Test Generation Strategies to Measure Worst-Case Execution Time

Nicky Williams, Muriel Roger
Worst Case Execution Time

Needed to schedule real-time software
Difficult to find even for sequential uninterrupted module
Naive solution:
sum execution times instructions longest path(s)
but:
software more and more complex, number of paths ↑
anticipation mechanisms microprocessor architectures
(data cache, branch prediction,..) => execution time
depends on context, cache miss >> simple instruction
WCET: two alternative approaches

1. **Static analysis:**
   complex
   specific to one microprocessor
   manufacturers may not divulge algorithms

2. **Measurement (on target platform or simulator):**
   which input values?
PathCrawler generates inputs for path coverage

Pathcrawler (ASE’04, EDCC’05) : concolic-type tool generating input data for path coverage of ANSI C code

No approximation (unlike CUTE, PeX,..) :
100% feasible path coverage if
- no constraint resolution timeout
- absence constructions not treated yet (pointer casts)

Uses Constraint Logic Programming: built-in backtrack
Measuring the execution time of each path

Measuring execution time of each feasible path in the source code will ensure we measure the longest effective execution time (~WCET) if

- Each feasible source code path corresponds to only one binary code path
- Path execution time is the same for all inputs activating the path
- We can put the machine in some initial worst state (cache, branch prediction registers)
- No influence from bus, DRAM refresh…
Measuring fewer paths: partial orders

Many real-life modules have too many feasible paths.

Don’t know the *longest* path but define partial orders depending on hypotheses on anticipation mechanisms which are weak enough to apply in many cases.

ex. only difference between 2 paths is that one has more identical loop iterations than the other.

ex. only difference between 2 paths is that one takes empty branch of if-then-else.
If-then-else partial order

Empty ITE branch contains no data references

ITEE = ITE with an empty branch

Partial order: path $P_i$ longer than path $P_j$ if the only difference between them is that
$P_i$ replaces at least one empty ITE branch in $P_j$ by a non-empty branch

Restrictions:
1) if data cache, don’t apply if a pointer assigned in ITE is dereferenced afterwards
2) doesn’t take branch prediction into account
Properties of modified strategy

• Generates inputs to cover all maximal paths

• Reduces (or, at worst, does not increase) generation of inputs to cover non-maximal paths

• Reduces (or, at worst, does not increase) unnecessary exploration of execution path tree
Example: source code and CFG

```c
f(int x, int y, int z){
    int r = -2;
    if (x > 5)
        r = x;
    if (y > 4)
        y = 4 - y;
    else
        z = 0;
    if (r <= z)
        z = -5;
    if (r >= (z + y))
        r = r + 1;
    else
        a = a - 1;
    return r; }
```
CFG and tree of execution paths

16 paths
Default strategy: arbitrary 1st path covered
Depth-first search: new path covered
Depth-first search: try to cover partial path
Infeasible partial path : tree pruned
Depth-first search: try to cover partial path
Left-right-non-determinist ...
How to modify the strategy for ITE partial order

Goal: optimise generation but cover maximal paths

Mechanisms:

- *Abstract partial path* = each ITEE replaced by abstract ITEE containing empty and non-empty branches
- *Memorise* feasible paths & abstract infeasible partial paths
- DFS modified to explore non-empty ITEE branches first => discover all maximal infeasible partial paths first
- LR non-determinism modified to explore maximal paths: on backtrack, if partial path ends in empty ITE branch then only cover *continuations* whose abstraction was infeasible
CFG and tree of abstract execution paths
Partial order on execution paths
Modified strategy: arbitrary 1st path covered
Try to cover 1st non-empty ITEE branch
Path covered has 2nd empty ITEE branch
Try to cover 2nd non-empty ITEE branch
Maximal path covered
Depth-first search: next maximal path infeasible
Try uncovered path with 2nd empty ITE branch
Infeasible with 2nd empty ITE branch too
Depth-first search
Path covered has 2nd empty ITE branch
Paths with 2nd non-empty ITEE branch infeasible
Depth-first search: other path infeasible
Infeasible with other empty ITE branch too
Maximal feasible path has 2 empty ITE branches
Path with other empty ITE branch is feasible too
All maximal feasible paths covered : stop here
Results of modified strategy

Example of industrial embedded real-time code:
1512 lines commented code
89 ITE, of which 45 with an empty branch, but no loops
default strategy: 846975 cases in several days
modified strategy: 6554 cases in one day

MaSCoTE project partner Geensys set up test rig to execute tests on a HCS12X simulator and measure execution time: the path with the longest execution time in the default set was also covered in the modified set
Future work

- Optimisation in progress: data dependences to avoid trying continuations of partial paths whose infeasibility cannot depend on ITEEs
- Restrictions: use PathCrawler to check no pointers are assigned in ITEE/loop and de-referenced after it
- Other partial orders: multiple conditions, function calls
- Uses of the same mechanisms in other test-generation strategies to treat ex. branch coverage, function calls, …
PathCrawler: process

- source code
  - instrument
  - instrumented code
    - add injection
    - inputs, compile
      - execution path
        - execute
          - substitute inputs
            - path predicate
  - definition domain
  - predicates of tested paths
    - rest of domain
      - difference
        - resolve
          - next inputs
            - inject
              - harness
                - add
                  - next inputs
                    - inject
                      - instrumented code
                        - add injection
                          - inputs, compile
                            - execution path
                              - execute
                                - substitute inputs
                                  - path predicate
Path predicate on input values:
\[ x \leq 5 \land \]
\[ y \leq 4 \land \]
\[ -2 \leq 0 \land \]
\[ -2 < -5 + y \]
\[ (\text{or: } x \leq 5 \land y \leq 4 \land y > 3 ) \]
path predicate on input values:

\[ x \leq 5 \land \]
\[ y \leq 4 \land \]
\[ -2 > 0 \land \]
\[ -2 < -5 + y \]
Second partial order

Pi slower than Pj if their only difference is that for at least one loop all iterations are identical in Pi and Pj and Pi contains at least one iteration and Pi contains more.

Hypothesis: no data cache or else don’t apply if a pointer assigned in the loop is de-referenced afterwards.

More complicated to implement because of multiple sub-conditions in loop-head.

Occurs in many examples of embedded code using interpolation in discretised graphs coded as arrays.
Default PathCrawler strategy (LR non-determinist DFS)

cover_all (P, i, PP) = 
if B_{iP} is_last_branch_in P then cover_subtree (PP:B_{iP}, i)
else 1st pass : cover_all (P, i+1, PP:B_{iP})
    on backtrack : cover_subtree (PP:B_{iP}, i)  init: PP=\emptyset, i=1

cover_subtree (PP, i) =
if gen_test (PP) = P'
   then if B_{iP} is_last_branch_in P'
      then backtrack
   else cover_all (P', i+1, PP)
else backtrack

P = execution path
i = branch number
PP = partial path
B_{iP} = branch i path P
B_{iP} = opposite branch
: = append branch
Modified strategy for ITEE partial order: intro

- **cover_max** replaces **cover_all**
- **cover_max_subtree** and **cover_rest_subtree** replace **cover_subtree**
- **feas** = set of feasible paths
- **infeas** = set of abstract infeasible partial paths
- special treatment for
  - branches starting non-empty ITEE paths
  - branches which are empty ITEE paths
Modified strategy for ITEE partial order (1)

APP = abstract partial path, · = append to APP

cover_max (P, i, PP, APP) =

if BiP is_last_branch_in P then
  if BiP st_nonempty_ITEEpath then bt
  else cover_max_subtree (PP:B_iP, APP·B_iP, i)
else if BiP st_nonempty_ITEEpath then
  1st: cover_max (P, i+1, PP:B_iP, APP·BiP)
  bt: cover_rest_subtree (PP:B_iP, APP·B_iP, i)
else if BiP is_empty_ITEEpath then
  if gen_test (PP:B_iP) = P' then
    1st: feas := feas U P' ; if BiP' is_last_branch_in P' then bt
    else cover_max (P', i+1, PP:B_iP, APP·B_iP)
    bt: cover_rest_subtree (i, PP:B_iP, APP·B_iP)
  else infeas := infeas U APP·B_iP ; cover_max (P, i+1, PP:B_iP, APP·B_iP)
  else
    1st: cover_max (P, i+1, PP:B_iP, APP·B_iP)
    bt: cover_max_subtree (PP:B_iP, APP·B_iP, i)
Modified strategy for ITEE partial order (2)

\[
\text{cover\_max\_subtree}\ (PP, \ APP, \ i) = \\
\begin{array}{l}
\quad \text{if} \ \text{gen\_test}\ (PP) = P' \ \text{then} \ \text{feas} := \text{feas} \cup P' ; \ \text{if} \ B_{i'P'} \ \text{is\_last\_branch\_in} \ P' \\
\qquad \quad \text{then backtrack} \\
\quad \text{else} \ \text{cover\_max}\ (P', i+1, PP, APP) \\
\end{array}
\]

\[
\quad \text{else} \ \text{infeas} := \text{infeas} \cup APP ; \ \text{backtrack}
\]

\[
\begin{array}{l}
\text{cover\_rest\_subtree}\ (i, PP, APP) = \\
\quad \text{foreach APPext} \in \text{infeas} \{ \text{foreach\_in\_order PPext concretises APPext} \{ \\
\quad \quad \text{if} \ PP \ \text{is\_a\_prefix\_of} \ PPext \ \text{then} \\
\quad \quad \quad \text{if not slower\_feas\_paths}\ (Pext, APPext) \ \text{then} \\
\quad \quad \quad \quad \text{if} \ \text{gen\_test}\ (PPext) = P' \ \text{then} \ \text{feas} := \text{feas} \cup P' ; \ \text{if} \ B_{iP'} \ \text{is\_last\_branch\_in} \ P' \\
\quad \quad \quad \quad \quad \quad \text{then backtrack} \\
\quad \quad \quad \quad \quad \quad \text{else} \ \text{cover\_max}\ (P',i+1,PP',APP) } \\
\quad \quad \text{else backtrack } \} \} \\
\end{array}
\]
Modified strategy for ITEE partial order (3)

\[
\text{slower_feas_paths} \ (P_{ext}, \ APP_{ext}) \ if \\
\ exists \ P_{ext}', \ PP_{ext}' \ such\_that \\
\ P_{ext}' \ \epsilon \ feas \\
and \ PP_{ext}' \ is\_a\_prefix\_of \ P_{ext}' \\
and \ PP_{ext}' \ concretises \ APP_{ext} \\
and \ P_{ext}' \ \text{slower\_than} \ P_{ext} \ )
\]