Testing Based on P Systems - An Overview

Marian Gheorghe[1,2]
&
Florentin Ipate[2]

[1]University of Sheffield
[2]University of Piteşti

Summary

- Software testing
  - needs
  - techniques

- P systems testing
  - coverage principle
  - grammar-like
  - finite state machine (X-machine)
  - model checking

- Further work and conclusions

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, some special issues of BioSystems, Handbook of MC, Scholarpedia

• Software packages developed for some of these applications

(P system web page http://ppage.psystems.eu) - P-lingua, Metabolic P systems, Stochastic P systems, IBW, P systems for reaction kinetics.

• Both formal verification and *testing* have been applied for some classes of P systems

Software testing

Software testing

• is the process of checking software, to verify that it *satisfies its requirements* and to *detect errors*.

• consists of, 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)

Major testing activity

• Test case (test suite) generation: selection of test values most likely to find faults

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 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

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

```
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

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

```
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

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

```
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

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

```
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

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

```
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)**

Coverage methods

• In *structural testing* a program is represented as a directed graph and various coverage criteria can be defined:

– Statement (node) coverage
– Branch (decision) coverage
– Multiple condition coverage
– etc

• Coverage criteria can also be used in *functional testing* (especially for model based testing), e.g., *rule coverage* for specifications represented as context-free grammars – each production rule of the grammar is applied at least once; compilers, syntax-oriented tools.

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: state/transition cover, UIO, W, Wp, etc.

Rule coverage based P system testing

Grammar-like 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 *(simple rule coverage)*

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

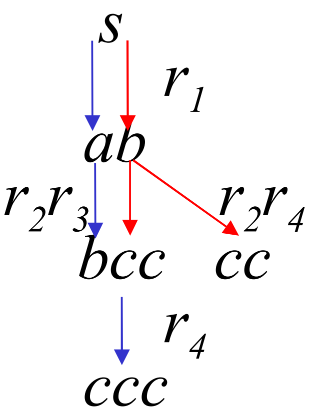
- Test application – checks whether all elements of the test set are computed by the implementation

- It will be considered that a P system model is given and an implementation of it is going to be tested

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

Example

Π has r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$ and s initial multiset



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

or

$T'=\{ab, bcc, cc\}; \{bcc, cc\}$

T or T' - rule coverage

Implementations:

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

Π_2: r_1: $s \rightarrow ab$; r_2: $a \rightarrow bc$; r_3: $a \rightarrow c$; r_4: $b \rightarrow c$ // computes both T, T'

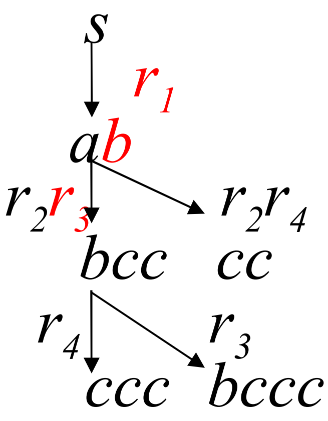
Obs. *bccc* is not computed by Π_2 but is produced by the model Π

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 $a \rightarrow c$



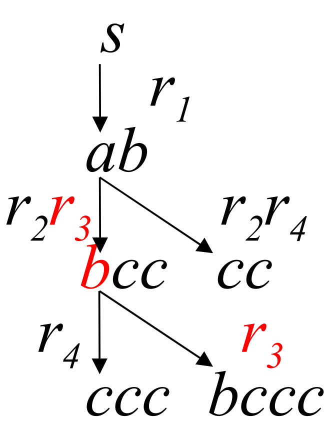
context-dependent rule coverage

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 $a \rightarrow c$

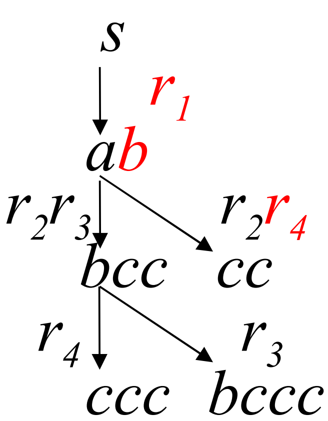


context-dependent rule coverage

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 $a \rightarrow c$

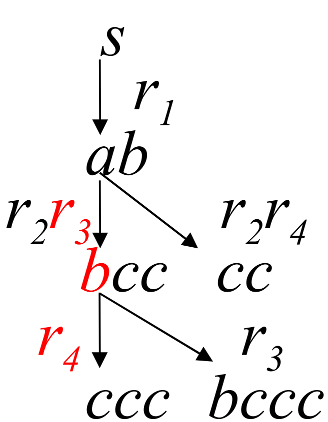


context-dependent rule coverage

Context-dependent rule coverage

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

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



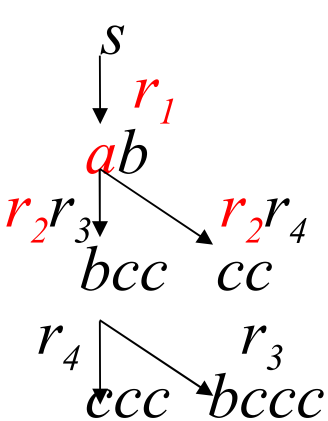
context-dependent rule coverage

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

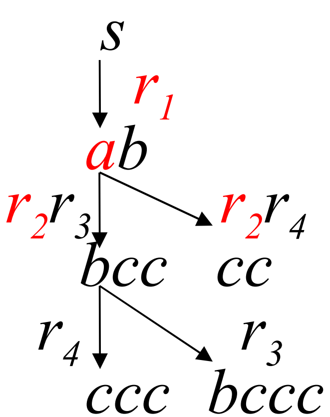
Π_2: r_1: $s \rightarrow ab$; r_2: $a \rightarrow bc$; r_3: $a \rightarrow c$; r_4: $b \rightarrow c$ //don't compute *bccc*

Context-dependent rule coverage. Test set

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

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

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



context-dependent rule coverage

Test sets: $T=\{bcc, cc, ccc, bccc\}$; $\{cc, ccc, bccc\}$

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

Testing based on Finite State Machine*

• Build all the computations of the P system for a finite sequence of steps, k – represented as a tree

• 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 minimal finite state machine that accepts all sequences in U and possibly sequences that are longer than any word of U (Theorem 4*)

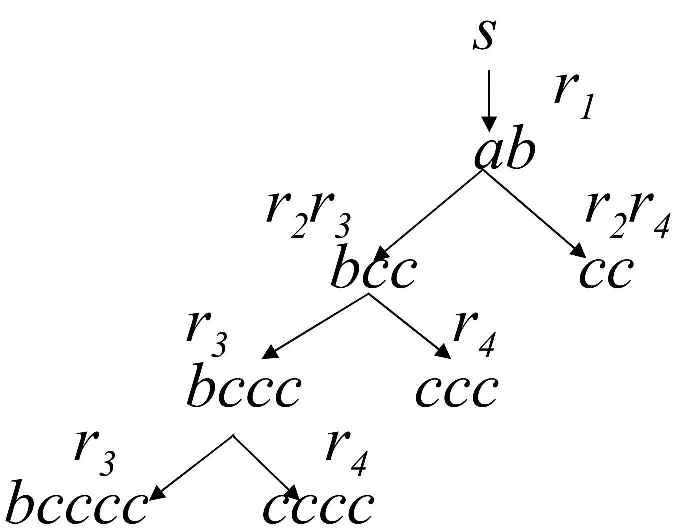
• Generate a test set, T, over the P system's alphabet V, for a certain coverage principle (e.g. state or transition coverage)

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

*F Ipate, M Gheorghe: Finite state based testing of P systems, Natural Computing, 8(2009).

All computations for a given k

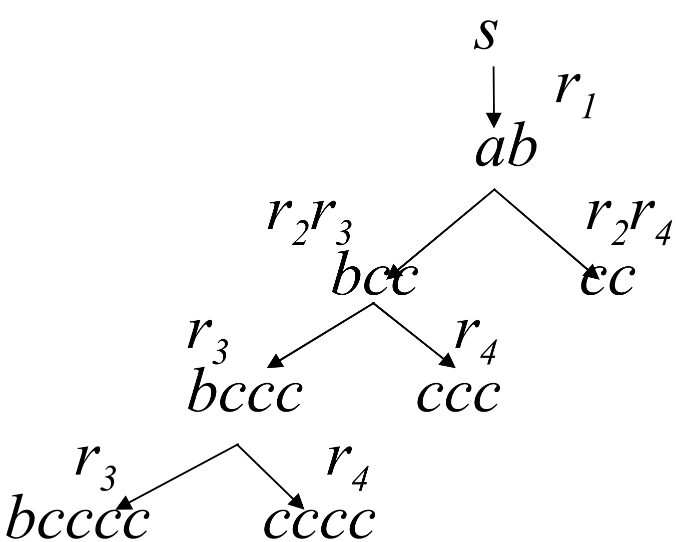
Example. For Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$

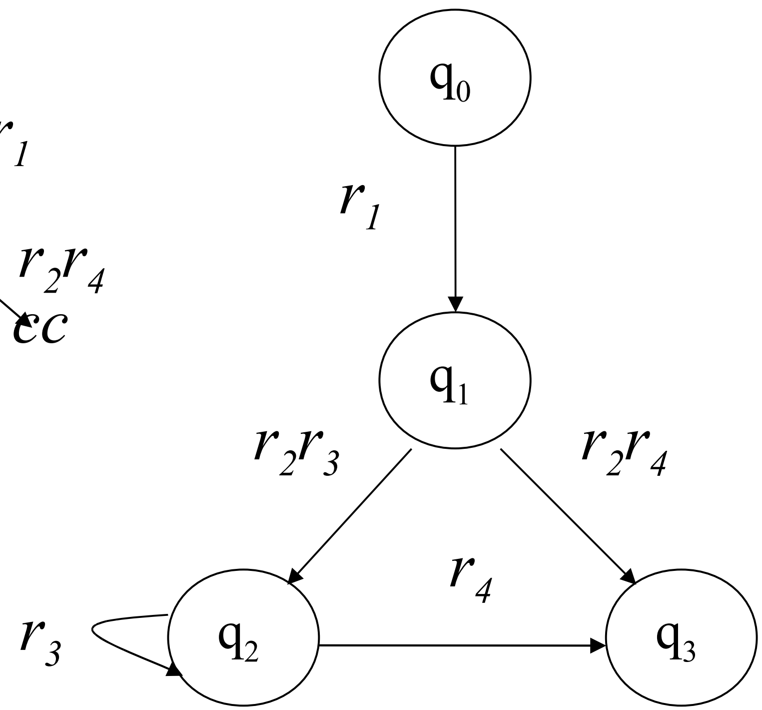


$k = 4$ steps, obtain Dt – a DFA over the set of labels defining the multisets of rules applied $\{r_1, r_2r_3, r_2r_4, r_3, r_4\}$ accepting L_{Dt}

DFC for L_{Dt}

Example. For *Π*, r_1: $s \to ab$; r_2: $a \to c$; r_3: $b \to bc$; r_4: $b \to c$; *DFA* is





DFC, *M*, for L_{Dt}

In general DFC (4) has less states than DFA (8) (also true for minimal FSM's)

Coverage criteria for DFC Automata

- Specification is a finite automaton with all states final.

- **State coverage** S: for each state q there is $u \in S$ and a path that reaches q such that u is computed from w through a computation defined by the path.

Transition coverage T: for each state q and each valid label of a transition from q (to q') there is $u \in T$ and a path that reaches q' and includes q such that u is computed from w through a computation defined by the path.

W method for DFC Automata

• Specification is a finite automaton with all states final.

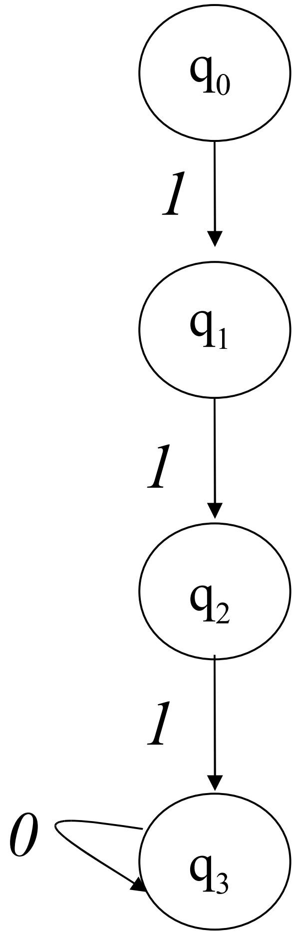
• Aim to show implementation behaves identically with the specification for all sequences of length less than or equal to an upper bound N.

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

• W method for DFC: *sequences of minimum possible length* are chosen to reach states or distinguish between states: ***Proper*** *state cover* and **Strong** *characterization set* ($\lambda \in W$)

• Test suite: $(S\ A[m\text{-}n+1]\ W) \cap A[N]$, where $A[k] = \{\lambda\} \cup \ldots \cup A^k$

Test set components. Example



$S = \{\lambda, 1, 11, 111\}$

$T = \{\lambda, 1, 11, 111, 1110\}$

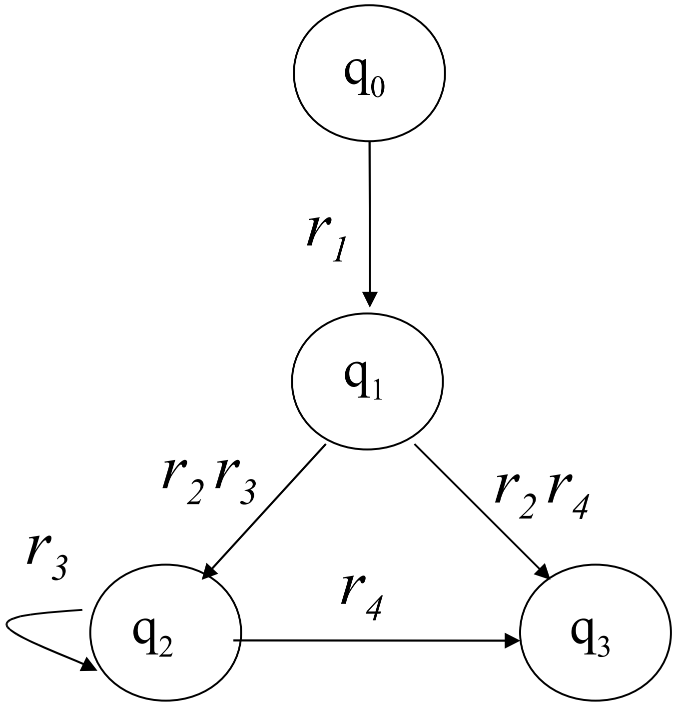
$W = \{\lambda, 1, 11, 111\}$

Incorrect

$S = \{\lambda, 1, 11\}$ – q_3 not covered

$W = \{\lambda, 111\}$

Example. DFC for *Π*



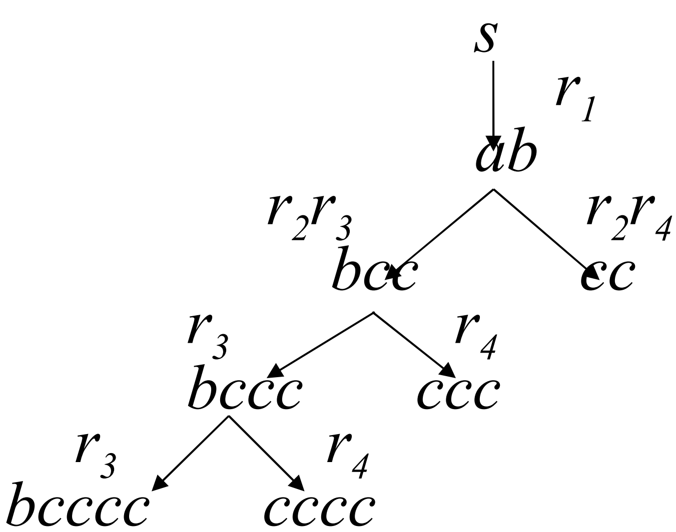
S = {λ , r_1, $r_1 . r_2r_3$, $r_1 . r_2r_4$}

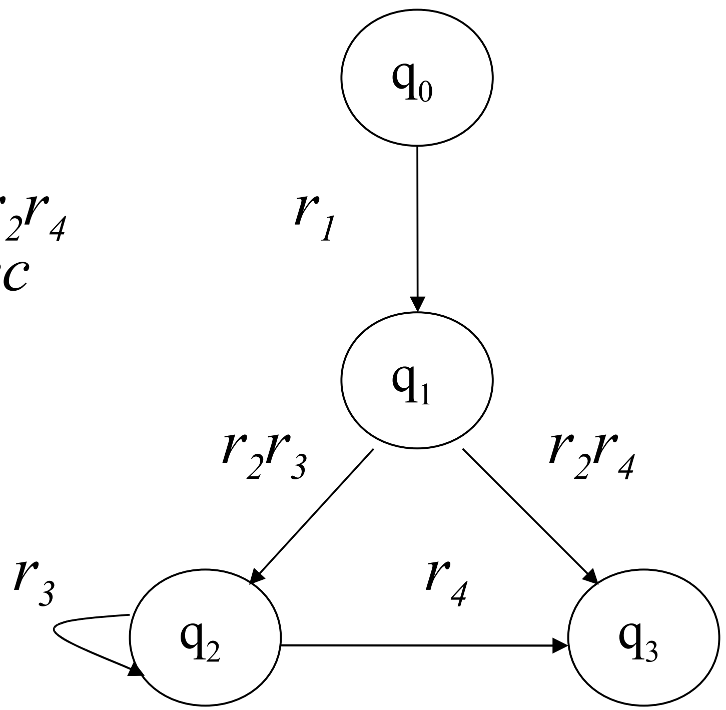
T = {λ , r_1, $r_1 . r_2r_3$, $r_1 . r_2r_4$, $r_1 . r_2r_3 .r_3$, $r_1 . r_2r_3 .r_4$}

W = {λ , r_1, r_2r_3, r_3}

Test set for state cover - S

Example. For Π, r_1: $s \rightarrow ab$; r_2: $a \rightarrow c$; r_3: $b \rightarrow bc$; r_4: $b \rightarrow c$; *DFA* is



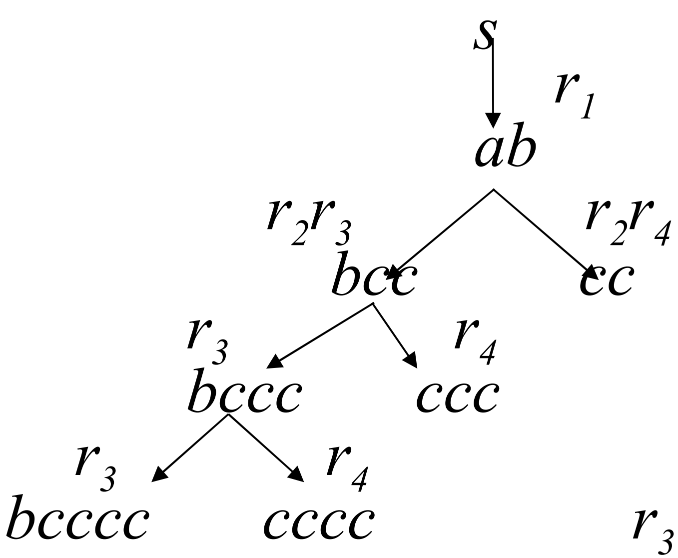


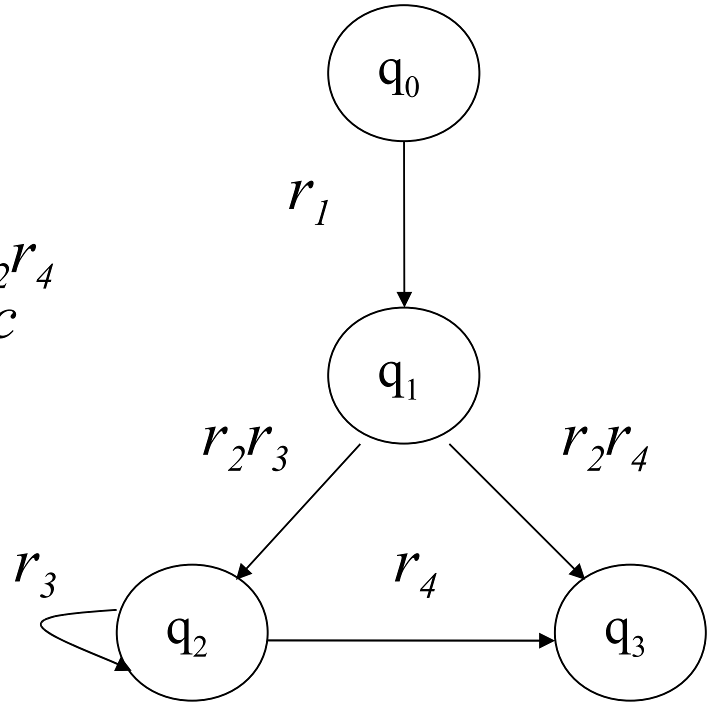
DFC, *M*, for L_{Dt}

$S = \{\lambda, r_1, r_1 . r_2r_3, r_1 . r_2r_4\}$; $Ts=\{s, ab, bcc, cc\}$

Test set for transition cover - T

Example. For *Π*, r_1: *s*→*ab*; r_2: *a*→*c*; r_3: *b*→*bc*; r_4: *b*→*c*; *DFA* is





DFC, *M*, for L_{Dt}

T = {λ , r_1, $r_1 . r_2r_3$, $r_1 . r_2r_4$, $r_1 . r_2r_3.r_3$, $r_1 . r_2r_3.r_4$}; *Tt={s, ab, bcc, cc, bccc, ccc}*

Results so far...

For grammar-like and FSM based testing strategies, test sets for Π

$T_1 = \{ab, bcc, ccc\}$ – simple rule coverage;
$T_2 = \{bcc, cc, ccc, bccc\}$ – context-dependent rule coverage;
$Ts_1 = \{s, ab, bcc, cc\}$ – state cover, k=3, 4, ...;
$Ts_2 = \{s, ab, bcc, cc, bccc, ccc\}$ – transition cover, k=3, 4, ...;

$T_1 \subset T_2 \subset Ts_2$; $Ts_1 \subset Ts_2$

• Context-dependent is better than simple rule coverage and transition cover outperforms state cover

• FSM based testing is better supported by FSM theory, produces in general better results, but depends on the number of computation steps (k); it requires more effort (build the DFC and then test sets)

• More elaborated test sets – take sequences of multisets (version of $T_1 = \{ab \cdot bcc, ab \cdot ccc\}$)

Empirical analysis of the two approaches*

- Context dependent rule coverage achieves better detection than simple coverage (100% vs 98.75% in some cases), but this is way below the increase in the size complexity of the test set
- Both achieve better fault detection for sequences of multisets (increase between 3.75% to 21.06%)
- The performance of FSM based approaches depend heavily on k (for state coverage and k=2, values as low as 52.63% fault detection; for transition coverage and high values for k, it achieves at least 78.94% fault detection)
- When sequences of multisets are utilised, 100% in many case is achieved, irrespective of the approach

*R Lefticaru, M Gheorghe, F Ipate: An empirical evaluation of P system testing techniques, Natural Computing (to appear 2010)

X-machine (Generalised FSM) based testing

• X-machine based testing is well elaborated (more than 15 years) and codification of various classes of P systems as X-machines provided (Aguado et al, 2001; Kefalas et al, 2003)

• Testing P systems using non-deterministic stream X-machines studied (Ipate, Gheorghe; ENTCS, 2008) – X-machine built similarly to DFA (a finite number of computation steps)

• Unfortunately the general theory of X-machines and the methodology of building X-machines from given P systems DO NOT provide a way to define suitable testing techniques for P systems as the X-machine representation does not adequately replicate the P system – many micro-steps

Model based testing

• Above presented approaches – grammar-like and FSM based testing, are model based techniques: the generation of the test set utilises a certain model

• Two main difficulties faced
- FSM and X-machine approaches require another model
- It involves building suitable algorithms for test sets

• Question: are there other techniques that help building the test sets from a generic model?

Model based testing

• Above presented approaches – grammar-like and FSM based testing, are model based techniques: the generation of the test set utilises a certain model

• Two main difficulties faced
•   FSM and X-machine approaches require another model
•   It involves building suitable algorithms for test sets

• Question: are there other techniques that help building the test sets from a generic model?

• Yes... model checking (Kripke structure representation) through counterexamples for properties that do not hold

Test suite using model checking

A test suite is obtained by following the 3 steps (Fraser et al, 2009):

• Define the *test purpose* by identifying a testing criterion as *features* to be tested (reaching a state, traversing a sequence of states, getting a value, verifying a condition)

• The *features* are specified as temporal logic formulas and then converted into *never-claim* conditions or *trap* properties; Examples: G !(state = s) or G !(x = val)

• The model checker verifies whether the never-claim or trap property holds. It it is false a counterexample is returned – this gives the exact path to state s or to where x becomes val

• Additionally, the P system is converted into a Kripke structure

Kripke structure

- A system $M = (S, H, I, L)$, where
  - S is a finite set of states
  - $I \subseteq S$ – initial states
  - $H \subseteq S \times S$ – left-total transition relation (for any s in S there is s' in S such that (s,s') in H)
  - L is an interpretation – associating toeach state a set of atomic propositions true in the state

Given a P system Π, a Kripke structure M_Π associated with Π is constructed using the predicates

$MaxPar(u, u_1, v_1, n_1, \ldots u_m, v_m, n_m)$ - m rules $u_i \to v_i$ are used n_i times, in maximal parallel mode

$Apply(u, v, u_1, v_1, n_1, \ldots u_m, v_m, n_m)$ – v is obtained by the rules above

F Ipate, M Gheorghe, R Lefticaru: Test generation from P system using model checking, JLAP, 2010

F Ipate, M Gheorghe et al: An integrated approach to P systems formal verification (CMC11)

Kripke structure - The basis of testing

• Similar to FSM based testing a model of a system, as a Kripke structure, K, is given and a (potentially faulty) model of the implementation under test, K', is provided

Theorem 4 (Ipate, Gheorghe, Lefticaru)

(i) if a a property is satisfied then the implementation includes all the paths of the specification

(ii) if the property is false then there is a path which has a finite prefix in K and K' but in the next state the property is only true in the model K, of the system

Represent the P system as a Kripke structure

• Convert various classes of P systems (with rewriting and communication (non)-cooperative rules, with electrical charges, with dissolving rules; more than one compartment; maximal parallelism or asynchronous mode) to NuSMV (Ipate et al, 2010, CMC11 presentation etc); basic principles:

• Kripke structure states are P systems multisets – a finite subset; these are computed based on *MaxPar* predicate (for maximal parallelism)

• Transitions between states are obtained utilising the *Apply* predicate

• The model should contain some terminal state and an unexpected halting state – when some conditions are not fulfilled

Test set construction – step 1

- In this first step a *testing criterion* is introduced – use simple and context-dependent rule coverage, as defined for grammar-like testing approach

- We can test not only "rule coverage" criteria, but also directly states – for instance whether the number of *a* > *threshold*

- All these criteria form the basis of the test set generation

Test set construction – step 2

• Transform these *testing criteria* into *never-claim* or *trap* properties by negation using LTLformulas

• For each rule $r_i \in R$ to test if it appears in a computation (rule coverage): G!((ni >0) & (state=running)) – where ni means the number of appearances of the rule r_i and running is one of the finite states considered

• To test that $r_i \in R$ appears in the context of $r_j \in R$ (context-dependent rule coverage): G!((ni >0) & X(nj>0) & (state=running))

• We can test that on a given pathway the number of a > *threshold*
G!((a >threshold) & (state=running))

...

Test set construction – step 3

- When the LTL formula is false, a counterexample is returned

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

G!((n1 >0) & X(n2>0) & (state=running)) -- checks that r_2 appears in the context of r_1 in running state

A counter-example is returned corresponding to the computation

$$s \Rightarrow ab \Rightarrow cc$$

utilising r_1 first and then r_2, r_4

Test set generation - Example

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

G!((ni >0) & (state=running)) – each rule is reached ($i=1..4$)
G!((ni >0) & X(nj>0) & (state=running)) – each contextual pair (ex r_1, r_2)
G!((ni >0) & (state=running) & F(state=halt)) – each rule is reached in a terminal computation ($i=1..4$)
G!((ni >0) & X(nj>0) & (state=running) & F(state=halt)) – each contextual pair (ex r_1, r_2) tested in a terminal computation

Integrity checks

G((state=running) ->(0<=a & a<=Max)) – *a* stays within the domain
G((state=running) ->(0<=n2 & n2<=Sup)) – n2, the number of applications of r_2 is within imposed limits

Limitations and some solutions

- Scalability (NuSMV can not cope with bigger domains for variables, >50, or many iterations, >25; solution – use other tools, SPIN – Ipate et al; 2010)

- Error prone when dealing with complex specifications (solution: automatic way of generating LTL specifications – Ipate, Gheorghe, Lefticaru; 2010)

- Readability of the results returned (solution: adequate tools)

- Limited repertoire of coverage criteria (testing strategies)

- Limited approximation of the system representation – considering a fixed number of steps

- Integration with existing P system development environments (P-lingua) – under consideration

Conclusions and further work

• Basic classes of P systems and simple testing criteria investigated

• Model based testing strategies adapted to P systems specifications (theoretical basis elaborated, some empirical analysis provided, promising results obtained)

• Investigate further testing options – initial candidates: mutation testing (Ipate, Gheorghe; 2009), evolutionary techniques for testing and evolving P systems: Research project (CNCSIS), PI- Ipate, co-I's – Gheorghe, Lefticaru & investigations on state based models (Lefticaru, Ipate; 2008, 2009)

• Develop appropriate tools

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

Thanks!

Questions?