m_a = m_b = m_c$,
- $p_a = p_b = p_c = 1$ and thus $\ell_a = \ell_b = \ell_c = 0$,
- and finally:

$$a + b + c \leq 2n$$

Note that the same reasoning works for edges that are in distant loops: only the number of conflict matters, not the precise structure of the graph.

Nested-loops conflict. Program 4 is another example of ternary conflict, but where the conflict propagates within nested loops. The edges a, b and c are appearing (respectively) in nested loops executed *locally* n_1, n_2 and n_3 times, thus:

- $m_a = n_1$,
- $m_b = n_1 n_2$,
- $m_c = n_1 n_2 n_3$,

The conflict propagates to the whole nested loop, i.e.: the first a is incompatible with the n_2 first b and the $n_2 n_3$ first c, and so on. For this example, the avatars are numbered form 0 to $m - 1$ in order to simplify the notations. The incompatibility holds for all the triples:

$$(i, \;\; n_2 i + j, \;\; n_3(n_2 i + j) + k)$$

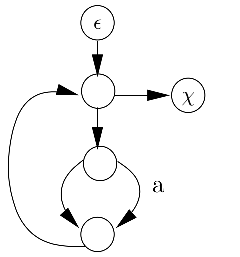
with $0 \leq i < n_1$, $0 \leq j < n_2$, $0 \leq k < n_3$.

It follows that:

- $s = n_1 n_2 n_3$,
- $p_a = n_2 n_3$, $p_b = n_3$, $p_c = 1$ and $\ell_a = \ell_b = \ell_c = 0$
- and finally:

$$n_2 n_3 a + n_3 b + c \leq 2 n_1 n_2 n_3$$

```
cond = read()  ;
for(i=0;i<n;i++){
   if (cond){
      /* a */
      cond = 0;
   } else {
      cond = read();
   }
}
```



Program 5: Edge a is "auto-conflicting" between two consecutive iterations.

Conflict between different iterations. Consider the Program 2, where the conflict holds from one iteration to the following (i.e., if b is taken at loop i, then c cannot be taken at loop $i+1$). In this case:

- $n = m_b = m_c$ and $m_a = 1$,
- $s = n - 1$, since all $\{a_1, b_i, c_{i+1}\}$ is conflicting,
- $p_a = n - 1$, and thus $\ell_a = p_a m_a - s = 0$,
- $p_b = p_c = 1$ and thus $\ell_b = \ell_c = p_b m_b - s = 1$; the global lack is then $2s + \ell_a + \ell_b + \ell_c = 2(n-1) + 2 = 2n$
- and finally:

$$(n-1) \cdot a + b + c \leq 2n$$

Auto-conflict. This example illustrates the fact that conflicting *concrete edges* do not have to be different. Consider Program 5: (1) the loop is bounded by the constant n, (2) whenever a is executed, it becomes unreachable for the next iteration. This is an example of pairwise conflict, covered by the general formula:

$$p_a a + p_b b \leq s + \ell_a + \ell_b$$

where, indeed, one has to keep in mind that $a = b$. The parameters are $m_a = n$, $s = n - 1$, $p_a = 1$ and thus $\ell_a = 1$, and finally:

$$a + a \leq n - 1 + 1 + 1 \quad \Leftrightarrow \quad 2a \leq n + 1$$

4. CONCLUSION

This paper presents a general method for translating infeasibility properties into Integer Linear Programming (ILP) constraints, suitable for the use in IPET method. The translation of infeasibility in terms of ILP constraints is far from new, and numerous examples can be found in the literature. But the goal of this work is not compete on precision or accuracy with existing approaches. It is a theoretical study that proposes a general formulation that in some sense encompass the existing ones and outlines the fundamental limits of the method. In particular, it does not consider the problem of finding "pruning" properties, but only the one of reflecting them as precisely as possible, without the help of any complementary method (e.g., graph transformation). The reasoning is based on the existence of an acyclic unfolding of the program. However this unfolding is kept mainly abstract: a rough solution only requires to identify (1) the number of times each particular edge is unfolded, (2) the number of conflicting sets of edges in the unfolding. In order to provide a finer solution, a more precise information is necessary, that intuitively gives the number of time a particular edge is involved in the infeasibility property. Even with this finer solution, the formulation is sometimes not perfect, in the sense that it does not reject all the infeasible paths. This is not a drawback of the proposed method, but a general limitation of ILP, that arises whenever the exact formulation requires to express disjunction. In this case, the general solution is to combine ILP with by-case reasoning, but this somehow orthogonal problem is not considered here since the idea was to explore the limits of the strict conjunctive ILP formulation.

5. REFERENCES

[1] B. Blackham, M. Liffiton, and G. Heiser. Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In *Real Time and Embedded Technology Applications Symposium*, Berlin, Germany, April 2014.

[2] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS*, pages 163–174, 2000.

[3] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, 2006.

[4] C. Healy and D. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. on Software Engineering*, 28(8), August 2002.

[5] T. H. Kim, H. Bang, and S. D. Cha. A systematic representation of path constraints for implicit path enumeration technique. *Softw. Test., Verif. Reliab.*, 20(1):39–61, 2010.

[6] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(12), 1997.

[7] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, pages 298–, Washington, DC, USA, 1995. IEEE Computer Society.

[8] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7(3), 2008.