Testing of Biology Inspired Systems

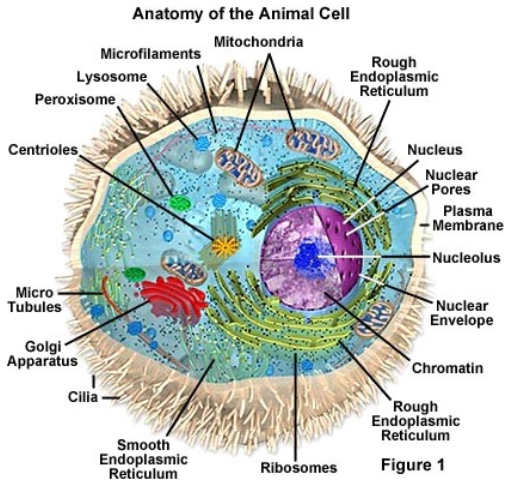
Florentin Ipate
Department of Computer Science
Faculty of Mathematics and Informatics, University of Pitesti
Ro

Summary

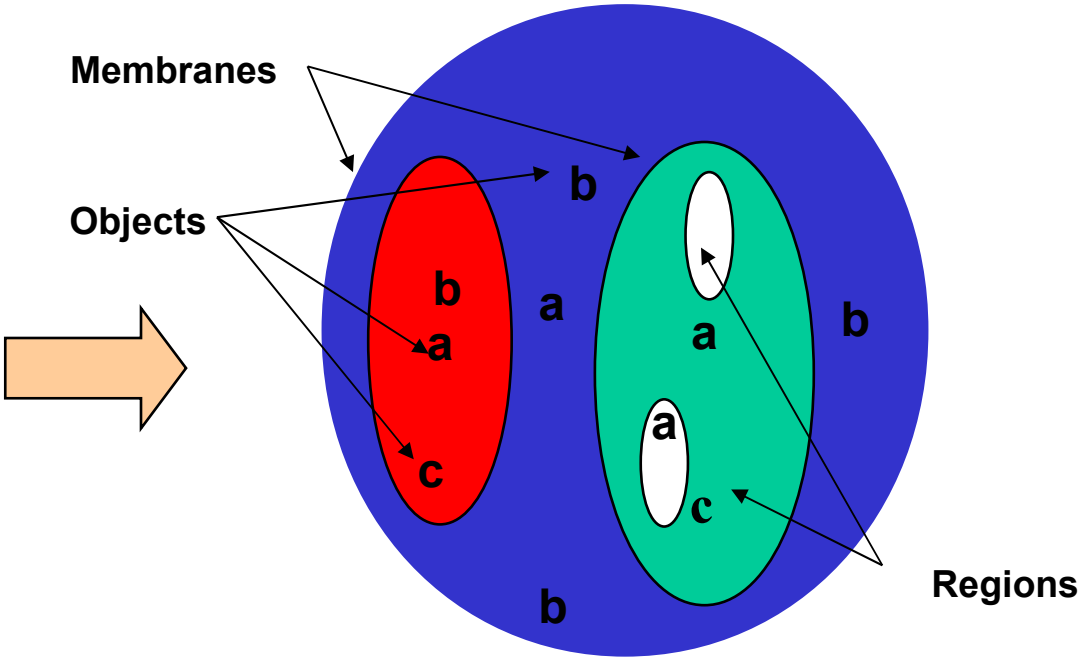
- Membrane computing – P systems
  - origin
  - variants
  - research interests
  - applications
- P System Testing
  - Rule coverage based
  - Finite State Machine based
  - Mutation based
- Further work

Membrane computing – inspired from biology

cell



P system



All cells are enclosed by membranes; the cell membrane acts as the defining principle of what constitutes a cell and the rest of the world. Cells need to be able to transport proteins, DNA, and ions across the membranes.

Science, 310, 2005

P Systems (1)

•A model of computing which abstracts from the functioning and structure of living cells (**P systems**) –Păun, 2000, JCSS

• Three essential features: (1) a hierarchical arrangement of **membranes** delimiting **regions** (***membrane structure***) – *tree structure*, (2) some ***multisets of objects*** and (3) finite ***sets of rules*** associated to regions

•A P system evolves from one configuration to the other by applying the rules according to a given strategy (maximally parallel way)

•Rules can transform objects, move objects, and even modify the membrane structure (creation/division/dissolution/moving)

P Systems (2)

- similar to other biologically inspired formalisms - kappa, brane calculus, chemical abstract machines, cellular automata, L systems, and notations – SBML etc
- the P system research soon became “fast growing area”, Thomson ISI 2003

Păun, Membrane Computing: An Introduction, Springer-Verlag, 2002

Frisco, Computing with Cells. Advances in Membrane Computing, Oxford University Press, 2009

Regular collective volumes are annually edited + Handbook (to appear in autumn 2009); 10th Workshop Membrane Computing 2009

P system web page http://ppage.psystems.eu

http://en.wikipedia.org/wiki/P-systems

P System – Basic definition

$\Pi = (V, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, i_0)$ – P system (with static membrane structure)

V – an alphabet

μ – a membrane structure with n membranes (regions)

$w_1, \ldots, w_n$ – multisets over V; w_i – initial values

$R_1, \ldots, R_n$ – sets of rules

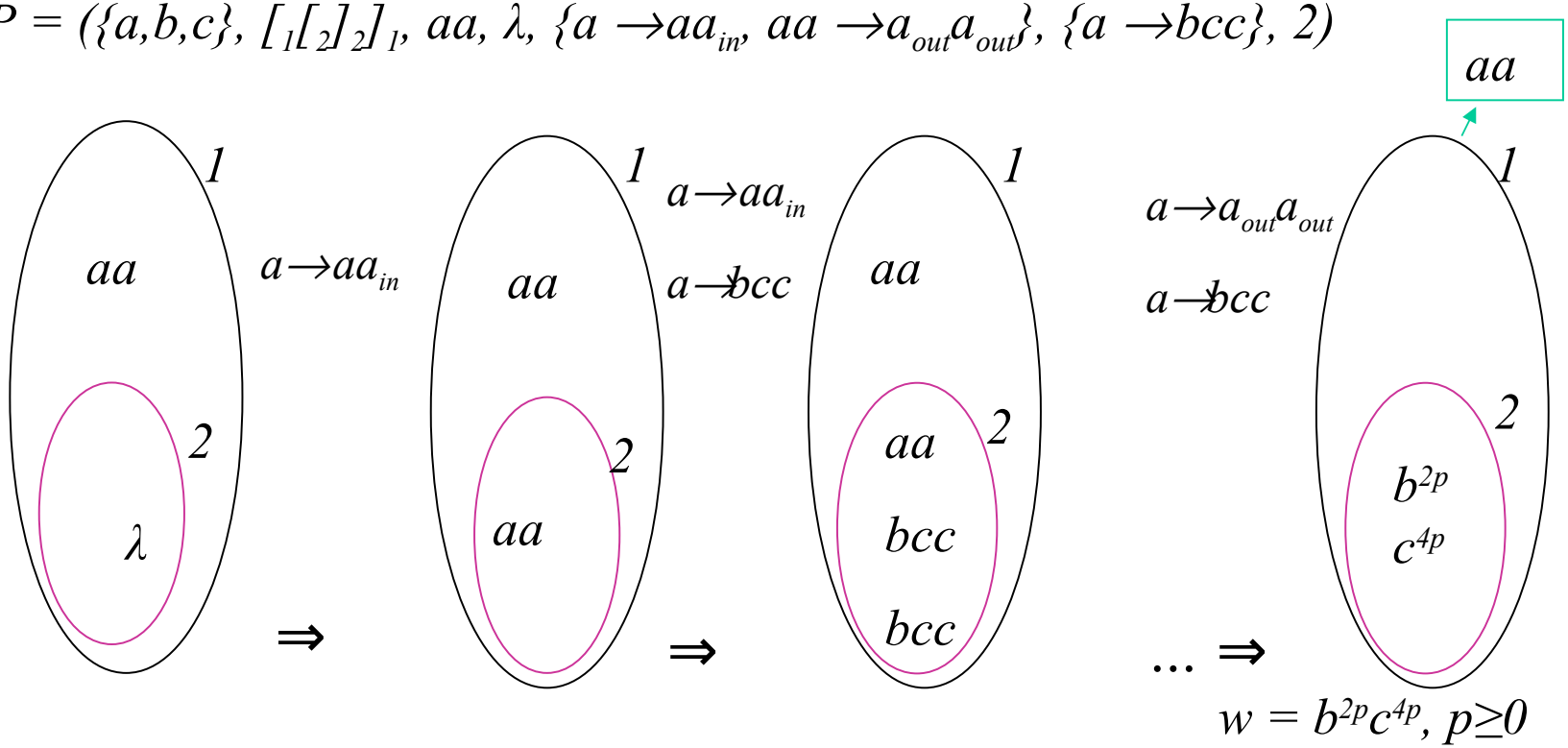
i_0 – the output cell (not significant for testing)

R_i – evolution-communication rules: $a \rightarrow (a_1,t_1)\ldots(a_m,t_m)$;

$A \in V^*$, $a_i \in V$ and $t_i \in \{in, out, here\}$

P system : simple example

$P = (\{a,b,c\}, [_1[_2]_2]_1, aa, \lambda, \{a \rightarrow aa_{in}, aa \rightarrow a_{out}a_{out}\}, \{a \rightarrow bcc\}, 2)$



$w = b^{2p}c^{4p}, p \geq 0$

P system : Computation

$P = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, i_0)$ - P system

A computation is a sequence of configurations $C_0, \ldots, C_p$ (C_i – n-tuple of multisets of symbols); the initial configuration is $C_0 = (w_1, \ldots, w_n)$

C_{i+1} is obtained from C_i by applying in (maximal parallel fashion) the rules available to all multisets of the n components

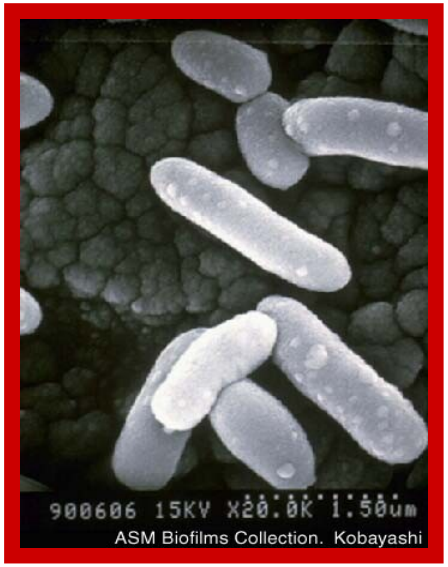
$C_0=(aa,\lambda)$, $C_1=(aa,aa)$, $C_2=(aa,aabccbcc)$, $\ldots C_{p+1}=(\lambda,b^{2p}c^{4p})$

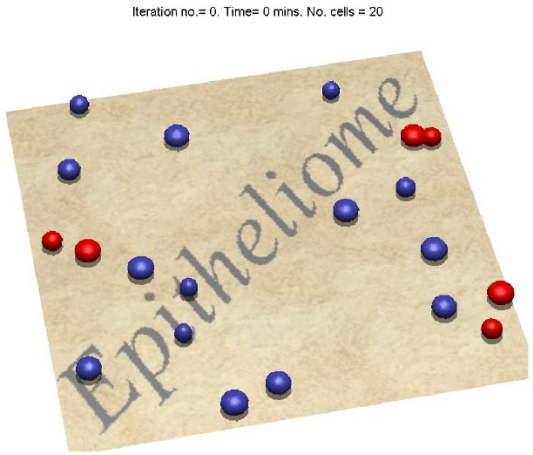
Notation: $C_0 \Rightarrow C_1 \Rightarrow \ldots \Rightarrow C_p \Rightarrow C_{p+1}$; $C_0 \Rightarrow^* C_{p+1}$

The result is read from i_0 (*2* – example); basic interpretation: number of symbols ($\{6p|p\geq 0\}$).

From cell to tissue, population, colony…

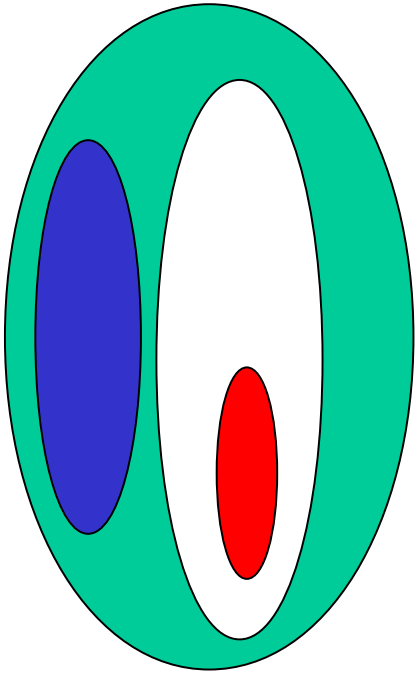
- A community of individuals (units: social insects, cells, bacteria etc.) performing different tasks
- The units evolve (transform, move etc)
- Interact with(in) an environment
- Dynamic system (individuals coming in and going out) with dynamic structure
- May self-assemble into specific patterns
- May self-organize to achieve some goals



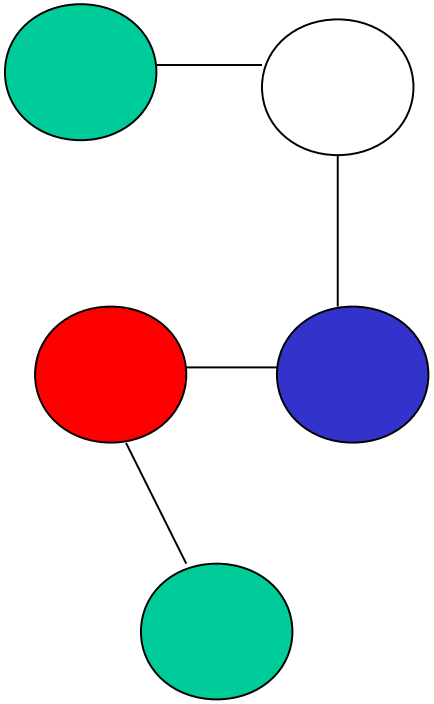


Classes of P systems

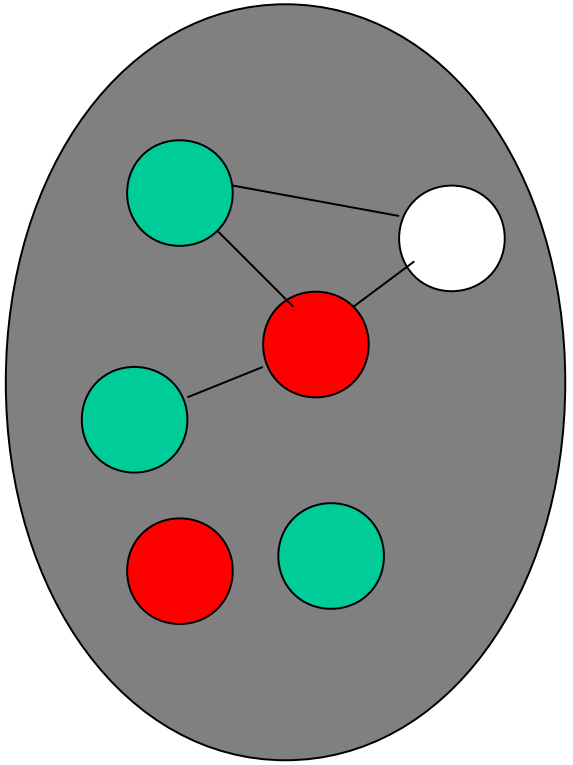
(cell) P systems



tissue P systems



population P systems



Research interests in P systems

- Many variants have been studied (a full account @: http://ppage.psystems.eu)

- Research focusing on

  - Theoretical aspects
    - establishing computational power of different variants of P systems
    - hierarchies of classes of languages produced by these devices
    - solving NP complete problems such as SAT, decidability, complexity aspects

  - Modelling
    - using P system to model natural processes in living cells
    - applications in graphics, networking, linguistics
    - relationships with other formalisms (Petri nets, brane calculi, X-machines)

  - Tools

P systems in modelling and simulation

• In the last years there have been significant developments in using the P systems paradigm to model, simulate and formally verify various systems (biology, economics, linguistics, graphics, computer science etc) – Ciobanu, Păun, Perez-Jimenez, 2006 and some special issues of BioSystems

• Software packages developed for some of these applications

(P system web page http://ppage.psystems.eu)

• Although formal verification has been applied, testing, until recently, has been completely neglected in this context

Software Testing

Software testing is

• the process of checking software, to verify that it *satisfies its requirements* and to detect errors.
• includes, but is not limited to, *the process of executing a program* or application with the intent of finding software bugs. (http://en.wikipedia.org/wiki/Software_testing)

• Test case (test suite) generation: selection of test values most likely to find faults – major part of software testing

Test Generation Strategies

- White box (structural): test values are derived from the implementation (code).

- Black box (functional): test values are derived from requirements (specification)

- Underlying idea (in both cases): *coverage* - select test values that *cover* the specification / implementation as much as possible

The Triangle Program

The aim of this program is to classify triangles. The program accepts three positive integers as lengths of the sides of a triangle. The program classifies the triangle into one of the following groups:

- *Equilateral:* all the sides have equal lengths (return 1)
- *Isosceles:* two sides have equal length, but not all three (return 2)
- *Scalene:* all the lengths are unequal (return 3)
- *Impossible:* the three lengths cannot be used to form a triangle, or form only a flat line (return 4)

Adapted from

http://www.cs.bris.ac.uk/Teaching/Resources/COMS12100/reports/triangle.html

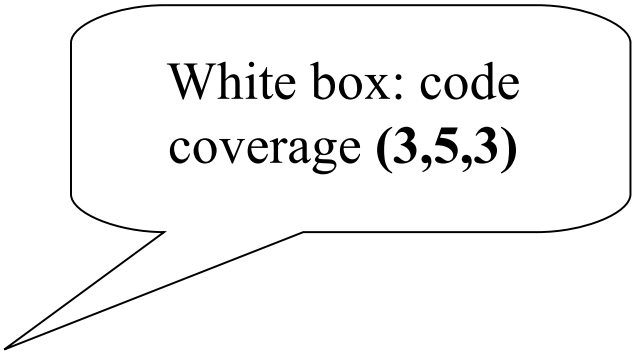
(appears in Myers' book)

Java Implemantation

```
int triangle(int a, int b, int c)
{
   int mx, x, y;
   mx = a; x = b; y = c;
   if (mx < b)
      {x = mx; mx = b;}
   if (mx < c)
      {y = mx; mx = c;}
   if (mx >= x + y)
      {return 4; // impossible}
   if (a == b && b == c)
      {return 1; // equilateral}
   if (a == b || b == c || a == c)
      {return 2; // isosceles}
   return 3; // scalene
}
```

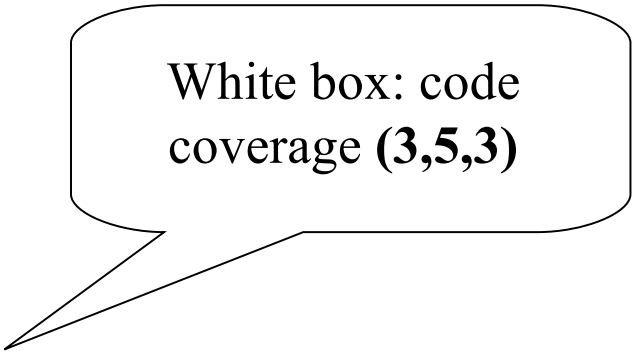
Java Implementation

```
int triangle(int a, int b, int c)
{
   int mx, x, y;
   mx = a; x = b; y = c;
   if (mx < b)
      {x = mx; mx = b;}
   if (mx < c)
      {y = mx; mx = c;}
   if (mx >= x + y)
      {return 4; // impossible}
   if (a == b && b == c)
      {return 1; // equilateral}
   if (a == b || b == c || a == c)
      {return 2; // isosceles}
   return 3; // scalene
}
```



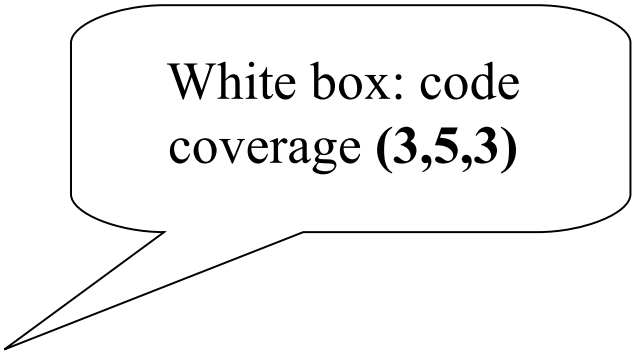
Java Implementation

```
int triangle(int a, int b, int c)
{
   int mx, x, y;
   mx = a; x = b; y = c;
   if (mx < b)
      {x = mx; mx = b;}
   if (mx < c)
      {y = mx; mx = c;}
   if (mx >= x + y)
      {return 4; // impossible}
   if (a == b && b == c)
      {return 1; // equilateral}
   if (a == b || b == c || a == c)
      {return 2; // isosceles}
   return 3; // scalene
}
```



Java Implementation

```
int triangle(int a, int b, int c)
{
   int mx, x, y;
   mx = a; x = b; y = c;
   if (mx < b)
      {x = mx; mx = b;}
   if (mx < c)
      {y = mx; mx = c;}
   if (mx >= x + y)
      {return 4; // impossible}
   if (a == b && b == c)
      {return 1; // equilateral}
   if (a == b || b == c || a == c)
      {return 2; // isosceles}
   return 3; // scalene
}
```



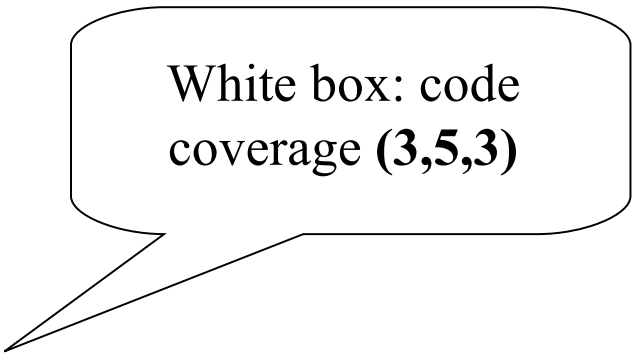
Java Implementation

```
int triangle(int a, int b, int c)
{
   int mx, x, y;
   mx = a; x = b; y = c;
   if (mx < b)
      {x = mx; mx = b;}
   if (mx < c)
      {y = mx; mx = c;}
   if (mx >= x + y)
      {return 4; // impossible}
   if (a == b && b == c)
      {return 1; // equilateral}
   if (a == b || b == c || a == c)
      {return 2; // isosceles}
   return 3; // scalene
}
```

White box: code coverage **(3,5,3)**

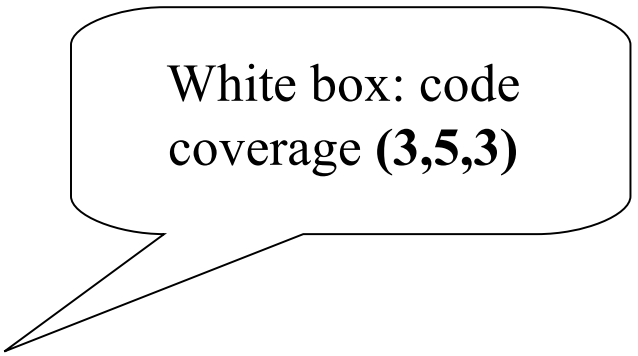
Java Implementation

```
int triangle(int a, int b, int c)
{
   int mx, x, y;
   mx = a; x = b; y = c;
   if (mx < b)
      {x = mx; mx = b;}
   if (mx < c)
      {y = mx; mx = c;}
   if (mx >= x + y)
      {return 4; // impossible}
   if (a == b && b == c)
      {return 1; // equilateral}
   if (a == b || b == c || a == c)
      {return 2; // isosceles}
   return 3; // scalene
}
```



Java Implementation

```
int triangle(int a, int b, int c)
{
   int mx, x, y;
   mx = a; x = b; y = c;
   if (mx < b)
      {x = mx; mx = b;}
   if (mx < c)
      {y = mx; mx = c;}
   if (mx >= x + y)
      {return 4; // impossible}
   if (a == b && b == c)
      {return 1; // equilateral}
   if (a == b || b == c || a == c)
      {return 2; // isosceles}
   return 3; // scalene
}
```



Coverage Methods (Structural Testing)

- Program represented as a directed graph and various coverage criteria can be defined:
  - Statement (node) coverage
  - Branch (decision) coverage
  - Multiple condition coverage
  - Sequences of branches of a given length, etc
- Coverage criteria can also be used in functional testing, e.g. rule coverage for specifications represented as Context-Free Grammars – each production rule of the grammar is applied at least once.

Mutation testing (mutation analysis)

- Method of *structural* software testing, which involves modifying program's source code in small ways.

- The modified source code is called a *mutant.*

- Various mutation “operators” (ways of introducing errors into the correct code) have been defined and implemented.
  - traditional mutants
  - mutants for specialized programming environments, e.g. object-oriented languages (MuJava).

Mutant construction

Original program

```
...

if (a && b) then
   c = 0
else
   c = 1;
```

Mutant 1

```
...

if (a || b) then
   c = 0
else
   c = 1;
```

Mutant 2

```
...

if (a && b) then
   c = 1
else
   c = 1;
```

Mutant 3

```
...

if (a || b) then
   c = 0
else
   c = 0;
```

Mutation testing (mutation analysis)

• If the test suite is able to detect the change (i.e. one of the tests fails), then the mutant is said to be *killed*.

• Two types of mutation
  - **weak mutation** - the test input causes different program states for the mutant and the original program
  - **strong mutation** - same condition but also the erroneous state to be propagated at the end of the program

•Related to coverage methods : in order to kill a mutant the faulty code must be exercised.

Test generation based on a formal model

- Functional testing based on a *formal* specification (model)
  - test values can be derived in a rigorous manner
  - test derivation can be automated

- **Conformance testing**: Assumption: the implementation under test (IUT) can be modelled by an unknown model, belonging to a known set – *the fault model*

- The test suite determines if the IUT *conforms* to the specification

- Example: FSM based techniques: UIO, W, Wp, etc.

- W and Wp method: Assumption: the maximum number of states of the implementation is known. (The size of the test set depends on the chosen upper bound).

Rule coverage based P system testing

One compartment P system, Π

• A test set T for Π consists of multisets such as for any rule r in Π there is $u \in T$ such that u covers r

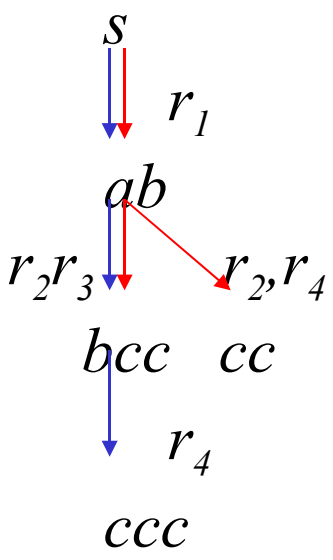
u **covers** r: $a \rightarrow v$ iff there is $w \Rightarrow^* xay \Rightarrow x'vy' \Rightarrow^* u$

(*simple rule coverage*)

• Test application – checks whether all elements of the test set are computed by the implementation

Example

Π consists of r_1: $s \to ab$; r_2: $a \to c$; r_3: $b \to bc$; r_4: $b \to c$



$T=\{ab, bcc, ccc\}$

or

$T=\{ab, bcc, cc\}$

rule coverage

Implementations:

Π_1: r_1: $s \to ab$; r_2: $a \to \lambda$; r_3: $b \to c$ //can't compute *bcc, cc, ccc*

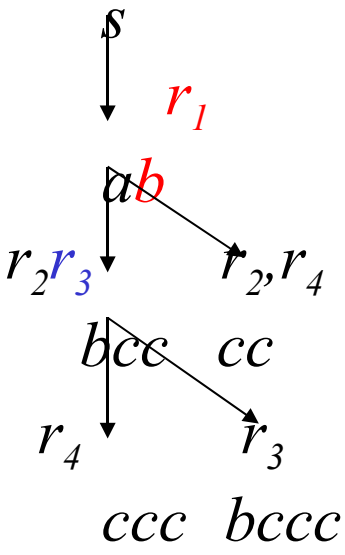
Π_2: r_1: $s \to ab$; r_2: $a \to bc$; r_3: $a \to c$; r_4: $b \to c$ // computes both T, T

Obs. *bccc* is not computed by Π_2 but is produced by the model Π

Context-dependent Rule Coverage

- Each rule should have a cover in every of its direct context

Example: for Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$,
The rules r_1 $s \rightarrow ab$ & r_3 $b \rightarrow bc$ represent the direct contexts of the rules r_3 $b \rightarrow bc$ and r_4 $b \rightarrow c$; r_1 $s \rightarrow ab$ direct context of r_2 $a \rightarrow c$

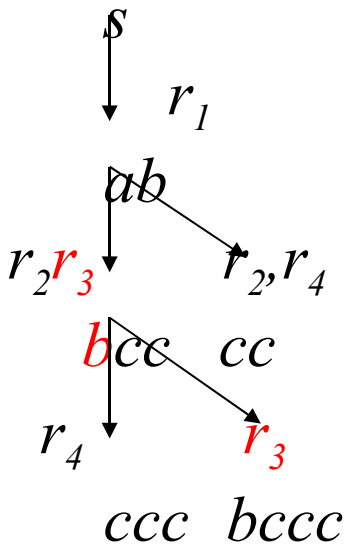


context-dependent rule coverage

Context-dependent Rule Coverage

- Each rule should have a cover in every of its direct context

Example: for Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$,
The rules r_1 $s \rightarrow ab$ & r_3 $b \rightarrow bc$ represent the direct contexts of the rules r_3 $b \rightarrow bc$ and r_4 $b \rightarrow c$; r_1 $s \rightarrow ab$ direct context of r_2 $a \rightarrow c$

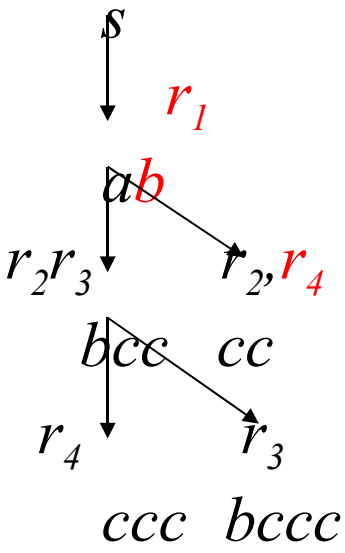


context-dependent rule coverage

Context-dependent Rule Coverage

- Each rule should have a cover in every of its direct context

Example: for Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$,
The rules r_1 $s \rightarrow ab$ & r_3 $b \rightarrow bc$ represent the direct contexts of the rules r_3 $b \rightarrow bc$ and r_4 $b \rightarrow c$; r_1 $s \rightarrow ab$ direct context of r_2 $a \rightarrow c$

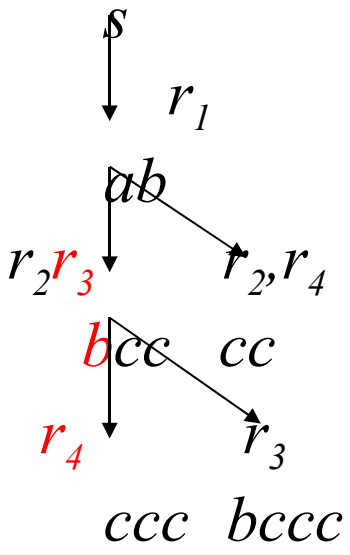


context-dependent rule coverage

Context-dependent Rule Coverage

- Each rule should have a cover in every of its direct context

Example: for Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$,
The rules r_1 $s \rightarrow ab$ & r_3 $b \rightarrow bc$ represent the direct contexts of the rules r_3 $b \rightarrow bc$ and r_4 $b \rightarrow c$; r_1 $s \rightarrow ab$ direct context of r_2 $a \rightarrow c$



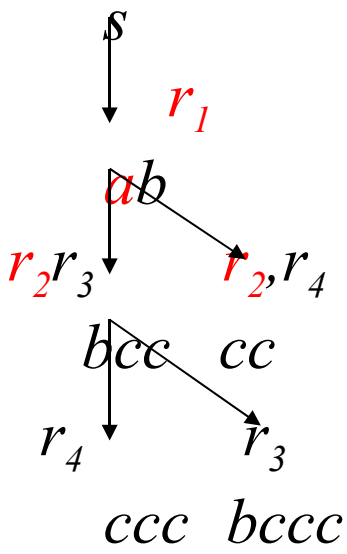
context-dependent rule coverage

Context-dependent Rule Coverage

- Each rule should have a cover in every of its direct context

Example: for Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$,

The rules r_1 $s \rightarrow ab$ & r_3 $b \rightarrow bc$ represent the direct contexts of the rules r_3 $b \rightarrow bc$ and r_4 $b \rightarrow c$; r_1 $s \rightarrow ab$ direct context of r_2 $a \rightarrow c$



context-dependent rule coverage

Π : r_1 : $s \rightarrow ab$; r_2 : $a \rightarrow bc$; r_3 : $a \rightarrow c$; r_4 : $b \rightarrow c$ // don't compute *bccc*

Multiple Compartment P Systems

• Rule coverage:
$(u_1, \ldots, u_n)$ covers $r_i: a_i \rightarrow v_i$ iff
$(w_1, \ldots w_n) \Rightarrow^* (x_1, \ldots x_i a_i y_i, \ldots x_n) \Rightarrow (x_1', \ldots x_i' v_i y_i', \ldots x_n') \Rightarrow^*$
$(u_1, \ldots u_n)$

• Simple rule coverage is defined similarly to one compartment

• Context-dependent rule coverage – consider evolution rules from the same cell and communication rules from the neighbouring cells:
$r': b \rightarrow uav$ in R_i is direct context for $r: a \rightarrow x$ in R_i
$r'': c \rightarrow u'(a,t)v'$ in R_j (t is either *in* or *out* and i, j are neighbouring cells) is also direct context for $r: a \rightarrow x$ in R_i

M Gheorghe, F Ipate (2008) On testing P systems. LNCS, 5397, 2008, pp 173—188.

Testing based on Finite State Machine

• Build a derivation tree of the P System for a finite sequence of steps, k

• Tree = DFA which accepts finite language U over alphabet A, composed of multisets of rules (labels of the tree arcs)

• Construct a deterministic finite cover (DFC) for U – a deterministic finite state machine that accepts all sequences in U and possibly sequences that are longer than any word of U

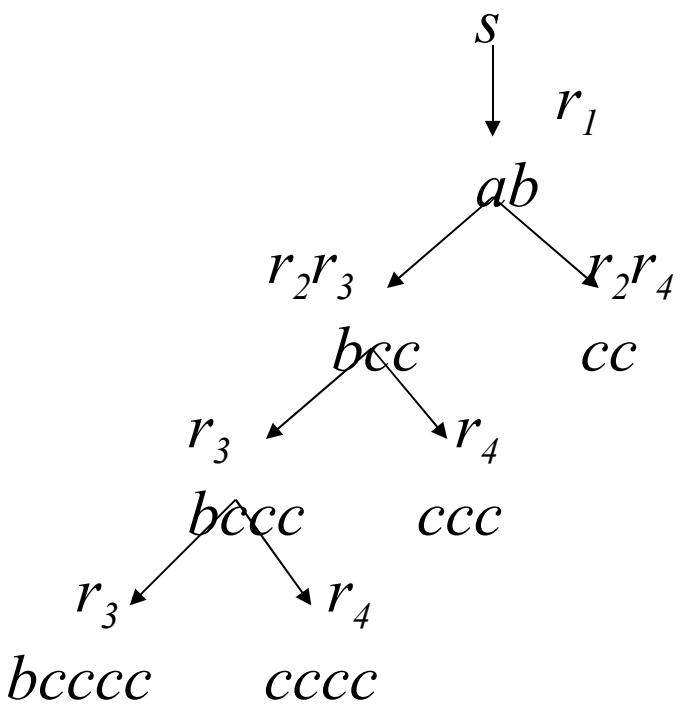
• Generate a test set, T, over the P systems alphabet V, for a certain coverage principle (e.g. state or transition coverage)

• Conformance testing for DFC (e.g. W method)

F Ipate, M Gheorghe: Finite state based testing of P systems, Natural Computing, to appear.

Derivation Tree for a Given k

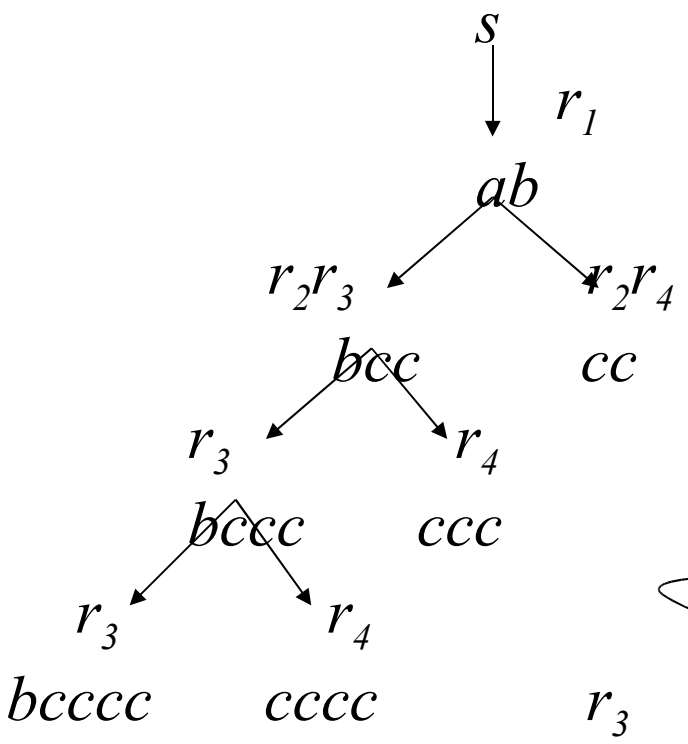
Example: for Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$ (1 comp)

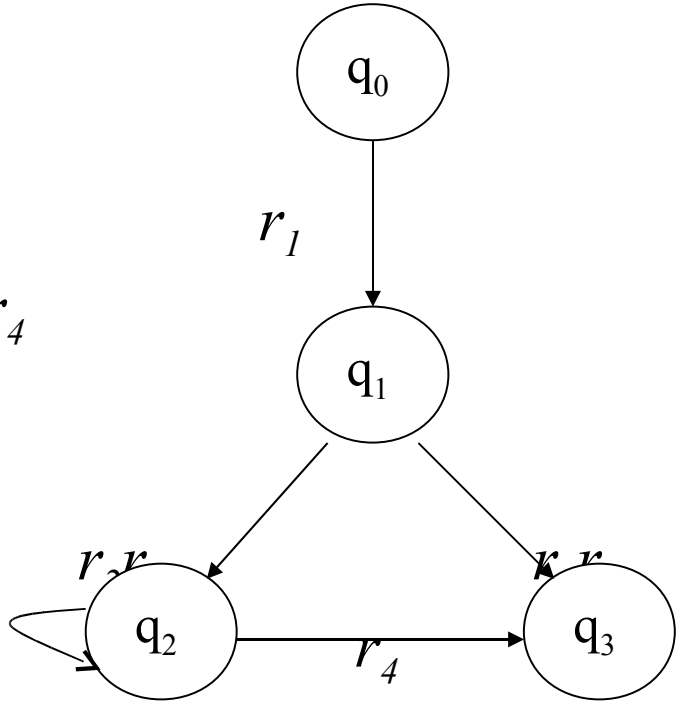


$k = 4$ steps, obtain Dt – a DFA over the set of labels defining the multisets of rules applied $\{r_1, r_2r_3, r_2r_4, r_3, r_4\}$ accepting L_{Dt}

DFC for L_{Dt}

Example: for Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$





DFC, M, for L_{Dt}

(Original) W method

- Specification is a FSM with outputs (Mealy machine).

- Fault model: all FSMs having at most k more states than the specification.

- **State cover** S: reaches every state q of the specification.

- **Characterization set W**: distinguishes between every pair of states of the specification.

- Test suite: S A[k+1] W, where A[i] = $\{\lambda\ \} \cup \ldots \cup A^i$

W method for DFC Automata

- Specification is a finite automaton (acceptor) with all states final.

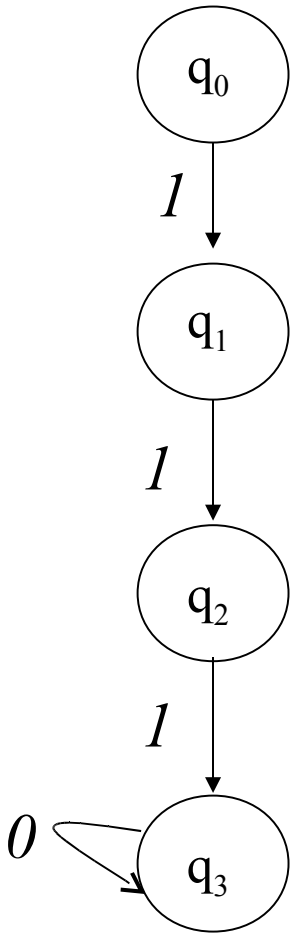
- Test suite checks if the implementation behaves identically with the specification for all sequences of length at most an upper bound N.

- Construction of S and W: *sequences of minimum possible length* are chosen to reach states or distinguish between states:

  - S - ***proper*** *state cover*
  - W - s**trong** *characterization set* ($\lambda \in W$)

- Test suite: $(S\ A[k+1]\ W) \cap A[N]$

W method for DFC: Example



S = {λ , 1, 11, 111}

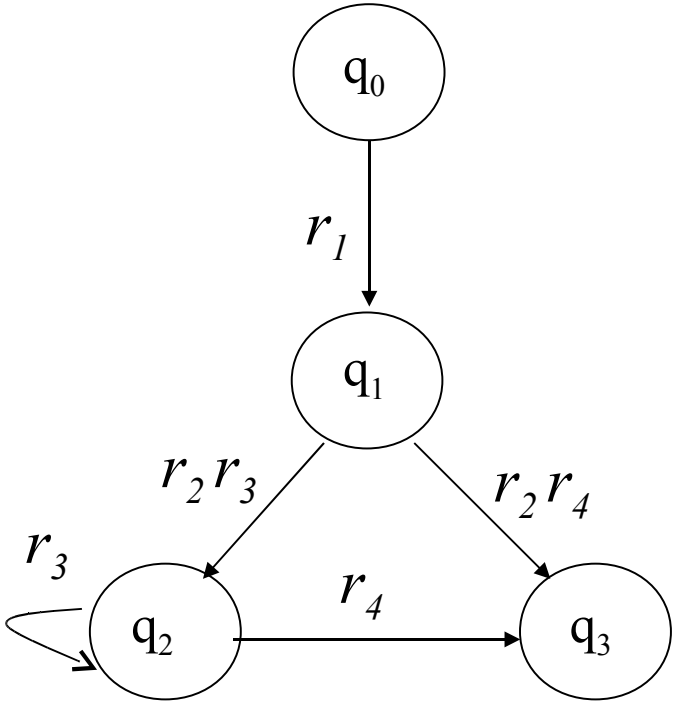
W = {λ , 1, 11, 111}

S = {λ , 1, 11, 1110}

W = {λ , 111}

Incorrect

Example



$S = \{\lambda \ , r_1 , r_1 . r_2 r_3 , r_1 . r_2 r_4\}$

$W = \{\lambda \ , r_1 , r_2 r_3 , r_3\}$

Model based mutation testing

- Mutation analysis has been largely used in white-box testing

- Only a few attempts to use mutation analysis in black-box testing have been reported in the literature:

- J. Offutt, P. Ammann, G. Mason, L. (Ling) Liu (2006).

Mutation Testing implements Grammar-Based Testing,

Proceedings of the Second Workshop on Mutation Analysis - ideas are rather general and lack formalization.

Model based mutation testing

- Model-based mutation testing: mutate the model (specification)

- A particular type of conformance testing: the fault model is the set of mutants

- If the model is described using a *context-free grammar*: construct mutants for the *parse tree*.

Context free grammar

- A system G = (V, T, P, S), where
  - V is the set of variables (nonterminals)
  - T is the set of terminals
  - P is the set of production rules of the form $A \rightarrow w$, A is a single nonterminal symbol, and w is a string of terminals and/or nonterminals;
  - S is the start symbol.

- The language L(G) described by G is composed of all strings that can be derived from the start symbol S.

- G is said to be proper if it has no useless symbols (inaccessible symbols or unproductive symbols), and no λ -productions.

- We only consider proper grammars.

Context free grammar

- A derivation (parse) tree for a (proper) context-free grammar G

is a tree that satisfies the following conditions:

  - each non-leaf node is labelled by a nonterminal in V
  - each leaf node is labelled by a terminal in T
  - if a non-leaf node is labelled A and its children are labelled X_1, …, X_k then $A \rightarrow X_1 \ldots X_k$ is a production rule of G.

- The yield of the tree is the string of terminals obtained by concatening the leaves from left to right.

- $w \in L(G)$ iff w is the yield of some parse tree with root S

- Parse trees have very high practical value as are used by compilers to represent the structure of the source code.

Context free grammar

• A grammar is said to be *ambiguous* if there exists a string that can be generated by more than one parse tree.

• Usually (but not always), ambiguity is a feature of the grammar, not of the language and unambiguous grammars can be found to describe the same context-free language.

• An ambiguous grammar presents a practical problem since a string may be associated with more than one parse tree.

• However, techniques for eliminating the causes of ambiguity are used in compiler construction.

• (Possibly) ambiguous grammars are considered, but the language has to be unambiguous.

CFG based mutation testing

- For each production rule define a set of rule mutants

- If r is $A \rightarrow X_1 \ldots X_k$ then a mutant has the form $A \rightarrow X_1' \ldots X_n'$, such that each X_i' is either a terminal or can be found among $X_1 \ldots X_k$.

- The mutated production rule also belong to the original grammar or have the form $A \rightarrow A$, where A is a nonterminal - this ensures that the resulting mutant is *syntactically valid*.

- Syntactically incorrect mutants have no practical value for test generation.

CFG based mutation testing

- Particular rules mutants of interest:

  – *Terminal replacement mutant*: a terminal in $X_1 \ldots X_k$ is replaced by another terminal

  – *Terminal insertion mutant*: one terminal is inserted into the string $X_1 \ldots X_k$ (at any position).

  – *String deletion mutant*: one or more symbols are removed from the string $X_1 \ldots X_k$

  – *String reordering mutant*: the string $X_1 \ldots X_k$ is reordered.

Mutant construction

- A one-node tree has no mutants

- A mutant of MakeTree(A, $SubTree_1$, …, $SubTree_k$, r) is of the form:

  – **Subtree mutation**: MakeTree(A, $SubTree_1$', …, $SubTree_k$', r), if there exists $1 \leq j \leq k$ such that $SubTree_j$' is a mutant of $SubTree_j$ and $SubTree_i$' = $SubTree_i$, $1 \leq i \leq k$, $i \neq j$

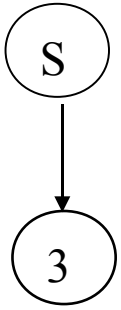
  – **Rule mutation**: MakeTree(A, $SubTree_1$', …, $SubTree_n$', r') if there exists a mutant r':$A \rightarrow B_1$' … B_n' of r such that for every $1 \leq i \leq n$ there exists $1 \leq j \leq k$ with $SubTree_i$' = $SubTree_j$

Mutant construction: example

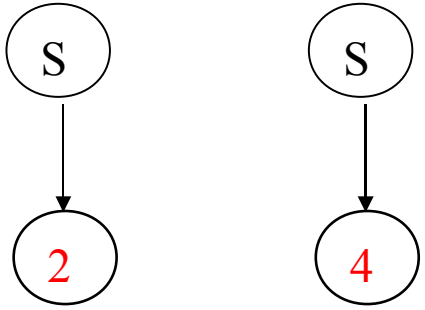
- G = (V, T, R, S), where
  - V = {S}
  - T = {0, … N} ∪ {+, -}
  - S → S + S | S – S | i, 0 ≤ i ≤ N.
- Rule mutants G = (V, T, R, S), where
  - S → S + S : mutants S → S - S | S
  - S → S - S : mutants S → S + S | S
  - S → i, 1 ≤ i ≤ k - 1 mutants S → i +1 | i – 1
  - S → 0 mutants S → 1
  - S → N mutants S → N - 1

Mutant construction: example

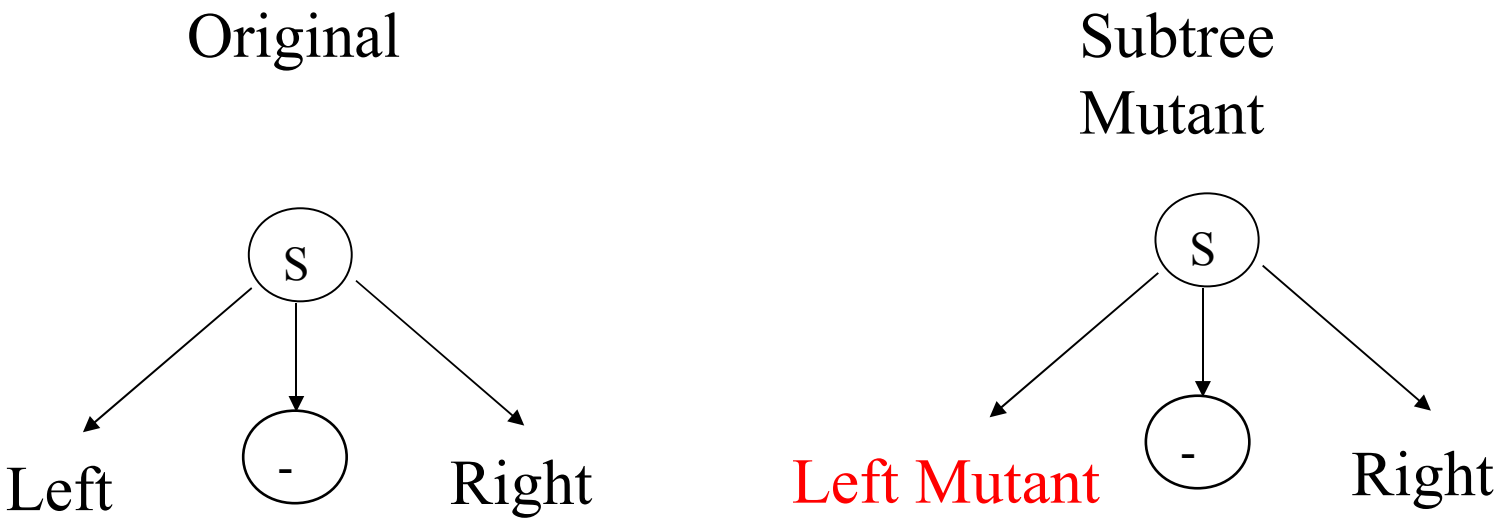
Original



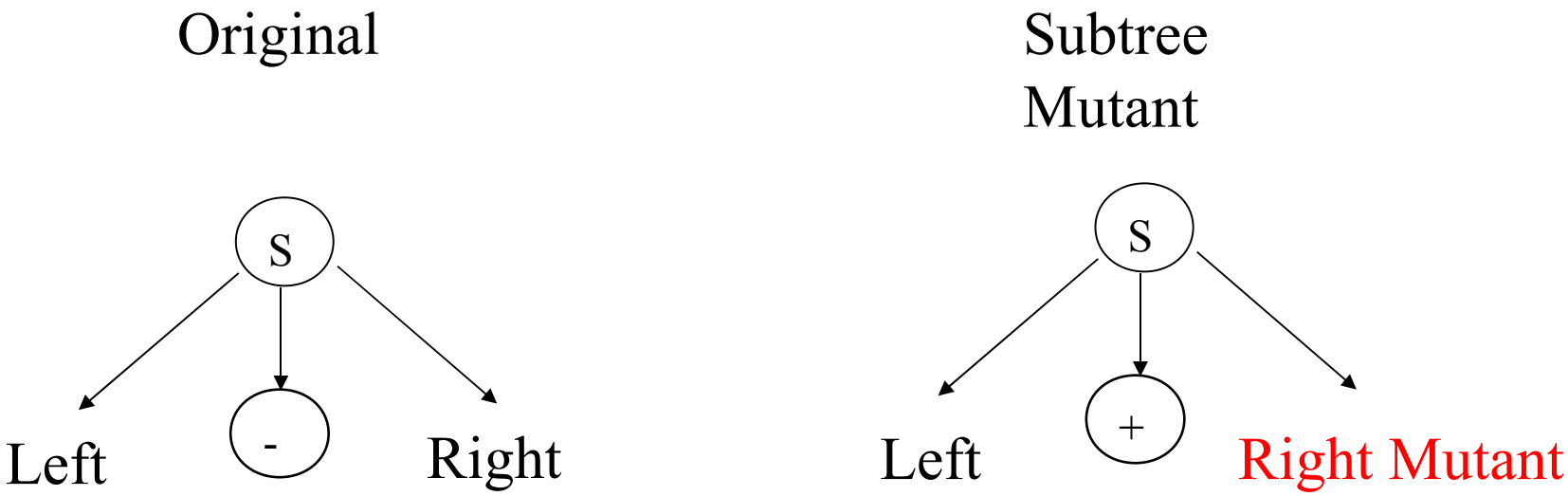
Mutants



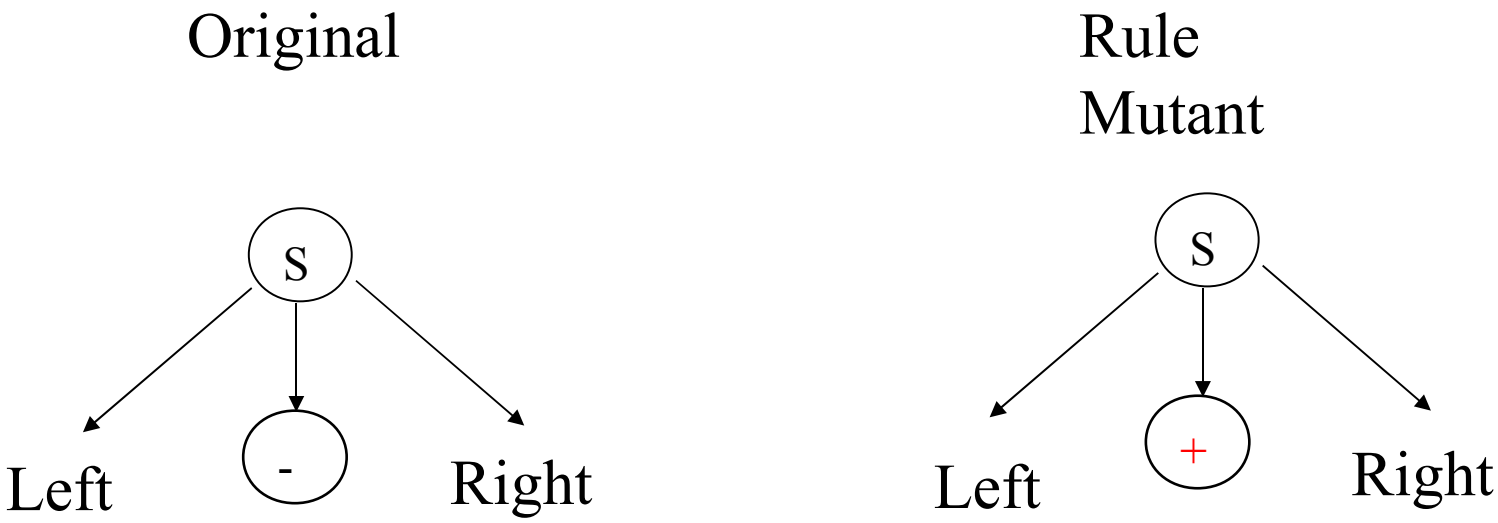
Mutant construction: example



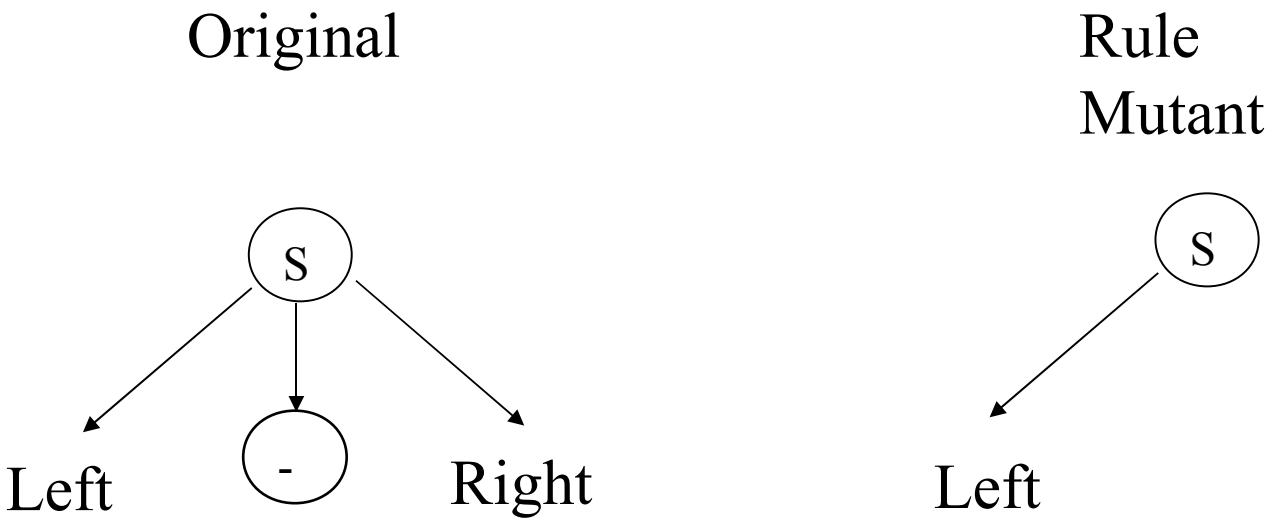
Mutant construction: example



Mutant construction: example

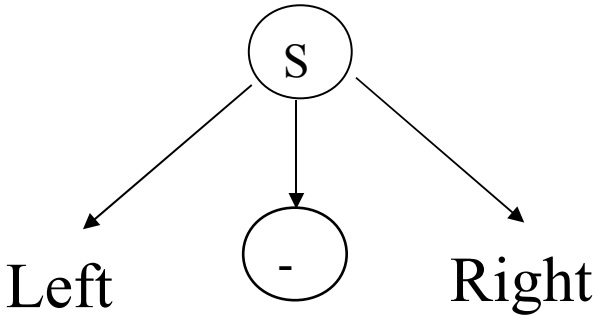


Mutant construction: example

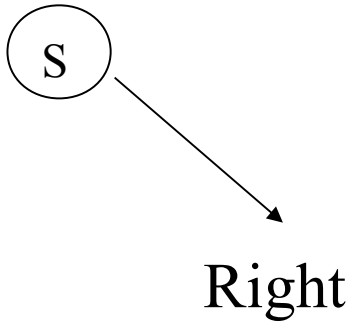


Mutant construction: example

Original



Rule
Mutant



Mutant construction : example

- Original tree : [1 + 2] – 3

- *Terminal replacement mutants*: [0 + 2] - 3, [2 + 2] - 3, [1 + 1] - 3, [1 + 3] - 3, [1 – 2] – 3, [1 + 2] - 2, [1 + 2] - 4, [1 + 2] + 3,

- *String deletion mutants*: 1 - 3, 2 – 3, 1 + 2, 3

Mutation Testing of P systems

- Construct the Kripke structure equivalent of the P system
- Mutate the Kripke structure
- Derive test values to kill mutants (through model checking)

Mutant P system construction

- Consider a 1-membrane P system $\Pi = (V, [_1]_1, w, R)$, $R = \{r_1: u_1 \rightarrow v_1, r_2: u_2 \rightarrow v_2, \ldots r_m: u_m \rightarrow v_m\}$

- A multi-membrane without dissolving rules can be transformed into an equivalent 1-membrane P system by renaming symbols

- A derivation in the maximal parallelism mode: $u \rightarrow v$ iff $\exists\ n_1, \ldots, n_m\ .\ \text{MaxParal}(u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m) \wedge \text{Apply}(u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$

- Given initial configuration u, rules $r_1, \ldots, r_m$ are applied for $n_1, \ldots, n_m$ times in maximal parallelism mode and the new configuration is v.

Kripke structure

- A Kripke structure over a set of atomic propositions AP is a four tuple M = (S, H, I, L), where
  - S is a finite set of states
  - $I \subseteq S$ is a set of initial states
  - $H \subseteq S \times S$ is a transition relation that must be total
  - $L: S \rightarrow 2^{AP}$ is an interpretation function
- Usually, the Kripke structure representation of a system results by giving values to every variable in each configuration of the system.

Mutant P system construction

• States: bounded vectors from N^k, where k is the number of symbols (finite number of configurations).

• Transition relation: $u \rightarrow v$ iff $\exists\ n_1, \ldots, n_m$. MaxParal($u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m$) $\wedge$Apply($u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m$)

• 2 additional states: Halt (successful termination) and Crash (unsuccesful termination) and transitions to these 2 extra states.

• Define an appropriate context-free grammar for the transition relation.

• Note: The primitives of the grammar depend on the level at which testing is to be performed, e.g. for high level testing the predicates MaxParal and Apply can be considered as terminals.

Example : 1-membrane P system with 2 rules

- Nonterminals: S, S_1, S_2, U, V, U_1, U_2, V_1, V_2

- Terminals: bounded vectors from N^k, extra states Halt and Crash, predicates Apply and MaxParal, logical conectors and quantifiers, etc.

- Production rules

$S \rightarrow \neg S \mid S \wedge S \mid S \vee S \mid \text{true} \mid \exists\ n_1\ .\ S_1$

$S_1 \rightarrow \exists\ n_2\ .\ S_2 \mid S_2 \wedge S_2 \mid \text{MaxParal}(U, U_1, V_1, N_1, U_2, V_2, N_n) \mid \text{Apply}(U, V, U_1, V_1, N_1, U_2, V_2, N_2)$

Rules that transform U, V, U_1, V_1, N_1, U_2, V_2, N_2 into vectors from N^k

Mutant P system construction

- Define mutants for the production rules, e.g.
  - logical mutants: change logical connectors, negate expressions, omit operands, replace $\exists$ by $\forall$, etc.

  - function application mutants: change the value/order of parameters (as long as the resulting expression remains syntactically correct), etc

- Mutants obtained (among others)
  - P systems in which one rules is changed. (Note: only one rule is mutated at a time)
  - P system in which rules r_1 and r_2 are interchanged (this will result in an equivalent mutant)
  - A system with r_1 and r_2 are, but which are not applied in a maximal parallel mode
  - Other erroneous Kripke systems which may have no P system equivalent

Further Work

- Fully formalize P system mutants

- Investigate relationship between
  - rule coverage based testing
  - finite state machine based testing
  - mutation based testing

- Use evolutionary techniques for modelling, testing and evolving P systems: Research project funded by Romanian Research Council for Higher Education (CNCSIS): co-investigator Marian Gheorghe (University of Sheffield).

- Develop appropriate tools

- Assess benefits and limitations w.r.t other similar verification and validation approaches

Thanks!

Questions?