

Coverage-Based Random Testing via Uniform Generation of Combinatorial Structures



Presented by: Frédéric Voisin

LRI, Univ. Paris-Sud & CNRS

Joint work with

Marie-Claude Gaudel

Sandrine-Dominique Gouraud

Alain Denise



Introduction

- ◆ There are many ways for testing programs
 - Dynamic vs Static testing
 - Unit vs Integration vs System testing
 - Structural vs Functional vs Randomized testing

With some recurrent problems (considering only the problem of « test generation »)

- How to generate « relevant » test sets ?
- How to assess the « quality » of a given test set ?

*Using **coverage criterion** is one way to answer that question but computing solutions by hand is a nightmare, and can yield small test sets.*

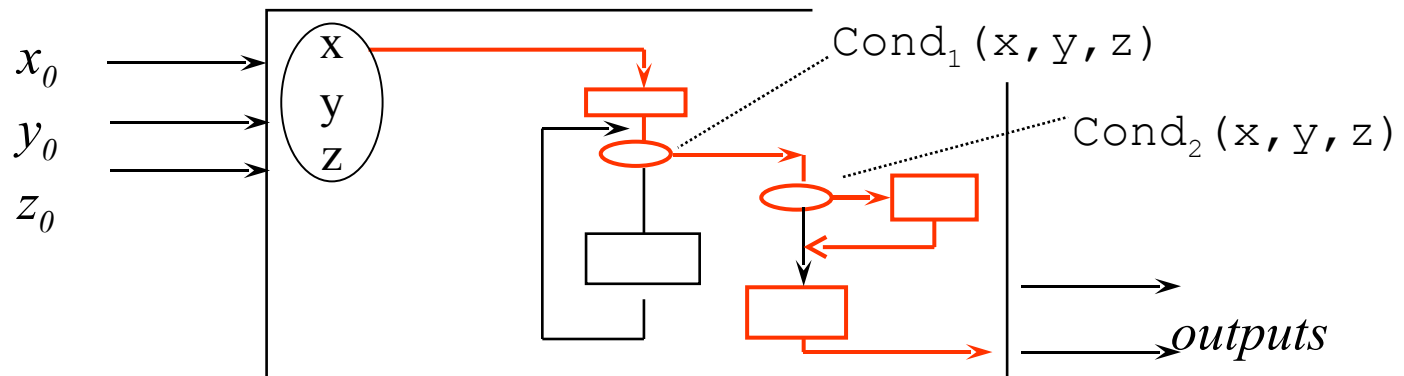
- Small test sets are usually not very good at finding defaults, but large test sets are not always better !

Randomized testing is good for generating large test sets but does not imply a good coverage ...



Coverage Criterion for Structural Program Testing

- ◆ Each criterion defines « relevant » sets of paths that fulfill the criterion
- ◆ Either one checks if a given set of paths is a solution to the criterion or one generates solutions to the criterion before trying to find values for them



one path = one logical expression over $x_0, y_0, z_0 = \underline{\text{zero}}$, one (or several) test case ?

$$\neg Cond_1(x_0, y_0, z_0) \wedge Cond_2(x_0, y_0, z_0)$$

How to go from a « target » path to the input values for traversing that path ?

- some path can be unfeasible (not an execution path)

- the problem of deciding if a path is feasible is undecidable



Some typical coverage criteria (among dozens)

- ◆ **All-paths:** *coverage set is the set of all paths = exhaustive testing !*
 - *Infinite as soon as there exists a loop in the program*
 - ◆ **All-k-paths:** all paths up to a given length k = based on some *regularity hypothesis*
 - Finite but sometimes too many paths to consider when k is large
 - ◆ **All vertices:** each vertex (or each squared node) is visited at least once along a path.
 - ◆ **All edges :** Each edge is visited at least once along a path
 - ◆ + several criteria for handling boolean operators controlling conditionals or loops
- ☞ Criteria have been adapted for data flow graphs, for function call graphs, for object-oriented languages, etc.
- ☞ Many criteria apply to any kind of « paths » in « graphs », not only to execution path in control flow graphs



Statistical or Random testing in the literature

- ◆ Generally speaking : “*drawing test data from the input domain, following some probability distribution*”

Given `int f(int x, int y)` draw initial values for `x` and `y` in their domains

- ◆ “Random” testing : select test data **uniformly** at random from the input domain of the program
- ◆ “Statistical” testing : select test data at random, **based on an operational** profile (e.g. one “knows” that there should be twice as many even values as odd values for `x`)

- ☞ It is possible to **test more intensively** than with deterministic methods : generate as many `(x, y)` pairs as you want (assuming you always know the expected output)
- ☞ **Bad coverage** of particular cases, like rare cases: for instance, for some optimization purpose, `f` might start with

```
if (x == y) { ... } else { ... }
```

The `else` part will probably be more tested than the `then` part.

- ☞ generating new test cases “blindly” will not ensure the traversal of a given path.



Another example with a loop

if $x^2 + y^2 < 1$ then $r \leftarrow \sqrt{x^2 + y^2}$

else $r \leftarrow \sqrt{\frac{1}{2} \left(\frac{x^2}{x^2 + y^2} + \frac{y^2}{x^2 + y^2} \right)}$ fi ;

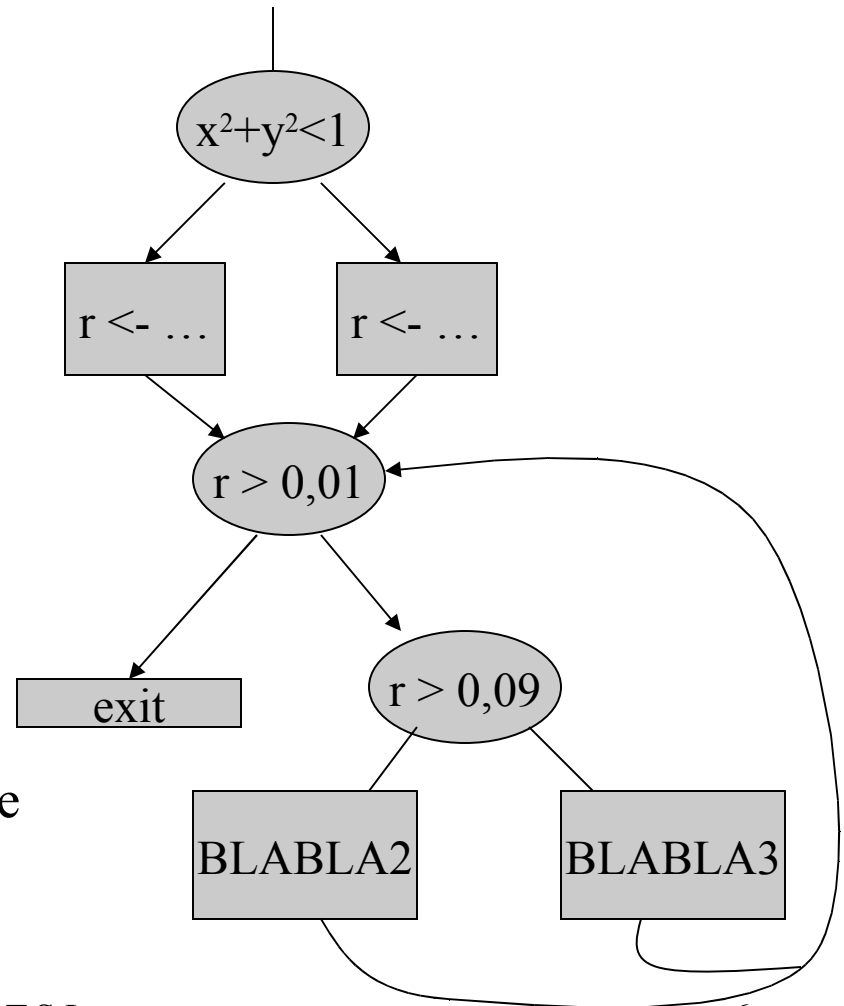
while $r > 0,01$ do

 if $r > 0,09$ then BLABLA2 ; $r \leftarrow r - 0,001$

 else BLABLA3 ; $r \leftarrow r - 0,099$ fi

done

x et y are real numbers



Probably difficult to have a fair handling of the various cases !



More novel : **statistical structural testing !**

- ◆ A coverage criterion is supposed to be given
- ◆ Probability distribution is biased w.r.t. a uniform distribution in order not to under-represent rare cases
- ◆ Any path should have a even chance to be drawn for « all-k-path »
- ◆ Any path has a non-zero chance to be drawn for the other criteria
- ◆ Give a way to automate the processus, as far as possible !

Ideas :

- ◆ Replace uniform drawing on **input values** by uniform drawing **among paths** in graphs (restricted to paths up to a given length, chosen by the tester)
- ◆ Balance the probability of each element (node, edge, ...) in the drawing but still use uniformity over the paths that covers an element.
- ◆ There exists a convenient conceptual and practical tool for that: **combinatorial structures**



About « combinatorial structures »

- ◆ Combinatorial Structures ?
 - either an atomic object
 - or the result of applying an operator to combinatorial objects ...
- ◆ « Décomposable objects » are **uniquely** defined from smaller objects (no equality between objects)
- ◆ Main results : there exists efficient methods for
 - given a set of combinatorial objects and an integer n , uniformly generate one (or a sequence of) objects of size n from this set;
 - counting the number of elements of size n generated from a Combinatorial Structure.



Combinatorial Specifications

◆ A kind of Combinatorial Structure generated with the following rules:

- « empty » objects of size 0 (written 1, or ϵ)
- « atomic objects » of size 1
- The following set of operators:

$+$: *disjoint union*

\times : *product*

Sequence(A): finite sequences of elements from A
(the sequences can be bounded or of fixed length)

Set(A): the set of finite sets of elements from A

Cycle(A): finite cycles made with elements from A

◆ Examples: let F be an atom:

- $A = F + A \times A$ -- complete binary trees with atoms at the leaves
- $A = S + S \times A \times A$ – here the vertices are the atoms.



- ◆ Combinatorial specifications can be viewed as a set of « langages equations » as in Formal Language Theory.
- ◆ Algorithms for drawing are decomposed in two steps :
 - First, counting the number of objects of size n (or $\leq n$)
 - Uniform drawing of objects in the set based on the previous counting
- ◆ Efficient algorithms exist for counting and drawing whenever the combinatorial specification corresponds to an algebraic or rational language.
 - Uniform random generation of paths with a $n \cdot \log n$ complexity + some pre-processing (in $n \cdot |G|^2$, or $n \cdot |G|$)
- ◆ In the sequel, we use only the operators $+$, \times , et $*$ (hence defining a rational language)
- ◆ A « syntactic » extension $\text{seq}(n)$ or $\text{seq}(\leq n)$, **where n is fixed**, is allowed (corresponding to a finite, static, unfolding of the definitions)



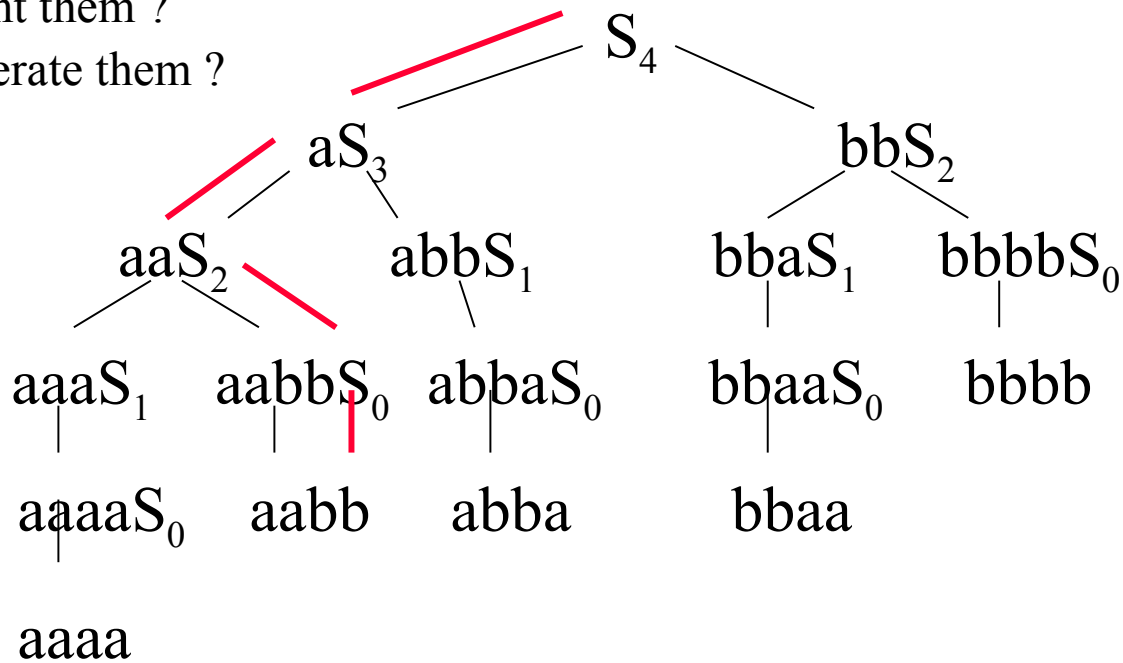
Uniform generation of labelled combinatorial structures ?

- ◆ Underlying theory: **uniform generation of “labelled combinatorial structures”** (Flajolet & al., 1994)
- ◆ Implemented by Alain Denise & al. : **CS, a MuPAD-Combinat package for counting and randomly generating combinatorial structures**
 - <http://mupad-combinat.sourceforge.net/>
 - Current development: Nicolas Thiéry, dept of Mathematics, Orsay

Uniform random generation: a basic example

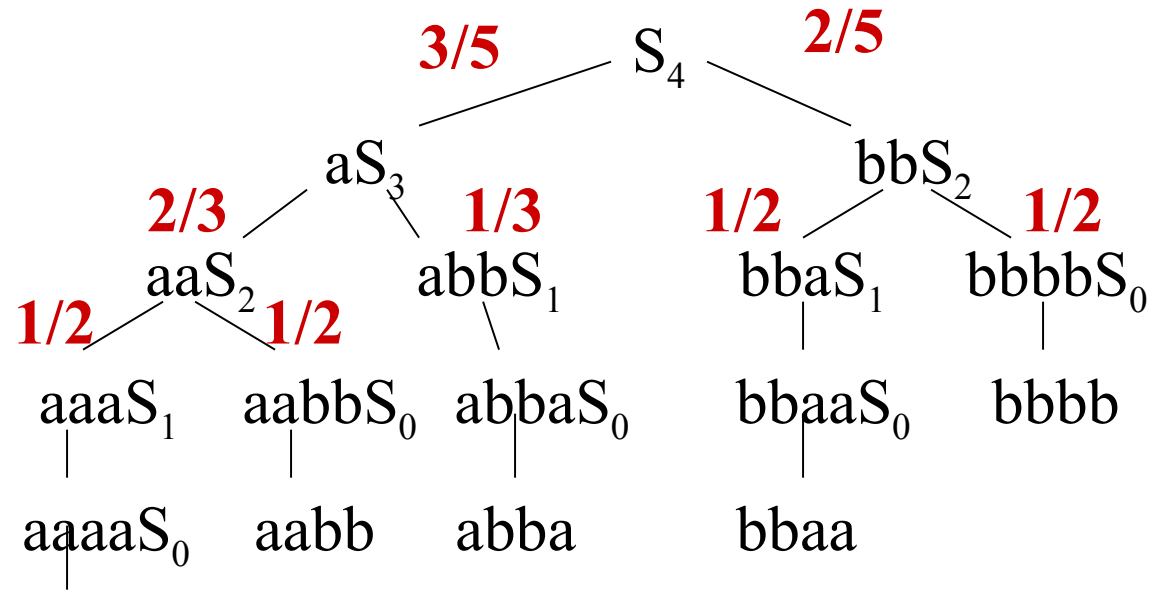


- ◆ Fibonacci language: $L = (a+bb)^*$
- ◆ $S \rightarrow aS \mid bbS \mid \epsilon$
 - **One** possible case of size 4: $S \rightarrow aS \rightarrow aaS \rightarrow aabbS \rightarrow aabb$
 - **All** the cases of size 4:
- ◆ How to count them ?
- ◆ How to generate them ?





Counting and randomising:



aaaa

1 - Counting: $S \rightarrow aS \mid bbS \mid \epsilon \Rightarrow$
 $S(0) = 1 ;$
 $S(1) = 1 ;$
 $S(n) = S(n-1) + S(n-2) \quad (n \geq 2)$



Generating:

- ◆ 2 - Using a computed table S of the $S(k)$, $k \leq n$, call `generate (S,n)` with, in our simple case, the following definition:

```
generate (S, k) {  
  if (k>0) {  
    -- select fairly between  
    -- S->aS (associated number of path:  $S(k-1)$ )  
    -- and S->bbS (associated number of path:  $S(k-2)$ )  
    h ← random integer in [1,  $S(k)$ ]  
    if h ≤  $S(k-1)$  { write("a") ; generate(S, k-1) }  
    else write("bb") ; generate(S, k-2)  
  }  
}
```



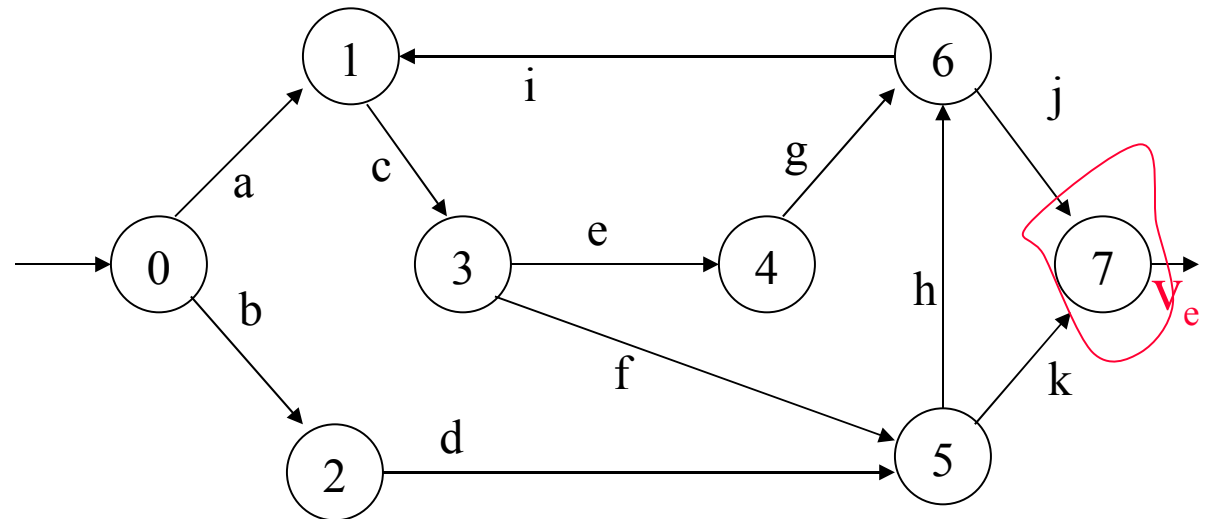
Principles of uniform path generation

- ◆ Given any vertex v , let $f_v(m)$ be the number of paths of length m that connect v to the end vertex v_e
- ◆ we are on vertex v with k successors v_1, v_2, \dots, v_k
- ◆ condition for path uniformity: choose v_i with probability $f_{v_i}(m-1)/f_v(m)$



An example (control graph, transition system, ...)

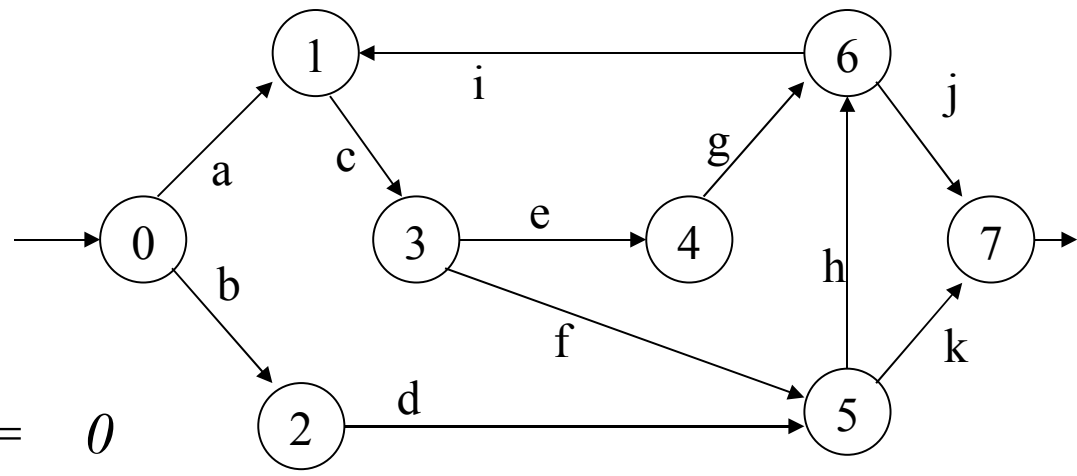
$$\left\{ \begin{array}{l} f_v(0) = 1 \text{ if } v = v_e \\ f_v(0) = 0 \text{ otherwise} \\ f_v(k) = \sum_{v \rightarrow v'} f_{v'}(k-1) \quad \text{if } k > 0 \end{array} \right.$$





Recurrence relations for the $f_v(m)$

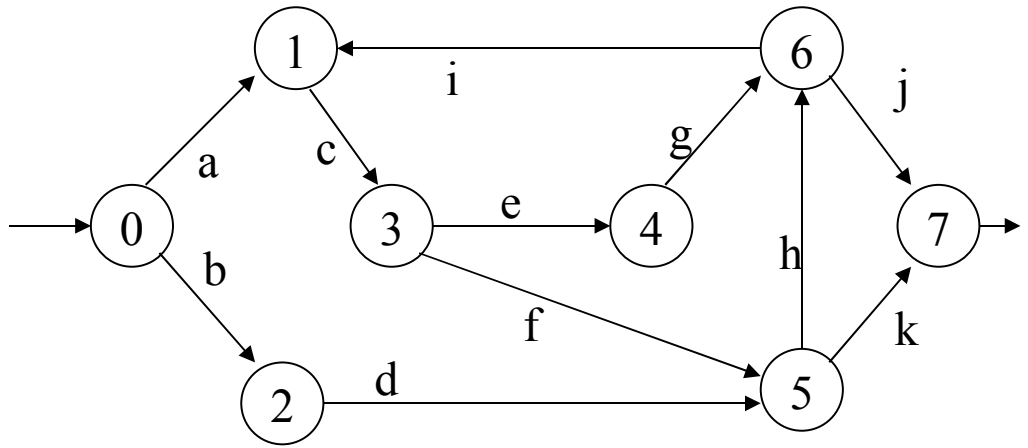
$$\begin{aligned}
 f_0(0) &= f_1(0) = f_2(0) = 0 \\
 f_3(0) &= f_4(0) = f_5(0) = f_6(0) = 0 \\
 f_7(0) &= 1
 \end{aligned}$$



$$\begin{aligned}
 f_0(m) &= f_1(m-1) + f_2(m-1) && (m > 0) \\
 f_1(m) &= f_3(m-1) && (m > 0) \\
 f_2(m) &= f_5(m-1) && (m > 0)
 \end{aligned}$$



Counting and randomising:



1 - Counting:

$$\text{Path}_0(n) = a. \text{Path}_1(n-1) \mid b. \text{Path}_2(n-1)$$

$$\text{Path}_1(n-1) = \dots \quad \text{Path}_2(n-1) = \dots$$

...

$$\text{Path}_7(0) = 1$$

- to be done once
- memory space requirement : $n \cdot |G|$
- $O(n \cdot |G|^2)$ arithmetic operations in the very worst case



Generating from the table:

- ◆ 2 - Using the “Path” table of the $\text{Path}_i(k)$, define the `generate` functions as follows and call `generate0(ϵ , n)`

where (for instance) :

```
generate0(Path, k) {  
  if (k>0) {  
    h ← random integer in [1, Path(0, k)];  
    if (h ≤ Path1(k-1)) { write(a); generate1(Path, k-1); }  
    } else { write(b) ; generate2(Path, k-1); }  
}
```

- ◆ Complexity: $O(n \cdot \log n)$

Control Flow Graph and Combinatorial Structures

Atoms = edges; Sequence of edges = **paths**

Combinatorial structure specification:

$$S = v' \cdot S + v \cdot e_0 \cdot C \cdot e_7$$

$$C = e_1 \cdot e_2 + e_3 \cdot B \cdot e_6$$

$$B = e_4 \cdot I + \varepsilon$$

$$I = e_5 \cdot B$$

Associated recurrence relations for

$T(X, n) = \# \text{ items } X \text{ of size } n:$

$$T(S, 0) = 0$$

$$T(S, n) = T(S, n-1) \text{ if } n < 3$$

$$T(S, n) = T(S, n-1) + T(C, n-3) \text{ otherwise}$$

$$T(C, 0) = T(C, 1) = 0$$

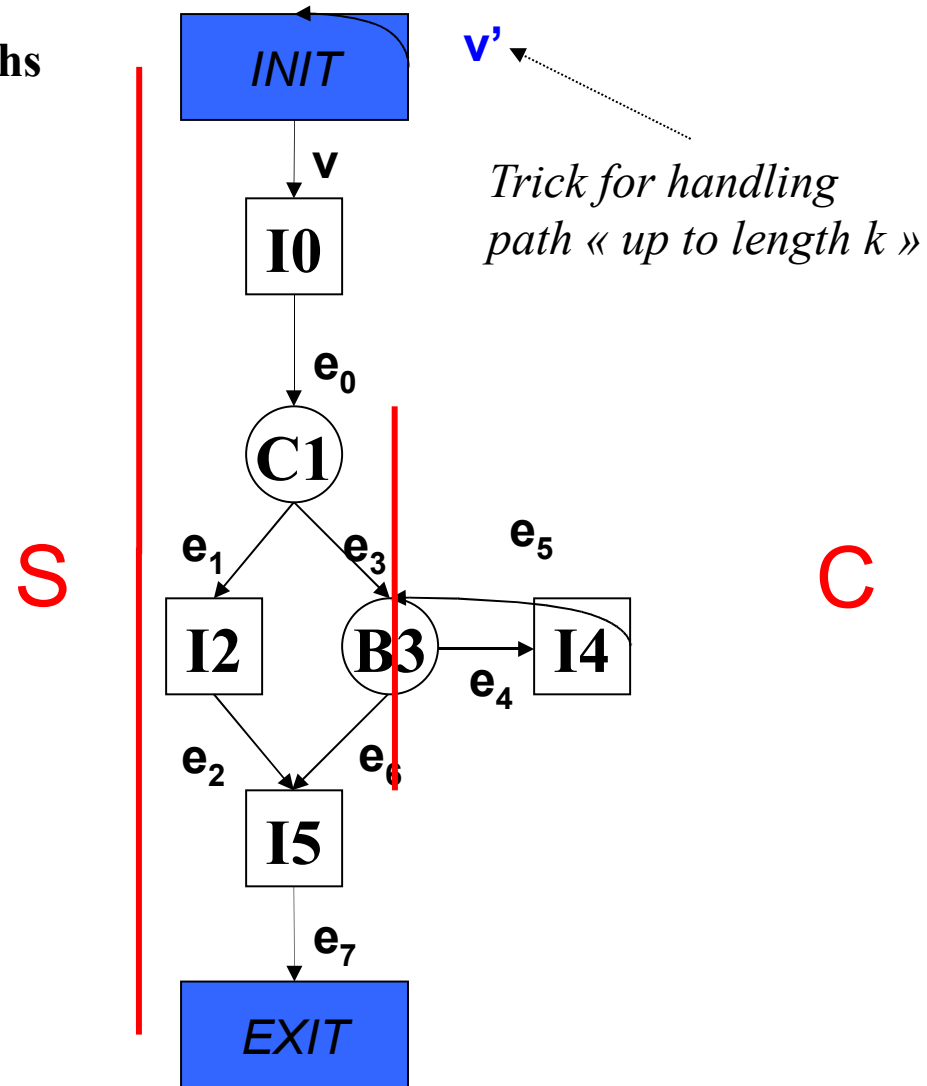
$$T(C, 2) = 1 + T(B, 0)$$

$$T(C, N) = T(N, n-2) \text{ otherwise}$$

...

$$T(B, 0) = 1$$

$$T(B, n) = T(I, n-1) \text{ otherwise}$$



Counting from the recurrence relations



length	0	1	2	3	4	5	6	7	8	9
C	0	0	2	0	1	0	1	0	1	0
B	1	0	1	0	1	0	1	0	1	0
I	0	1	0	1	0	1	0	1	0	1
S	0	0	0	0	0	2	2	3	3	4

In S, there are 3 paths of length (\leq) 7, one coming from an extension to T(C, 4) and two coming from T(S, 6)



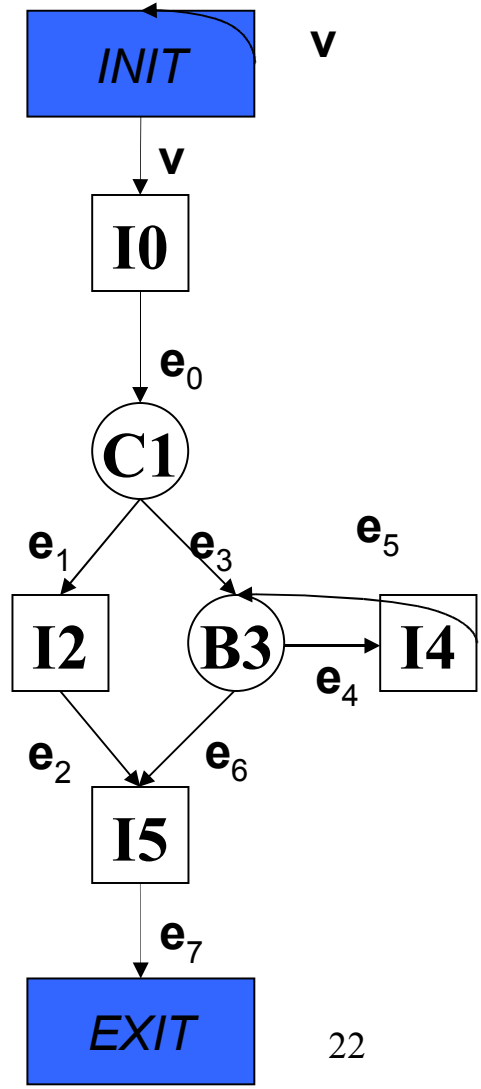
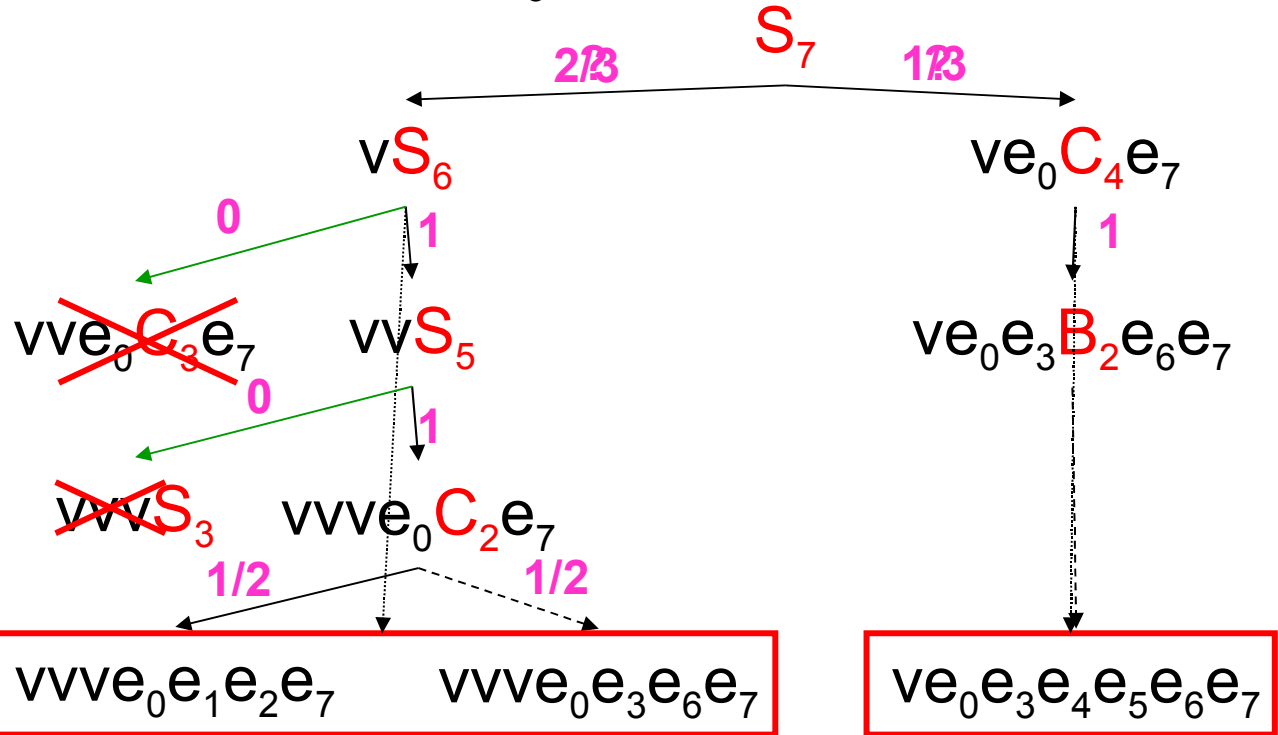
Drawing (Length ≤ 7)

$$S = v.S + v.e_0.C.e_7$$

$$C = e_1.e_2 + e_3.B.e_6$$

$$B = e_4.I + \varepsilon$$

$$I = e_5.B$$





Where things get harder in practice ...


- ◆ The drawing is based on the flow graph (via the Combinatorial Structure)
 - Not all the paths are execution paths
 - It is undecidable to know if a given path is feasible or not, hence we have to rely on constraint solving (of the path predicate) to find actual input values, if any.

Heuristics for detecting unfeasible paths ?

How to avoid drawing unfeasible paths ?

What do we do if a path is coined as unfeasible ? Or if we don't know ?

- ◆ Not all coverage criteria have a straightforward translation as a problem of a uniform drawing of paths. How to adapt the method to such criteria ?



Generation of Input Data

- ◆ For each path drawn, compute the “path predicate”, expressed as a constraint solving problem
- ◆ Resolution of each path predicate if possible
- ◆ **Constraints resolution library** based on the ECLiPS^e system + powerful heuristics to deal with resolution failures (cf. LOFT and GATEL systems by Bruno Marre)
- ◆ “Lazy” constraint propagation + uniform choice for of variable instantiation (similar to **random traversal** of the resolution tree, and then **uniform drawing**...)
- ◆ **Multiple attempts**: When there is a suspicion of failure (the resolution tree becomes too large), another attempt may succeed ... *and succeeds very often*

Existing prototype and intended improvements will be described later in the talk.



Other coverage criteria

- ◆ All vertices
- ◆ All edges
- ◆ MC/DC (various criteria about coverage of conditions in loops or conditionals)
- ◆ How a uniform drawing among **paths** can ensure a good coverage of such **elements** of a graph?



All-vertices and all-edges

- ◆ **First naive approach:**
 - Draw uniformly N elements e_1, \dots, e_N among those to be covered
 - For each e_k , generate uniformly a path among those traversing e_k , of length $\leq n$
- ◆ Let S be the set of elements to be covered (statements, branches, ...); $\forall e \in S$, the probability of e to be activated is

$$p(e) = \frac{1}{|S|} + \frac{1}{|S|} \sum_{e' \in S, e' \neq e} \frac{|C_n^{e, e'}|}{|C_n^{e'}|}$$

where:

$|C_n^e|$ is the nb of paths of length n traversing e and

$|C_n^{e, e'}|$ is the nb of paths of length n traversing both e and e'



Note on the power of combinatorial structures

- ◆ The use of C.S. makes it possible to express, count, and draw uniformly from C_n^e and $C_n^{e,e'}$: in particular there exists algorithms for drawing uniformly in $C_n^{e,e'}$, through recurrence relations and an explicit counting of paths, as in the all-k-path case.
- ◆ Our prototype builds automatically, from the text of the program, CS specifications for C_n and C_n^e . The $C_n^{e,e'}$ are useful for “optimisation” purpose only
- ◆ *Generic, again: applicable to FSM and LTS with S the set of states, or of transitions, or of transition pairs...*



Test Quality

- ♦ What does it mean to “satisfy” a coverage criteria in a randomised framework?

The *test quality* q_N is the *weakest probability* that any element of S has to be covered when N tests are exercised

$$q_N = 1 - (1 - p_{\min})^N$$

where $p_{\min} = \min\{ p(e), / e \in S \}$

Conversely if one wants a “quality of test” q_N this gives a lower bound for the number of tests

$$N \geq \log(1 - q_N) / \log(1 - p_{\min})$$



How to **increase** $\min_{e \in S} p(e)$?

- ◆ By drawing **non uniformly** from S:

let $S = \{e_1, \dots, e_m\}$, and $p(e_i)$ the probability of drawing e_i in S, and

$$c_{i,j} = \frac{|C_n^{e_i, e_j}|}{|C_n^{e_j}|}$$

Then the **probability of reaching e_i with a uniform drawing of path given $p(e_i)$** is

$$\sum_{1 \leq j \leq m} p(e_j) \cdot c_{i,j}$$

- ◆ Given the $c_{i,j}$, the problem of finding the values $\{p(e_1), \dots, p(e_m)\}$ s. t.:

- $\min\{p(e_i), i = 1, \dots, m\}$ is maximum
- $p(e_1) + \dots + p(e_m) = 1$

is a classical one in combinatorial optimisation



PB : Maximise p_{\min} under these constraints

$$\left\{ \begin{array}{l} p_{\min} \leq c_{1,1} \cdot p_1(e_1) + \dots + c_{1,|S|} \cdot p_1(e_{|S|}) \\ \dots \\ p_{\min} \leq c_{|S|,1} \cdot p_1(e_1) + \dots + c_{|S|,|S|} \cdot p_1(e_{|S|}) \\ 1 = p_1(e_1) + p_1(e_2) + \dots + p_1(e_{|S|}) \end{array} \right.$$

This optimisation problem is solved by a Simplex algorithm.

The $p_1(e_j)$ give the distribution for drawing from S 😊

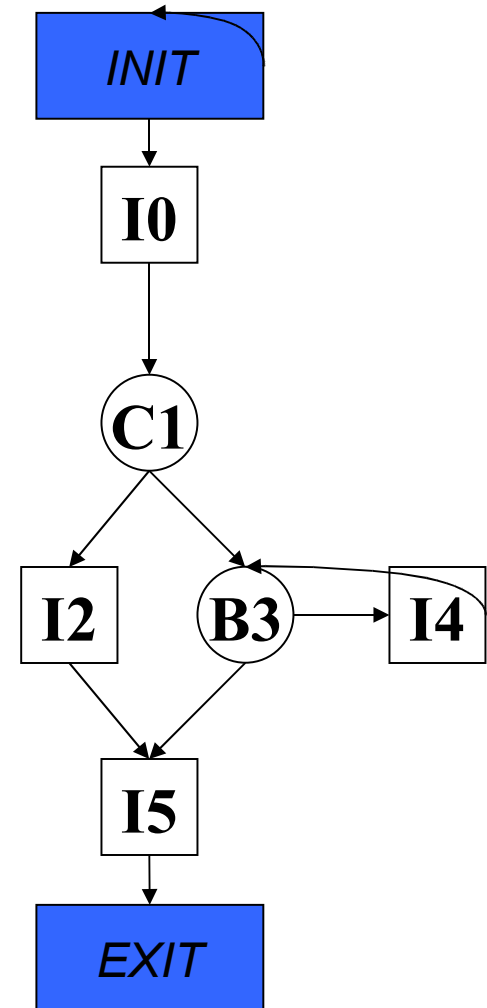
Example: “all squared nodes”

- ◆ $S = \{I0, I2, I4, I5\}$
- ◆ 5 paths of length ≤ 11 (in the original program) or 11 in the C.S., corresponding to 0 (2 ways), 1, 2, 3 traversals of the loop B3-I4
- ◆ *Uniform distribution on S* ☹️
 - $p(I2) = \frac{1}{4} + \frac{1}{4} \quad (1/5 + 0/5 + 1/5) = 7/20 = 0.35,$
 - $p(I4) = \frac{1}{4} + \frac{1}{4} \quad (3/5 + 0/1 + 3/5) = 11/20 = 0.55,$
 - $p(I0) = 1,$
 - $p(I5) = 1$

Hence $p_{\min} = p(I2) = 0.35$

Problem: selecting I0 or I5 gives no way to control what is going on afterwards since they appear in all paths.

Here with $q_N = .99$ one gets $N \geq 11$





Example: all squared nodes

Much better!

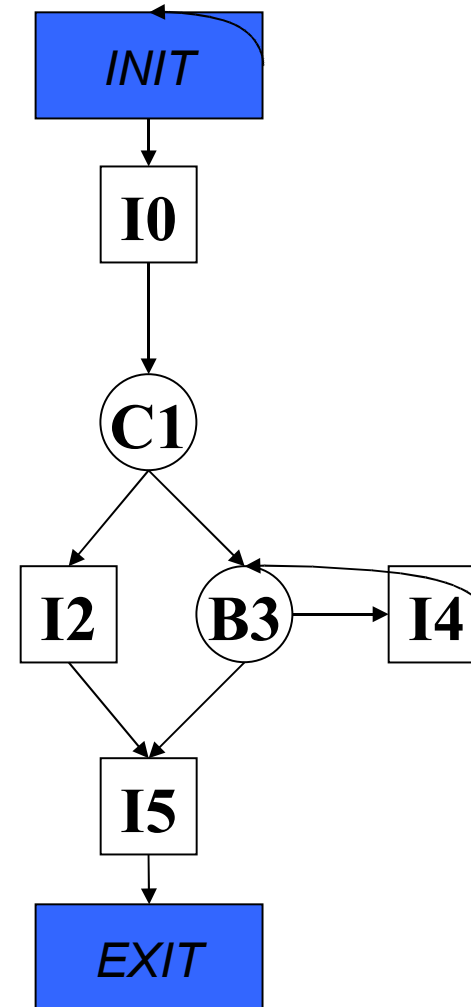
↪ I0 and I5 are “for free”

$$p_1(I2) = p_1(I4) = 0.5$$

↪ $p_1(I0) = p_1(I5) = 0$

↪ $p_{\min} = 0.5$

With $q_N = .99$ one gets $N \geq 6$.

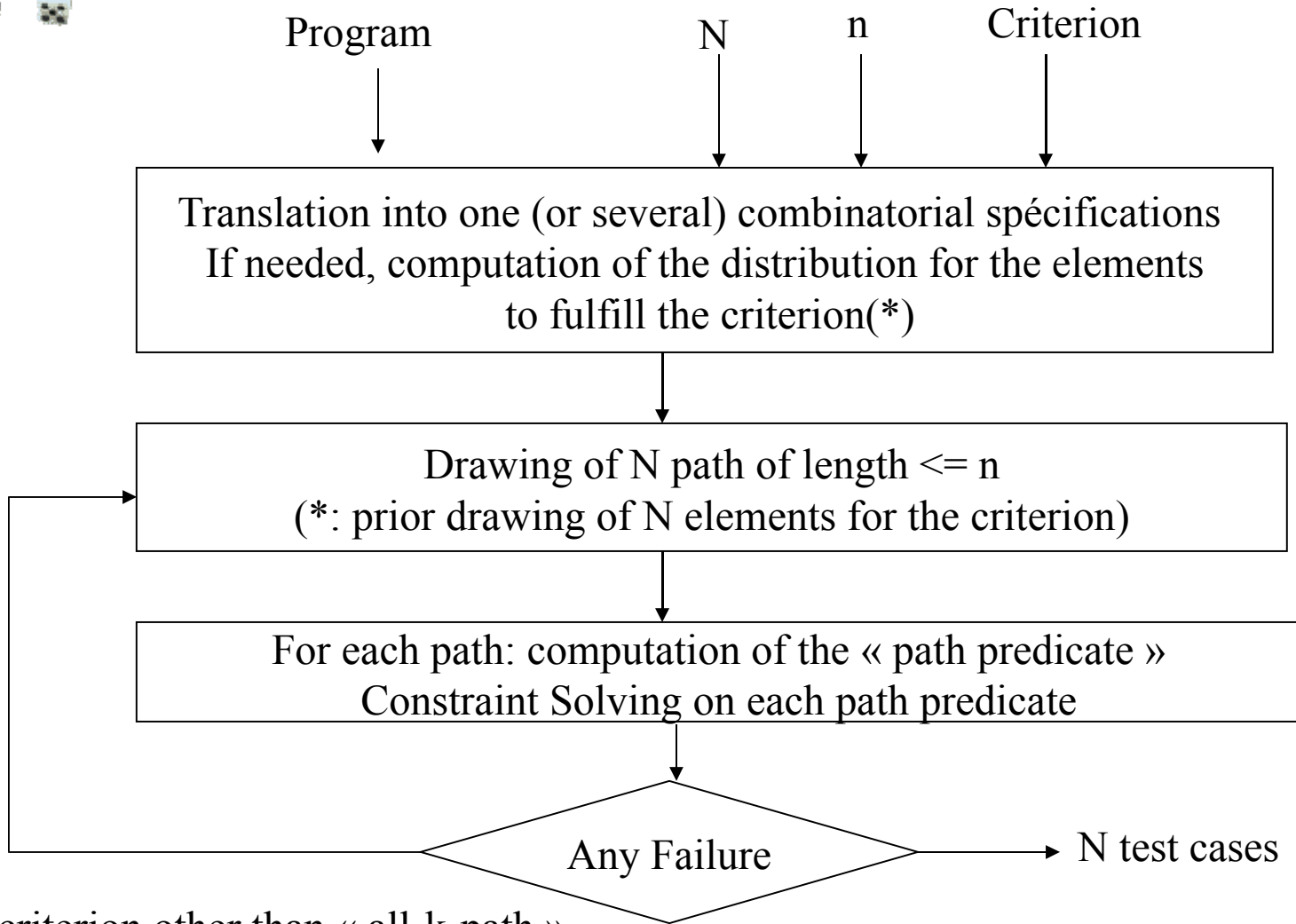




Some references

- ◆ *A calculus for the random generation of labelled combinatorial structures*, Flajolet & al., TCS 132 (1994) 1-35
- ◆ *CS: a MuPAD package for counting and randomly generating combinatorial structures*, Dutour, Denise & Zimmerman, FPSAC'98, 195-204
- ◆ *A new way of automating statistical testing methods*, S. Gouraud & al., 16th IEEE ASE conference, 2001, 5-12
- ◆ *A generic method for statistical testing*, A. Denise & al., 15th IEEE ISSRE, 2004, 25-34

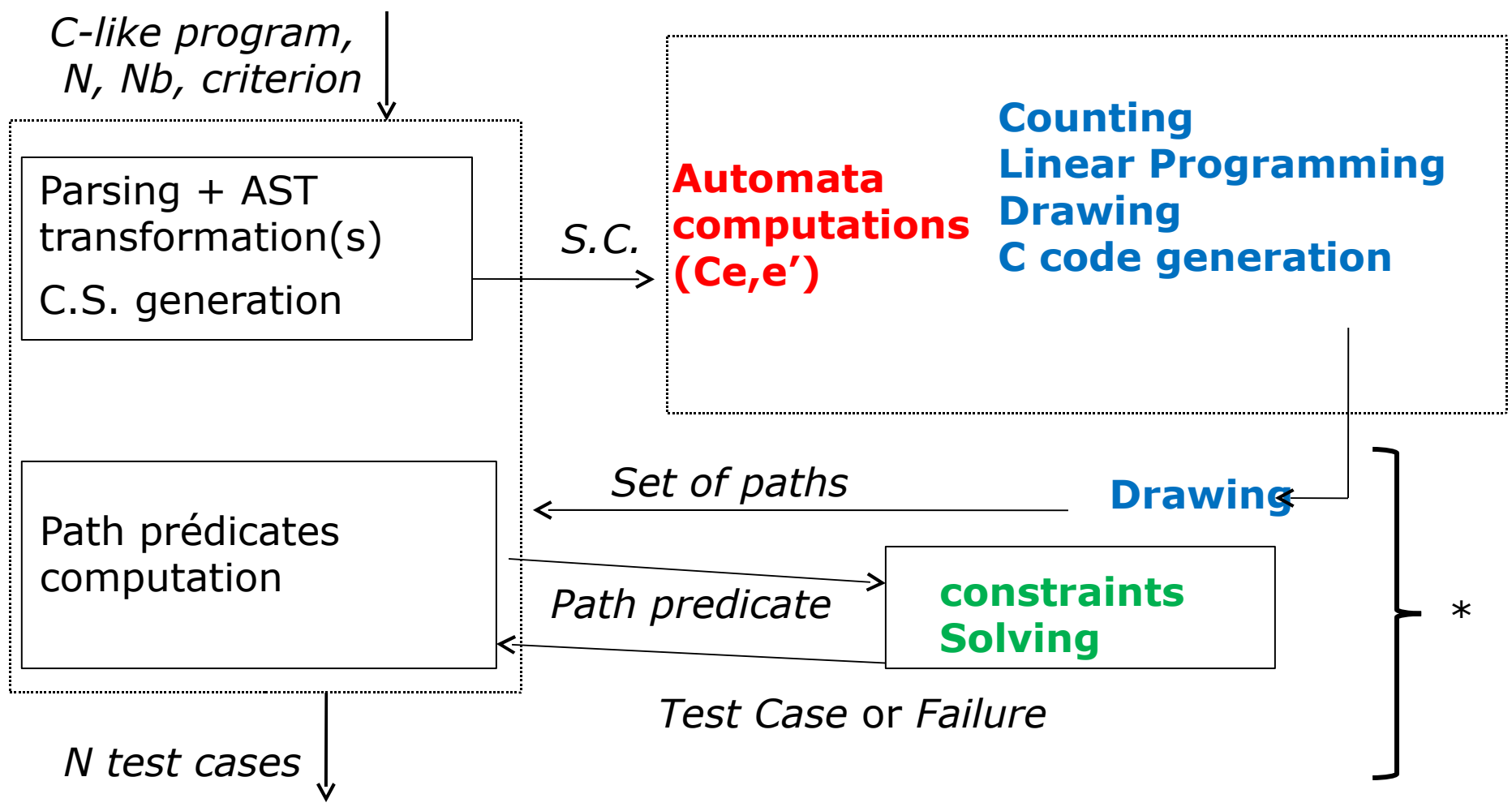
Auguste: General Architecture



(*) for criterion other than « all-k-path »



AUGUSTE – V0: Software Architecture





Comparative Experiments

- ◆ using FCT1, ..., FCT4, programs and mutants used in Waeselynck's thesis at LAAS. There are loops in FCT4 and **a lot** of infeasible paths.
- ◆ LAAS method: explicit construction of an input distribution
 - Solving as many equations as paths in the graph. No automation !
- ◆ FCT1 and FCT2: same results as LAAS
- ◆ FCT3: small variations of the mutation scores due to sensitivity of some mutants to the environment (uninitialized variables in code !)



Experiences with $q_N=0.9999$

	#lines	#paths	Coverage criterion C	Cardinality of C
Fct1	30	17	All paths	17
Fct2	43	9	All paths	9
Fct3	135	33	All paths	33
Fct4	77	∞	All branches	41



Mutation scores

		FCT1	FCT2	FCT3	FCT4	
TFWaCr	Min	1	1	1	0.9898	
	Ave			1	0.9901	
	Max			1	0.9915	
AuGuSTe	Min	1	1	0.9951	0.9854	0.9634
	Ave			0.9989	0.9854	0.9762
	Max			1	0.9854	0.9854

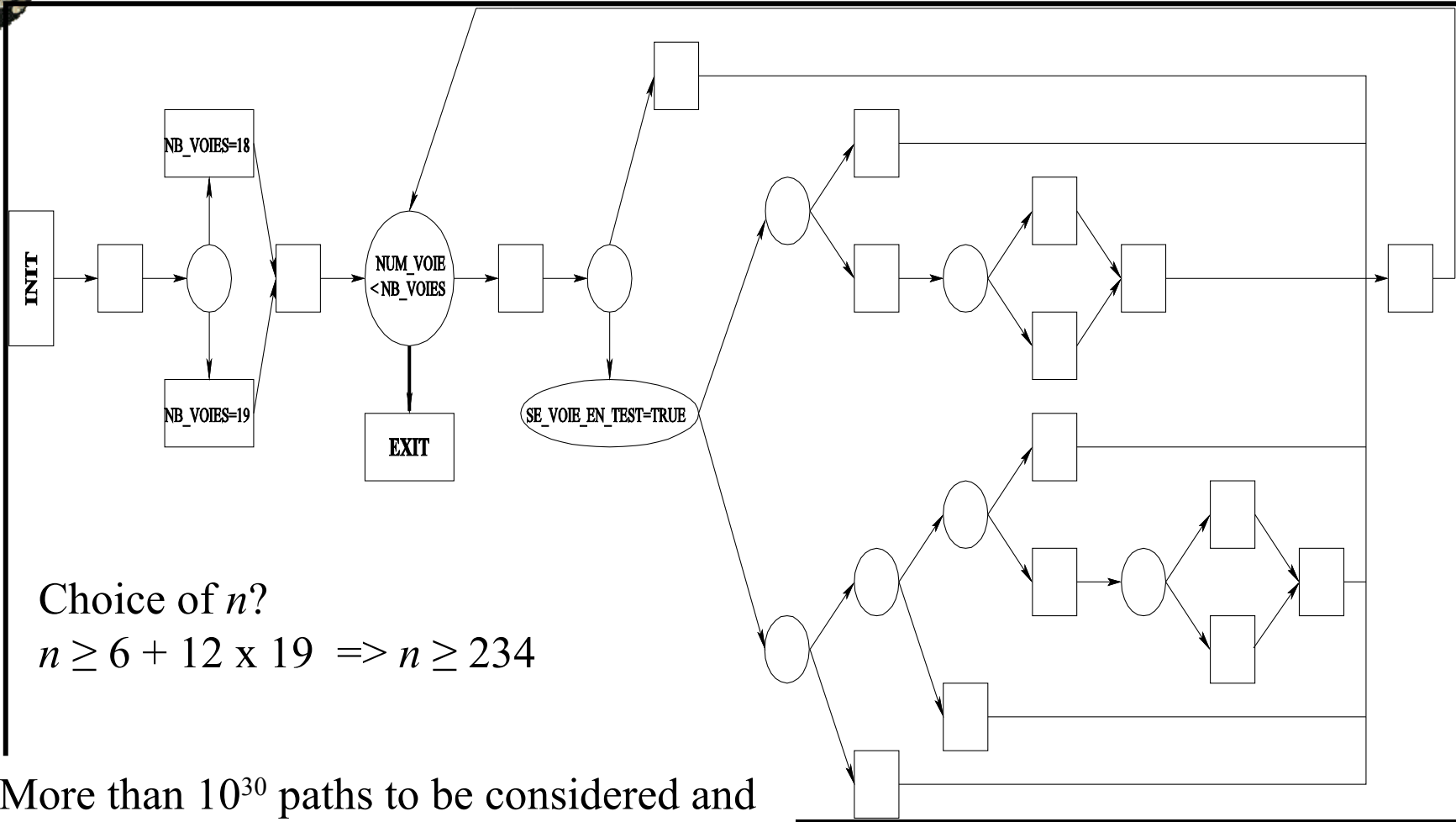
- More than 10000 runs performed on 2914 mutants
- Fct3: non independence of the test experiments



Comparative Experiments (cont.)

- ◆ FCT4 raised a lot of problems
 - Refinement of the combinatorial structure to eliminate the majority of infeasible paths (already included in the prototype) : moving constant test out of loops and static unfolding of loops (the number of iterations was known beforehand) when translating to the C.S.
 - Problem with some “dead code”, which has been removed
 - generation (and resolution) of $5 \cdot 850$ paths of length 250

FCT4 (original version)



Choice of n ?

$$n \geq 6 + 12 \times 19 \Rightarrow n \geq 234$$

More than 10^{30} paths to be considered and a considerable number of unfeasible paths



Potential improvements for Auguste

- ◆ Using more powerful constraint solver(s)
- ◆ Handling a larger subset of C
 - Syntactic extensions
 - *handling « structures » or more primitive types*
 - *Handling more control structures like `continue`; `break`; ...*
 - more « semantical » extensions (handling function calls => needs pre/post conditions for functions !)
- ◆ User annotations for C programs, syntactic transformations of the C.S., Program Slicing, etc.
 - => finer translation into a C.S. that would reflect the dynamic behaviour of a program more precisely
- ◆ Inductive learning of unfeasible paths
- ◆ Going beyond the current method of drawing path (beyond regular languages: adding « real » counting information about loops)



Adding counting information ?

```
int [] f (int[] T1, T2; int IM, JM) {
int[] T3: new int[IM + JM];
int i=1, j=1, k=1;
while (i <= IM && j <= JM) {
    if (T1(i) < T2(j)) {
        T3(k)= T1(i); i++;
    } else { T3(k)=T2(j); j++;}
    k++;
}
while (i <= IM) {
    T3(k)= T1(i); i++; k++;
}
while (j <= JM) {
    T3(k)= T2(j); j++; k++;
}
return t3;
}
```

What does f compute ?

Given that T1 is of size IM and T2 of size JM,
what are the invariants about the numbers
 n_1 , n_2 and n_3 of traversal of the three loop ?

But we are no longer in rational languages.