Search Based Software Engineering in Membrane Computing

work in progress

Marian Gheorghe[1], Florentin Ipate[2], **Ana Țurlea**[2]

[1] University of Bradford, UK

[2] University of Bucharest, Romania

ana.turlea@fmi.unibuc.ro

Table of contents

1. Motivation
2. Search-Based Software Engineering
3. SBST approaches
4. Genetic Algorithms
5. Identifiable Kernel P-Systems
6. KPWorkbench
7. Proposed Approach
8. Conclusion and future work

Motivation

Motivation

Considering the following context:

- an Identifiable kP-system with one compartment;
- a set of N evolution steps: each step i contains a list of rules that can be applied on the configuration obtained after step $i-1$, using some inputs.

✓We want to automatically generate the inputs nedeed to trigger the given evolution steps.

Motivation

A common testing methodology

- automatic generation of inputs used to create given scenarios.

Software systems testing

- very important stage from the development process.
- high percent of the overall software development cost.

Manual testing

- expensive,
- time consuming,
- more likely to produce errors.

Motivation - P Systems Testing

Why Testing?

- Testing is finding out how well something works.
- Under certain conditions, one needs to ensure that the implementation of a system conforms to its specification.

Why P system testing?

- Membrane computing: very fast growing field
- Rapid development of many tools and P system simulators
- This issue raises the problem of testing all these implementations of P systems.
- The models are complex: non-deterministic, parallel, can have polarizations (charges), transformation - communication rules, membrane creation/division etc.

Motivation

✓ Automated testing applied to P-systems.

How we do this:

- genetic algorithms → generate the inputs.
- kPWorkbench tool → simulate the evolution of the kP-system.

Outcomes of the testing method

For a given kP-system and an evolution scenario → generate the multisets that trigger the scenario.

Search-Based Software Engineering

Search-Based Software Engineering (SBSE)

Software engineering problems → *optimization* problems + metaheuristic search techniques (GAs, PSO, SA)

Applications: software testing, requirements engineering, quality assessment, project planning, cost estimation and many others.

Search-Based Software Testing (SBST) Methodology

- is an automated search of a potentially large input space, guided by a problem-specific fitness function

- **Search space = ?**
    - Depending on the problem, the input parameters of the program. Codification?
- **Fitness function = ?**
    - The fitness function guides the search to the test goal
    - It scores different inputs to the system according to the test goal
- Which search algorithms to use? Global, local, hybrid?

SBST approaches

SBST approaches

- SBST
  - Functional Testing
  - Structural Testing

- SBST
  - Local Search
  - Global Search
  - Hybrid Search

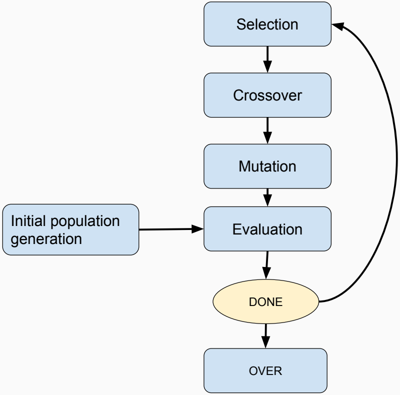
- SBST
  - Test cases generation
  - Test suite generation

- SBST
  - Single-Objective
  - Multi-Objective

Genetic Algorithms

Genetic Algorithms

- class of evolutionary algorithms;
- use selection, recombination (crossover) and mutation, applied on a population of potential solutions, called chromosomes (or individuals) .



Identifiable Kernel P-Systems

Kernel P Systems

Definition (Kernel P systems)

A *kP system* of degree n is a tuple $k\Pi = (A, \mu, C_1, \ldots, C_n, i_0)$, where

- A is a finite set of elements called objects;
- $\mu =$ the membrane structure (a graph (V, E): V is a set of vertices representing compartments and E is a set of edges - links between compartments);
- $C_i = (t_i, w_{i,0}) =$ compartment: consisting of a *compartment type*, $t_i \in T$ and an initial multiset, $w_{i,0}$ over A; the type $t_i = (R_i, \rho_i)$ consists of a set of evolution rules, R_i, and an execution strategy, ρ_i;
- i_0 is the output compartment where the result is obtained.

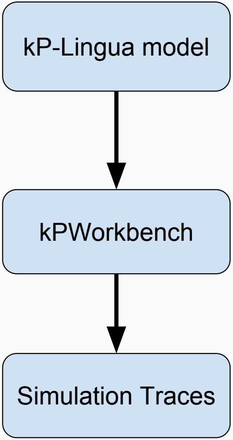
Definition

Two rules $r_1 : x_1 \rightarrow y_1\{g_1\}$ and $r_2 : x_2 \rightarrow y_2\{g_2\}$ from R_1, are said to be *identifiable* if there is a configuration c where they are applicable and if $c \Rightarrow^{r_1} c'$ and $c \Rightarrow^{r_2} c'$ then $b(r_1) = b(r_2)$.

KPWorkbench

KPWorkbench - A software framework for Kernel P systems

- Integrated software suite aimed to provide tool support for kP-systems.
- Provides tool for modelling, **simulating** and verifying kP-systems.
- Simulation traces → the evolution of the system over time


kP-Lingua model

kPWorkbench

Simulation Traces

Proposed Approach

Proposed Approach

The idea:

Input: an identifiable kP-system with one compartment and a set of evolution steps (with list of rules)

Output: a list of multisets corresponding to each step, that will be passed as input to trigger the scenario.

We need to automatically generate the multisets and to simulate the system.

Proposed Approach - How to simulate the system?

Manually?

We can evolve step by step the system: At each step, having the previous obtained configuration and adding a new multiset as input → apply rules and obtain new configuration.

Automatically?

We can simulate the whole evolution of the system using kPWorkbench, passing all inputs from the begining. The only problem remains how to pass during the simulation the corresponding inputs to each step.

✓ **Automatically**

- create a new compartment type with rules that passes each input multiset to the first compartment type;

Proposed Approach

Algorithm Steps:

- Preprocessing the kP-system (kpl file and other configuration files).
- Run the Genetic Algorithm using the following steps:
  1. Initialize the population (Size N).
  2. Evaluate objective functions.
  3. Apply Crossover on pairs of parent chromosomes to create new individuals.
  4. Apply Mutation.
  5. Evaluate objective functions simulating the system.
  6. Combine parent and child populations.
  7. Apply Selection.
  8. If the stopping criteria is met, print the best individual and stop, otherwise go to step 3 and repeat.

Preprocessing the kP-system

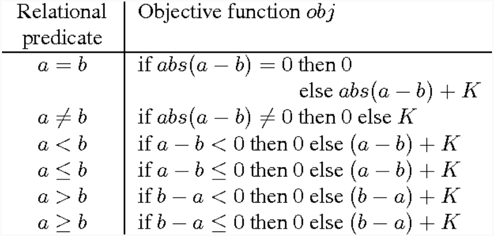
- kP-Lingua file;
- XML file:
    - the list of Ns steps and for each step the list of rules;
    - the input multisets domain: the symbols and a range for the number of occurences for each symbol;
    - the initial configuration of the system;
- Restrictions:
    - one compartment
    - only rewriting and communication rules
    - identifiable transitions
    - maximal parallelism strategy

Genetic Algorithm Configuration

- **Input Symbols**: N_s letters from $\{'a'..'z'\}$
- **Population**: $N = 50$ chromosomes/individuals;
- **Chromosome**: a list of multisets $c = (g_1, g_2, ..., g_{Ns})$, $g_i, 1 \leq i \leq Ns$ is the input corresponding to step i.
- **Genes** are represented as List of Strings $g_i = [n_1 a_1, n_2 a_2, \ldots n_{N_s} a_{N_s}]$.
- **Genetic Operators**:
  - Selection - Best Solution Selector
  - Mutation - replace random g_i, replace random n_i from random g_j, switch values between random g_i and g_j.
  - Crossover - one point crossover

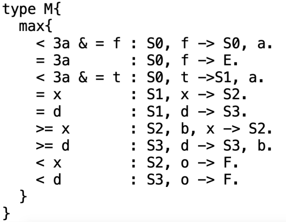
The Objective Function

- generate the input type compartment;
- create the kpl file containing the input system, the new compartment type and the link between them;
- run kPWorkbench;
- parse the output file: evolution traces;
- check if it contains the same steps as we needed and adapt the formula: approach level + normalised branch level.
    - *approach level*: records the number of steps that are not ok;
    - *branch distance*: uses Tracey's objective functions for relational predicates (using the rules' guards) - normalised in $[0, 1)$

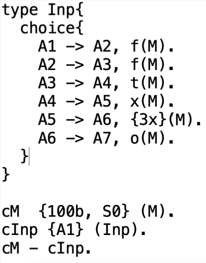


Relational predicate	Objective function obj
$a = b$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + K$
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a-b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a-b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b-a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b-a) + K$

Experiments



```
type M{
  max{
    < 3a & = f : S0, f -> S0, a.
    = 3a       : S0, f -> E.
    < 3a & = t : S0, t ->S1, a.
    = x        : S1, x -> S2.
    = d        : S1, d -> S3.
    >= x       : S2, b, x -> S2.
    >= d       : S3, d -> S3, b.
    < x        : S2, o -> F.
    < d        : S3, o -> F.
  }
}
```



```
type Inp{
  choice{
    A1 -> A2, f(M).
    A2 -> A3, f(M).
    A3 -> A4, t(M).
    A4 -> A5, x(M).
    A5 -> A6, {3x}(M).
    A6 -> A7, o(M).
  }
}

cM  {100b, S0} (M).
cInp {A1} (Inp).
cM - cInp.
```

Steps executed:

1. rule 0
2. rule 0
3. rule 2
4. rule 3
5. rules 6, 6, 6
6. rule 9

Conclusion and future work

Conclusion and future work

Preliminary Results

- the algorithm finds very quickly a value very close to the solution;
- still working at the algorithm;

Conclusion

We extend the SBST methodologies to kP-Systems.

Future work:

- Finish the implementation → find the solution.
- Generalise the algorithm - remove the restrictions on the kP-system.

Thank you!