WCET measurement using modified path testing

Nicky Williams

Hybrid approaches to WCET measurement

Increasing demand for modern micro-processors for real-time applications

but they use mechanisms which make the execution time of an instruction variable and

modelling the micro-architecture is difficult and details may be confidential

⇒ direct measurements of execution time on target architecture

⇒ static analysis to limit the number of measurements

Our approach to WCET measurement

Other hybrid approaches propose using static analysis to **decompose** the program then perform direct measurements of program **fragments**

We propose to measure the time to execute the **whole** program but use static analysis to cut the **number of whole program runs** measured

WCET measurement using path testing

Path testing =

run program on 1 test case (set of input values) for each feasible execution path in the source code

With a test-harness to run the program on each test case and a way to measure the execution time, path testing guarantees measurement of the WCET under certain hypotheses :

1st hypothesis

One to one correspondence between execution paths in source (or assembler) code and in binary code

May need to disable certain compiler options ?

For operations such as division, square root the number of iterations can depend on operand values (but can correct for this)

2nd hypothesis

A given path always has the same execution time (from a given initial machine state)

i.e. execution time of an operation may depend on the state of the machine but not on operand values

i.e. micro-architectural events are deterministic for a given path

Attn: cache behaviour if the path contains references to elements of a very large data structure which may be close together or very far apart

3rd hypothesis

We can define the worst possible initial machine state for each execution path and set the test-bed to this state

- **worst possible state of caches = full of "useless" data: we must devise a program to initialise them**
- **branch prediction ?**
- **in the case of cyclical programs, initial state depends on previous program run ...**

PathCrawler : automatic generation of path test cases

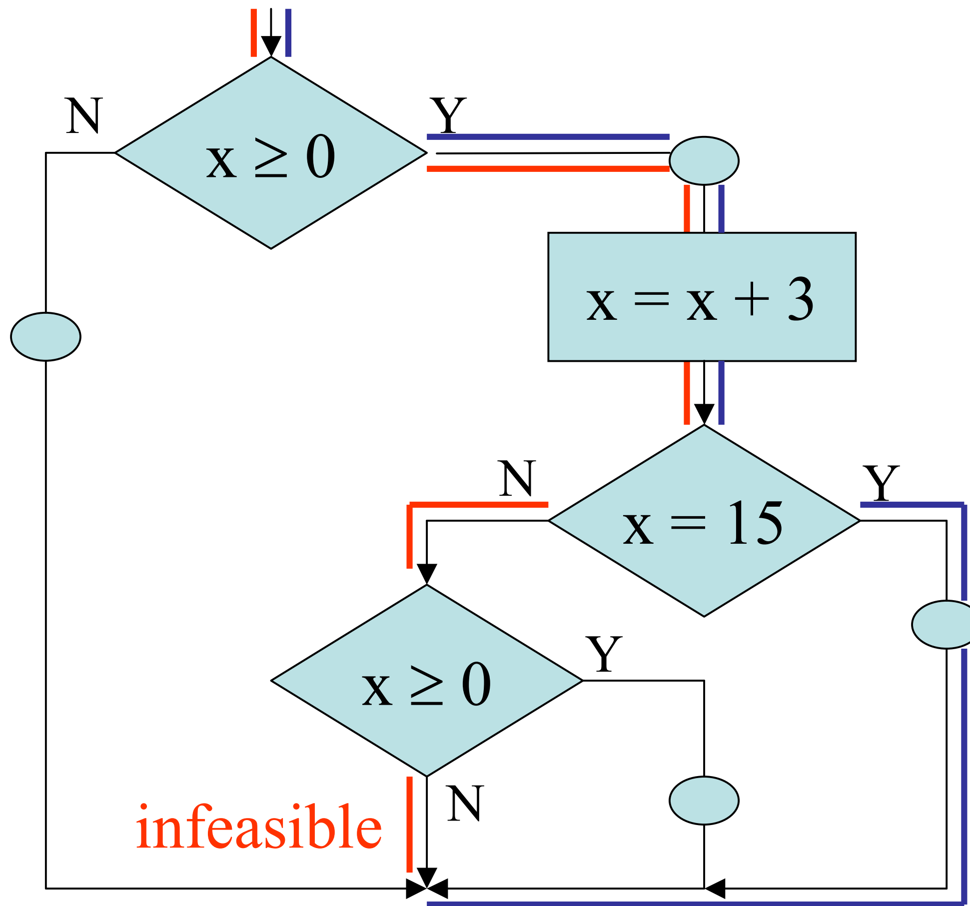
Inputs : source code and definition domain of program
Output : tests covering 100% feasible execution paths

An original approach :

- use tests to iteratively construct execution path tree
- only analyse individual feasible (complete) paths with unrolled loops (can take aliases into account)
- use the structure of covered paths to search efficiently for the next test-case using constraint logic programming

Application : imperative languages, sequential software
prototype for C
status: imprecision current treatment floats
recursive functions, union not treated yet

Paths and path predicates




```
if (x >= 0)
   y = 2;
   x = x + 3;
   if (x = = 15)
      y = y + 1;
   else
      if (x >= 0)
         y = y+2;
else
   y = 1;
```

$X \geq 0 \wedge (X + 3) = 15$

PathCrawler: iterative generation process

source code

instrument

instrumented code

add injection inputs, compile

driver

execute

execution path

substitute inputs

path predicate

add

predicates of tested paths

definition domain

difference

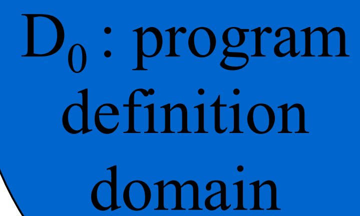
rest of domain

resolve

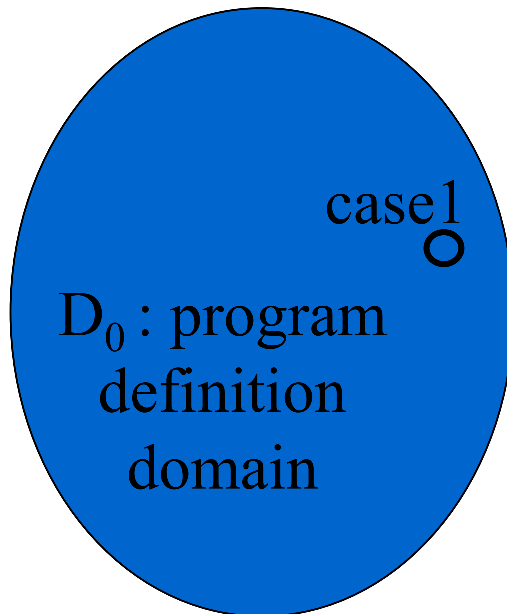
next inputs

inject

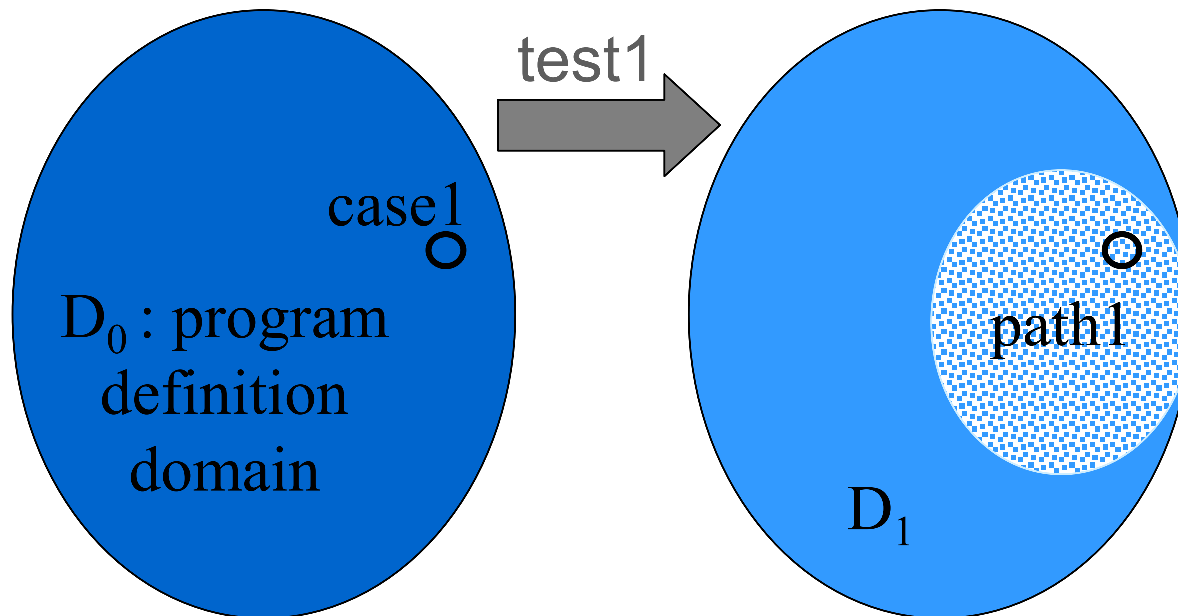
PathCrawler : input domains



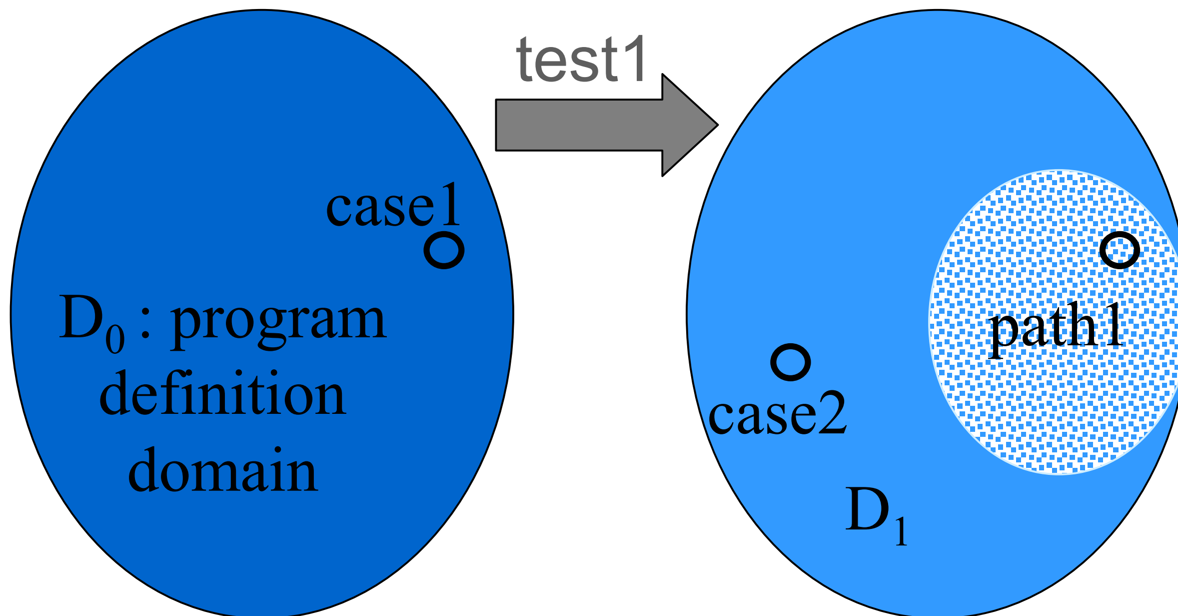
PathCrawler : input domains



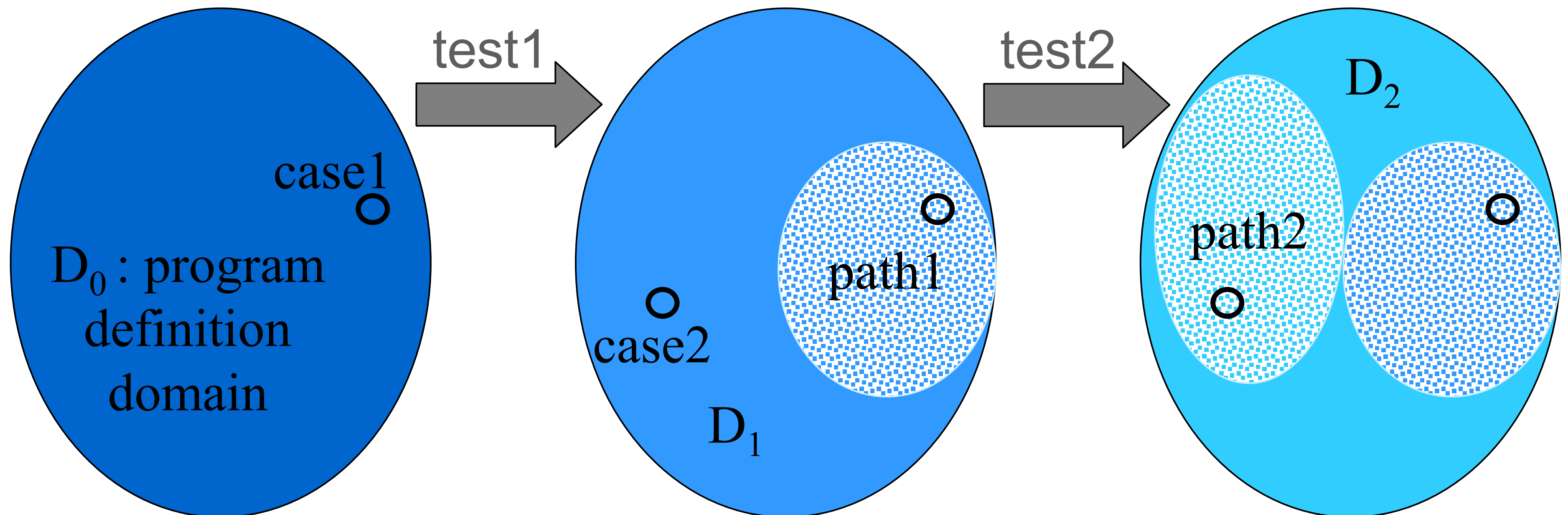
PathCrawler : input domains



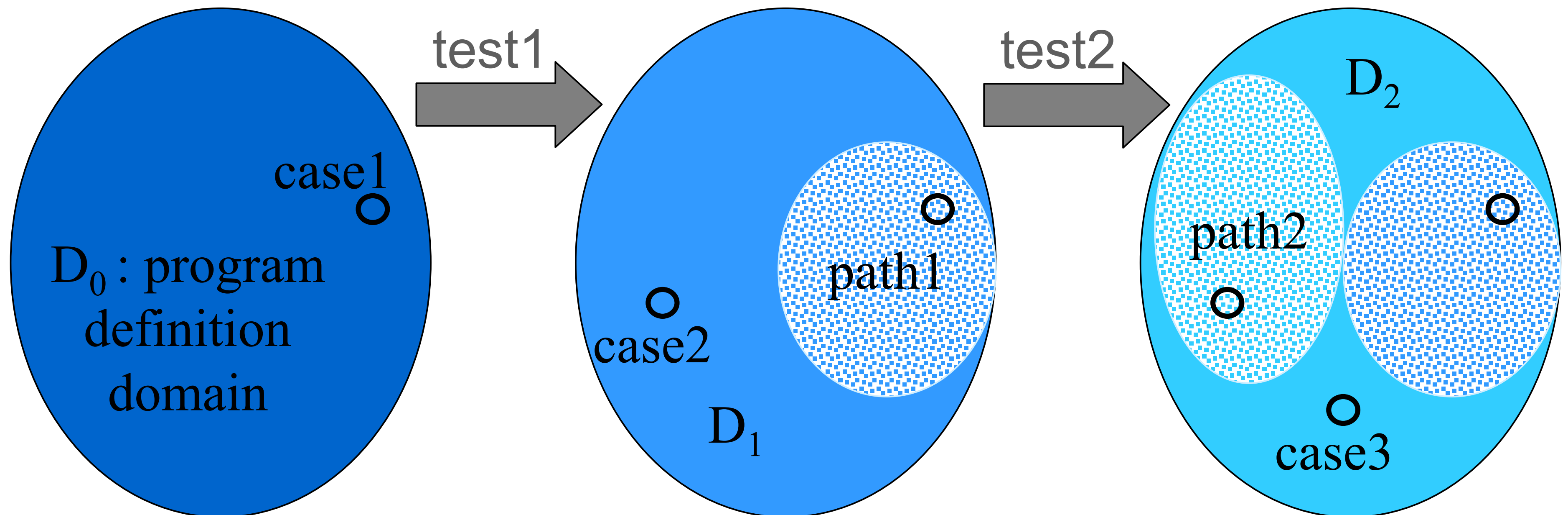
PathCrawler : input domains



PathCrawler : input domains



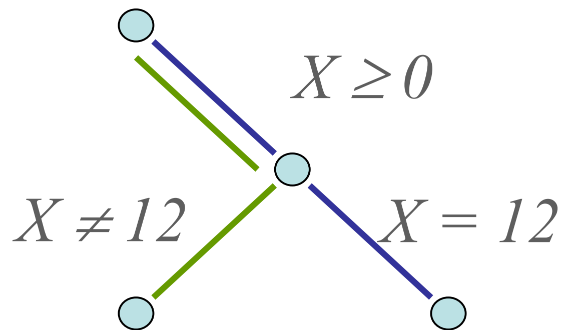
PathCrawler : input domains



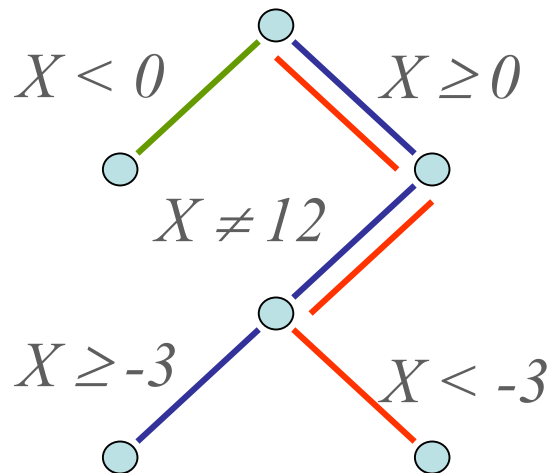
Default strategy for constraint solving/path selection

solve prefixes with last condition negated, depth-first

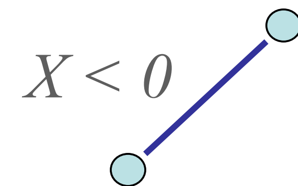
case 1: x = 12



case 2: x = 5



case 3: x = -2



path predicate — selected prefix — infeasible

Complexity

For each negated prefix,

- **Solution** of constraints (generation of new test case) or detection of **infeasibility** (of ALL paths with this prefix) is potentially **NP-complete** for integer values

- But constraint **propagation** and the **heuristics** used to enumerate remaining solutions reduce complexity in practice

How to measure fewer paths

Define a partial order on paths in control flow graph

e.g. path p *contains a subset of path* q's *variable accesses and operations, in the same order*

=> *execution time of p < execution time of q*

Modify path selection strategy to favour generation of longest paths first

Then only generate tests for new paths which may be longer than those already generated

e.g. loops with variable number of iterations

If paths *p* and *q* are identical before and after the loop and for each iteration

then the path, *q*, with more iterations

has the longest execution time

⇒ don't generate paths identical to already measured path except for fewer loop iterations

⇒ explore all branches after the loop in *q*, then

in *p*, only explore negated prefixes which were unsatisfiable in *q*

Other possibilities ?

trivial differences between paths which contribute to the combinatorial explosion in the number of paths

e.g. if a > b then max = a else max = b

(but branch prediction effect ?)

Conclusion : gradual approach

test generation strategy

	simple	caches	branch prediction	other ?
other ?				
less some paths with few iterations			next version ◆	
all-paths	current prototype ◆			

micro-architecture