TreeFrog for Exhaustive Branch Coverage

Nicky Williams

Université Paris-Saclay, CEA, List,

SAC 2023 30/3/23

Exhaustive branch coverage

Bug finding: cover many branches fast
eg. fuzz testing, DSE, model-checking, combinations techno, heuristics,…

Exhaustive branch coverage

- Try to cover **all reachable branches**
- Provide an **explanation** for uncovered branches
  - either prover/solver demonstrates unreachability
  - or fails/times out: provide formula/Constraint Satisfaction Problem

Needed for certification, to find dead code, assertion violations,…

In fact, many verification problems require either demonstration of unreachability or else witness (ie. test case) of *numerous objectives*

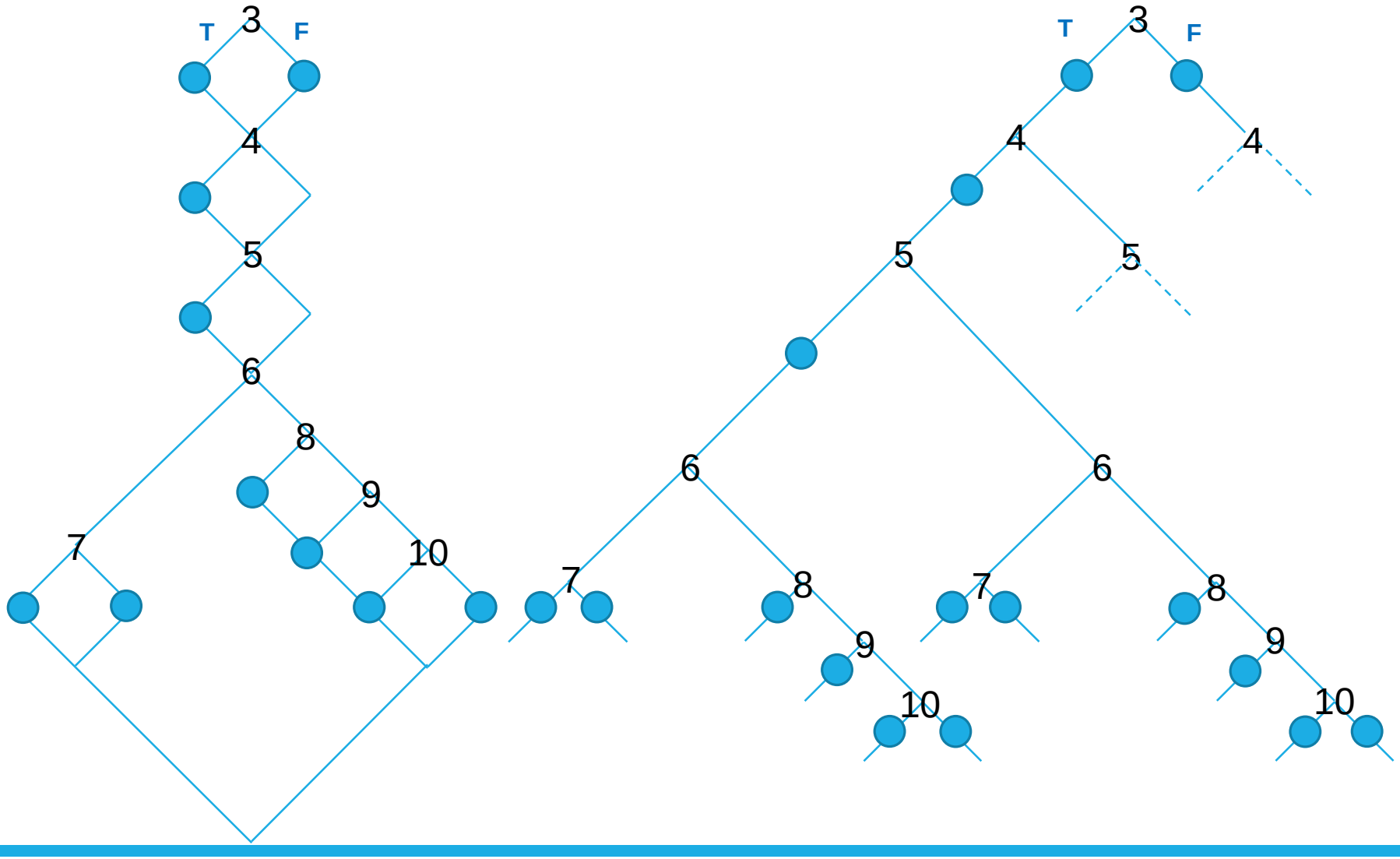
Overview of TreeFrog approach

- We start with concolic test generation
- Like concolic, our approach :
  - Treats all branches at once, is opportunistic, solver-result-driven
  - Is incremental, avoids recalculation, superfluous solver calls and tests
- We supplement concolic with :
  - Learn conflicts between branches
  - Controlled path suffix enumeration
  - Additional tricks
- We treat C functions

Example : buggy tritype

```
1.  int testme(int i, int j, int k){
2.    int t;
3.    if (i == j) t = 1; else t = 0;
4.    if (i == k) t = t + 2;
5.    if ((j == k) && t > 3) t = t + 3;                    +5² unreachable
6.    if (t == 0){
7.      if ((i+j <= k) || (j+k <= i) || (i+k <= j)) t = 4; else t = 1;}
8.    else if (t > 3) t = 3;                               +8 unreachable
9.    else if ((t == 1) && (i+j > k)) t = 2;
10.   else if ((t == 2) && (i+k > j)) t = 2; else t = 4;
11.   return t;
12. }
```

Ex. CFG and Execution Path Tree



Traversal and pruning of Execution Path Tree

Each branch in EP Tree has 1 *occurrence* for each **path prefix** which might reach it

Not all paths are *feasible:* our *search space* is the *pruned* EP Tree

Test-case input values satisfy the *predicate* of the path which they cover

Infeasible path ⇔ *unsat* path predicate

Unreachable branch ⇔ all paths to branch are infeasible

Use *Symbolic Execution* to transform path (sequence branches, assigns,…) into *predicate* ~ conjunction of *constraints* on input values

Solution to constraints = input values for *test-case* to *cover* path

Reminder: concolic test generation

Given a covered path in the EP Tree (ie. with *satisfiable predicate*)
negate the condition of (*flip*) one branch in the predicate
and throw away the following conditions
to form the predicate of a new (*flipped*) path prefix
up to and including flipped branch

If predicate of flipped prefix is *unsat*: prune EP Tree, ie. remove subtree
If solver timeout: memorise predicate
Else solution is test-case inputs which cover flipped prefix and also *some new suffix*

Which new suffix depends on solution chosen by solver

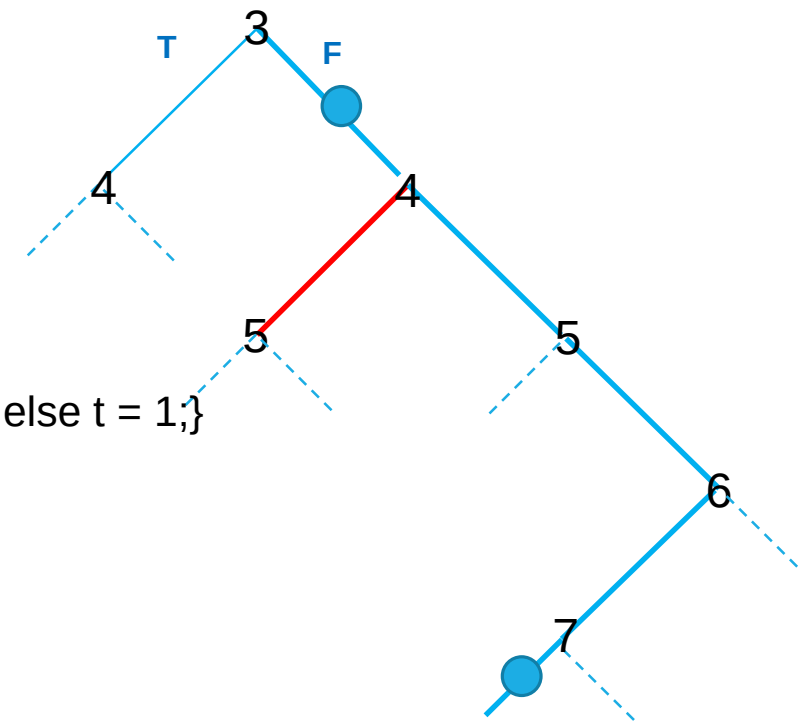
Opportunistic ⇔ partly uncontrolled (driven by solver results)

…

Ex. flip branch -4

path -3 -4 -5^1 +6 $+7^1$

3. if (i == j) t = 1; else t = 0;
4. if (i == k) t = t + 2;
5. if ((j == k) && t > 3) t = t + 3
6. if (t == 0){
7. if ((i+j <= k) || (j+k <= i) || (i+k <= j)) t = 4; else t = 1;}



path predicate

-3 $i \neq j \wedge t_0 = 0 \wedge$

-4 $i \neq k \wedge$

-5^1 $j \neq k \wedge$

+6 $t_0 \neq 0 \wedge$

$+7^1$ $i+j \leq k \wedge t_1 = 4$

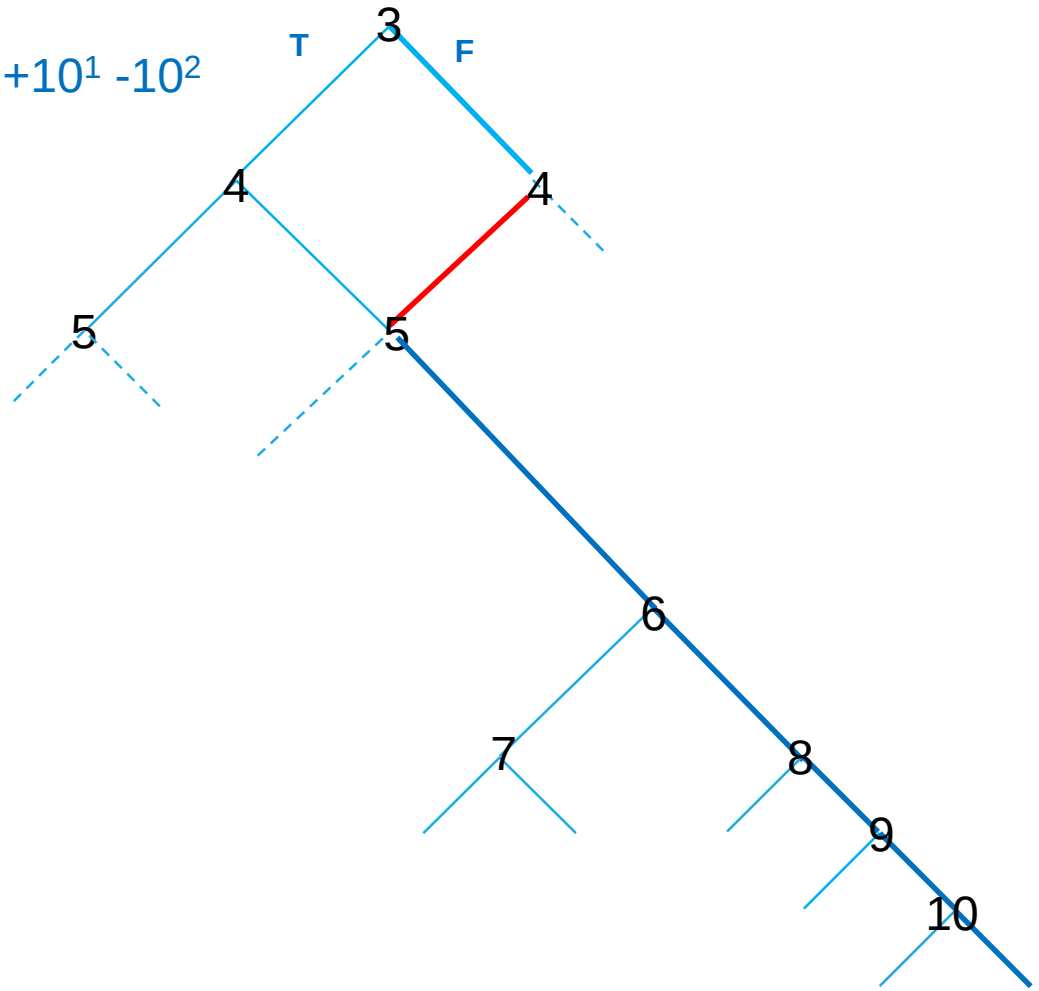
flipped prefix and predicate

-3 +4

$i \neq j \wedge t_0 = 0 \wedge$ i = k

Ex. new suffix flipped branch

TC2: -3 +4 -5^1 -6 -8 -9^1 $+10^1$ -10^2



Reminder: concolic test generation

Given a covered path in the EP Tree (ie. with *satisfiable predicate*)
negate constraint of (*flip*) one branch in the path
to form new (*flipped*) path prefix up to and including flipped branch

If predicate of flipped prefix is *unsat*: prune EP Tree, ie. remove subtree
If solver timeout: memorise predicate
Else solution is test-case inputs which cover flipped prefix and also *some new suffix*

Which new suffix depends on solution chosen by solver

Opportunistic ⇔ partly uncontrolled (driven by solver results)

Ideal for covering all feasible paths because must flip all branches

But to cover all reachable branches, don't usually need (or want) to cover all feasible paths

Only decision to control concolic generation : *which branch to flip next ?*

Concolic test generation for branch coverage

Eager flip: new (*flipped*) branch is an occurrence of an uncovered branch, the solution will cover it (and maybe other uncovered branches)

Hopeful flip: flipped branch is already covered (with a different prefix) but solution may cover some uncovered branches … or not!

Previous work:
concolic method with eager flips first to minimise solver calls

Example of concolic branch coverage

TC1: -3 -4 -5^1 +6 $+7^1$

1: eager +3

TC2: +3 -4 -5^1 -6 -8 $+9^1$ $+9^2$

2: eager +4

TC3: +3 +4 $+5^1$ -5^2 -6 -8 -9^1 -10^1

3: infeas -5^1 … $+10^1$
2: infeas $+5^1$ … -9^1, eager -9^2

TC4: +3 -4 -5^1 -6 -8 $+9^1$ -9^2 -10^1

4: infeas $+10^1$
1: infeas -6, eager -7^1

TC5: -3 -4 -5^1 +6 -7^1 -7^2 $+7^3$

5: eager $+7^2$

TC6: -3 -4 -5^1 +6 -7^1 $+7^2$

5: eager -7^3

TC7: -3 -4 -5^1 +6 -7^1 -7^2 -7^3

1: hopeful $+5^1$ to cover $+5^2$ or +8 or $+10^1$

TC8: -3 -4 $+5^1$ -5_2 +6 -7^1 $+7^2$

8: *no extra coverage*, infeas $+5^2$, -6
1: hopeful +4

TC9: -3 +4 -5^1 -6 -8 -9^1 $+10^1$ -10^2

9: infeas $+5^1$, +6, +8, eager $+10^2$

TC10: -3 +4 -5^1 -6 -8 -9^1 $+10^1$ $+10^2$

Learn conflicts between branches

Because of redundancy in EP Tree,
different flipped prefixes may be infeasible for the same “reason” :
ie. have in common a set of mutually inconsistent branches

Inspired by sat-solving, lifted to branches in EP Tree
[Delahaye, Botella, Gotlieb, ICST 2010]
[Gomes do Val, Master Thesis, Univ British Columbia 2014]

Conflict : ordered set of branches which never occurs in a feasible path

Only flip if no known conflict in flipped prefix (if eager)
or else in flipped prefix + uncovered target branch

How we learn a conflict from an infeasible path

Project constraints from branches and assigns backwards from path end
-> branch seqences with their path-based *weakest precondition (WP)*
WP ~ path predicate for the sequence

At each branch, ie. new sequence, check satisfiability of WP
If sat: memorise sequence and WP and continue
If unsat -> *Minimum Infeasible Sequence (mis)*

…

Ex. conflict

9. else if ((t == 1) && (i+j > k)) t = 2;
10. else if ((t == 2) && (i+k > j)) t = 2; else t = 4;

infeas: $+3\ -4\ -5^1\ -6\ -8\ +9^1\ -9^2\ +10^1$

feas: $-9^2\ +10^1$ $(i_q+j_p) \leq k_r \wedge t_n = 2$

mis: $+9^1\ -9^2\ +10^1$ $t_n = 1 \wedge (i_q+j_p) \leq k_r \wedge t_n = 2$

How we learn a conflict from an infeasible path

Project constraints from branches and assigns backwards from path end
-> branch seqences with their path-based *weakest precondition (WP)*
WP ~ path predicate for the sequence

At each branch, ie. new sequence, check satisfiability of WP
If sat: memorise sequence and WP and continue
If unsat -> *Minimum Infeasible Sequence (mis)*

Remove successive constraints from WP (and corresponding branches from mis) and check satisfiability -> ordered branches of *Unsat Core*

…

Ex. conflict

9. else if ((t == 1) && (i+j > k)) t = 2;
10. else if ((t == 2) && (i+k > j)) t = 2; else t = 4;

infeas: $+3\ -4\ -5^1\ -6\ -8\ +9^1\ -9^2\ +10^1$

feas:	$-9^2\ +10^1$	$(i_q+j_p) \leq k_r \wedge t_n = 2$
mis:	$+9^1\ -9^2\ +10^1$	$t_n = 1 \wedge (i_q+j_p) \leq k_r \wedge t_n = 2$
unsat core:	$+9^1\ \ +10^1$	$t_n = 1 \wedge t_n = 2$

How we learn a conflict from an infeasible path

Project constraints from branches and assigns backwards from path end
-> branch seqences with their path-based *weakest precondition (WP)*
WP ~ path predicate for the sequence

At each branch, ie. new sequence, check satisfiability of WP
If sat: memorise sequence and WP and continue
If unsat -> *Minimum Infeasible Sequence (mis)*

Remove successive constraints from WP (and corresponding branches from mis) and check satisfiability -> ordered branches of *Unsat Core*

Add back (respecting order in mis) branches from mis which *ensure no interference from assignments* between branches in unsat core

-> the resulting ordered set of branches is a *conflict*

Ex. conflict

9. else if ((t == 1) && (i+j > k)) t = 2;
10. else if ((t == 2) && (i+k > j)) t = 2; else t = 4;

infeas: $+3\ -4\ -5^1\ -6\ -8\ +9^1\ -9^2\ +10^1$

feas:	$-9^2\ +10^1$	$(i_q+j_p) \leq k_r \wedge t_n = 2$
mis:	$+9^1\ -9^2\ +10^1$	$t_n = 1 \wedge (i_q+j_p) \leq k_r \wedge t_n = 2$
unsat core:	$+9^1\quad +10^1$	$t_n = 1 \wedge t_n = 2$
conflict:	$+9^1\ -9^2\ +10^1$	*$+9^2$ assigns t tested in 10^1*

Instead of hopeful flipping, reuse feasible sequences

If the flipped branch is already covered but some known feasible sequence goes from the flipped branch to some uncovered target branch,
and if appending the (internally consistent) feasible sequence to
the (internally consistent) prefix
gives a path with no known conflicts
then try applying the WP of the feasible sequence
instead of the flipped branch condition

ex.

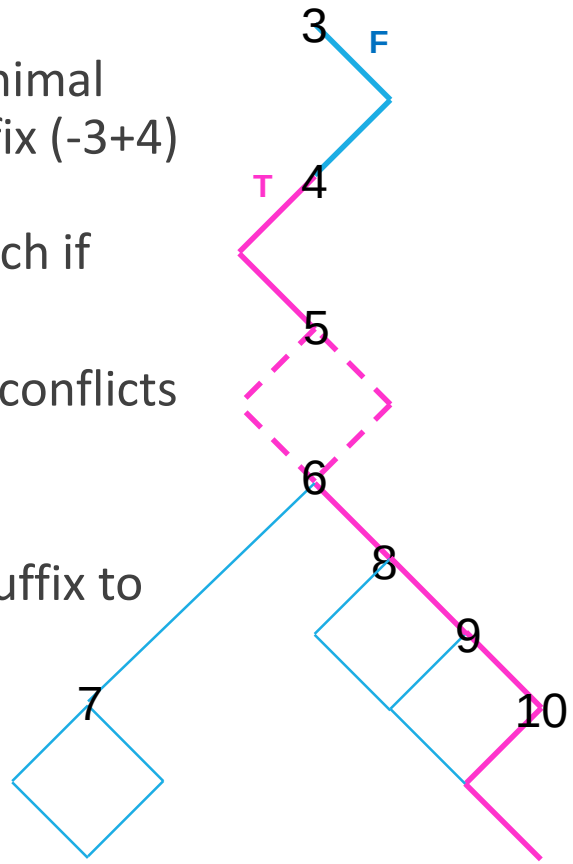
infeas and mis:	$+3\ +4\ +5^1\ -5^2\ -6\ -8\ -9^1\ +10^1$
feasible seq:	$-9^1\ +10^1$
feasible seq:	$-8\ -9^1\ +10^1$
…	
feasible seq:	$+5^1\ -5^2\ -6\ -8\ -9^1\ +10^1$

covered path: $-3\ -4\ -5^1\ +6\ +7^1$
try: $-3\ -4\ +5^1\ -5^2\ -6\ -8\ -9^1\ +10^1$

Instead of hopeful flipping, synthesise new suffixes

If flipped branch (eg.+4) is already covered but could lead to some uncovered target (eg.+10) then

1. try to construct conflict-free *skeleton* : minimal branches to reach target from flipped prefix (-3+4) while avoiding known conflicts
   eg. for CFG diamond (5), only choose branch if necessary to avoid conflict
2. try to find intermediate branches with no conflicts
   eg. choose branch in CFG diamond
   -> path with no known conflicts
3. project constraints back down new path suffix to check satisfiability
   -> WP of new feasible path suffix
4. try applying the WP of the new suffix instead of the flipped branch condition



Summary of the TreeFrog method

For each branch in each covered path

- if opposite branch uncovered and no known conflict in flipped prefix
  then try to eagerly flip and if flipped prefix infeasible
  then extract and memorise a conflict and feasible sequences + wp

- if opposite branch covered and all eager flips tried
  and known feasible sequence to some uncovered branch
  and no known conflict in flipped prefix + feasible sequence
  then try wp of feasible sequence and if resulting path infeasible
  then extract and memorise a conflict and feasible sequences + wp

- if opposite branch covered and all eager flips tried
  and conflict-free path to an uncovered branch includes flipped prefix
  then try wp of new suffix and if suffix or resulting path infeasible
  then extract and memorise a conflict and feasible sequences + wp

- if no solver timeout, remaining uncovered branches are unreachable

Example of the TreeFrog method

TC1: -3 -4 -5^1 +6 $+7^1$

Same eager flipping to generate TC2-TC7 as for concolic method

…

TC4: +3 -4 -5^1 -6 -8 $+9^1$ -9^2 -10^1

…

Then instead of concolic method:

TC8: -3 -4 $+5^1$ -5^2 +6 -7^1 $+7^2$
TC9: -3 +4 -5^1 -6 -8 -9^1 $+10^1$ -10^2
TC10: -3 +4 -5^1 -6 -8 -9^1 $+10^1$ $+10^2$

1: hopeful $+5^1$
8: *no extra coverage*, 1: hopeful +4
9: eager $+10^2$

TreeFrog does:

infeas: -3 -4 *$+5^1$* *$+5^2$*
infeas: -3 -4 *$+5^1$* *-5^2* *-6 -8* *-9^1* *$+10^1$*
infeas: -3 +4 $+5^1$ -5^2 -6 -8 -9^1 $+10^1$
infeas: -3 +4 -5^1 -6 +8
TC8 : -3 +4 -5^1 -6 -8 -9^1 $+10^1$ $+10^2$
TC9: -3 +4 -5^1 -6 -8 -9^1 $+10^1$ -10^2

1: feasible sequence from $+5^1$
1: feasible sequence from $+5^1$
1: feasible sequence from +4
1: new suffix from +4
1: new suffix from +4
8: eager -10^2

Results on other examples

Tcas : 80 branches, 1 non-trivially unreachable

strategy	cases	flipped prefixes
Concolic	395	411
TreeFrog	19	20

Complex : 270 branches, 2 non-trivially unreachable

strategy	cases	flipped prefixes	execution time (mins)
Concolic	424839	424838	30
TreeFrog	90	248	3

Future work

TreeFrog currently implemented as an experimental option of PathCrawler

Does not treat much of C, notably loops, arrays, pointers, function calls
These necessitate

- controlled generation of suffixes including loops
- more precise backward substitution

… using techniques developed for static analysis?

Then TreeFrog can be evaluated on more examples to find whether

- *pruning of the search space effectively counteracts path explosion*
- *checking wp satisfiability becomes too expensive*