Search Based Software Engineering in Membrane Computing

Ana Ţurlea[1], Marian Gheorghe[2], Florentin Ipate[1]

[1] Faculty of Mathematics and Computer Science and ICUB
University of Bucharest, Bucharest, Romania
ana.turlea@fmi.unibuc.com, florentin.ipate@ifsoft.ro
[2] School of Electrical Engineering and Computer Science,
University of Bradford, Bradford, UK
m.gheorghe@bradford.ac.uk

Summary. This paper presents a testing approach for kernel P Systems (*kP systems*), based on test data generation for a given scenario. This method uses Genetic Algorithms to generate the input sets needed to trigger the given computation steps.

Keywords: membrane computing; kernel P systems; testing; genetic algorithms: test data generation.

1 Introduction

Membrane Systems [17], now known as P Systems, were founded by Gheorghe Păun in 1998 [15, 16]. Initially inspired by the structure and functioning of the living cells, the field has been developed very fast and different types of *P systems* being investigated. *Kernel P systems* (*kP systems*, for short), have been introduced in [2]. These systems can be simulated using a software framework, called *kPWorkbench* [1] or some earlier variants (so called *simple kP systems*) using P-Lingua and the MeCoSim simulator [3]. Having many computational models with different software implementations, associated with various applications, it is very important to develop testing methods, to check that the implementation agrees with the system specification. This testing methodology is called conformance testing, which tries to find the differences betweem the behaviour of an implementation and its specification. The testing task is not trivial, given the fact that the models are parallel and non-deterministic. Previous works on P systems testing include testing cell-like P systems with methods like finite state-based inspired [6], stream X-machine based testing [7], mutation testing for evaluating the efficiency of the test sets [11], model-checking based testing [8] and testing identifiable kernel P systems using X-machines [4].

Automated test data generation is a topic of interest in software engineering community. There are many evolutionary testing approaches that generate test

date from code, finite state machines and other models, but there are no applications in membrane computing community.

In kernel P systems, we can simulate the evolution of the model for a given number of steps, starting with an initial multiset. We can change the evolution of the system by adding new multisets as inputs for each evolution step.

This paper presents a testing approach for kernel P systems, using genetic algorithms to generate test data that leads to a given set of computation steps.

In Section 2 we present some basic information about kP systems, evolutionary functional testing, genetic algorithms, search based testing for extended finite state machines. Section 3 describes the kP system type used for testing, the configuration of the algorithm and some experimetal results. In Section 4 we present conclusions and future work.

2 Preliminaries

In this section we will present some basic details about kernel P systems and Search Based Testing using Genetic Algorithms. We will also present some approaches that use evolutionary algorithms to test Extended Finite State Machines.

2.1 Kernel P systems

In the following we will give a formal definition of kernel P systems (or kP systems) [2]. We start by introducing the concept of a *compartment type* utilised later in defining the compartments of a kernel P system.

Definition 1. *T is a* set of compartment types, *$T = \{t_1, \ldots, t_s\}$, where $t_i = (R_i, \sigma_i)$, $1 \leq i \leq s$, consists of a set of rules, R_i, and an execution strategy, σ_i, defined over $Lab(R_i)$, the labels of the rules of R_i.*

Definition 2. *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;*
- *μ defines the membrane structure, which is a graph, (V, E), where V is a set of vertices representing components (compartments), and E is a set of edges, i. e., links between components;*
- *$C_i = (t_i, w_{i,0})$, $1 \leq i \leq n$, is a compartment of the system consisting of a* compartment type*, t_i, from a set T and an initial multiset, $w_{i,0}$ over A; the type $t_i = (R_i, \sigma_i)$ consists of a set of evolution rules, R_i, and an execution strategy, σ_i;*
- *i_0 is the output compartment where the result is obtained.*

Within the general kP systems framework, the following types of evolution rules have been considered so far:

- *rewriting and communication* rule: $x \rightarrow y\{g\}$, where g represents a **guard**, $x \in A^+$ and $y \in A^*$, where y is a multiset with potential different compartment type targets (each symbol from the right side of the rule can be sent to a different compartment, specified by its type; if multiple compartments of the same type are linked to the current compartment, then one is randomly chosen to be the target). Unlike cell-like P systems, the targets in kP systems indicate only the types of compartments to which the objects will be sent, not particular instances (for example, $y = (a_1, t_1) \dots (a_h, t_h)$, where $h \geq 0$, and for each $1 \leq j \leq h$, $a_j \in A$ and t_j indicates a compartment type from T).
- *structure changing* rules: membrane division, membrane dissolution, link creation and link destruction rules, which all may also incorporate complex guards and that are covered in detail in [2]. However, this type of rules will not be considered in the following discussion.

For a multiset w over A and an element $a \in A$, we denote by $|w|_a$ the number of objects a occurring in w. Let us denote $Rel = \{<, \leq, =, \neq, \geq, >\}$, the set of relational operators, $\gamma \in Rel$, a relational operator, and a^n a multiset, consisting of n copies of a. We first introduce an *abstract relational expression.*

Definition 3. *If g is the* abstract relational expression *denoting γa^n and w a multiset, then the guard g applied to w denotes the* relational expression $|w|_a \gamma n$.

The abstract relational expression g is true for the multiset w, if $|w|_a \gamma n$ is true.

We consider now the following Boolean operators $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction). An *abstract Boolean expression* is defined by one of the following conditions:

- any abstract relational expression is an abstract Boolean expression;
- if g and h are abstract Boolean expressions then $\neg g$, $g \wedge h$ and $g \vee h$ are abstract Boolean expressions.

The concept of a guard, introduced for kP systems, is a generalisation of the promoter and inhibitor concepts utilised by some variants of P systems.

Definition 4. *If g is an* abstract Boolean expression *containing g_i, $1 \leq i \leq q$, abstract relational expressions and w a multiset, then g applied to w means the* Boolean expression *obtained from g by applying g_i to w for any $i, 1 \leq i \leq q$.*

As in the case of an abstract relational expression, the guard g is true with respect to the multiset w, if the abstract Boolean expression g applied to w is true.

For example, if g is the guard defined by the abstract Boolean expression $\geq a^4 \wedge < b^2 \vee \neg > c$ and w a multiset, then g applied to w is true if it has at least 4 a's and less than 2 b's or no more than one c.

2.2 Evolutionary Functional Testing

Software testing is the process of finding errors in a system, also measuring the quality of the system. The correctness of a system is the most essential purpose

of testing. Automated testing can be divided into white-box testing and black-box testing. White-box testing (structural testing) uses the source code of the system to generate test cases, while black-box testing (functional testing) uses the systems specifications for test generation. In white box testing the tester needs to have a look inside the source code and find out which unit of code is behaving inappropriately. In black box testing, a tester uses the system architecture or specification and does not have access to the source code [10].

One of the common approaches of automated testing is model based test cases generation. The generated test cases (based on the model) reveal faults and verify if the implementation conforms to its specification. Transforming this problem into an optimisation problem, we can use evolutionary approaches.

Search based software testing represents automated search in a potentially large input space, guided by a problem specific fitness function. The search space depends on the problem and on the configuration of the parameters of the system. The fitness function guides the search to the test goal and scores different inputs of the system according to the test goal.

Test cases generation has been intensively studied for EFSMs.

Test cases are set of input values and expected results developed to cover certain test conditions. A test suite is a collection of test cases.

2.3 Genetic Algorithms

Genetic Algorithms (GAs) are metaheuristic search techniques (mainly applied in optimization problems) that simulate the biological evolution and have the following elements: populations of chromosomes (individuals, candidate solutions), selection according to a fitness function, crossover to produce new offspring and random mutation of new offspring [14].

GAs start with initialization of a population with random candidate solutions, evolve the population several times, until a solution is found or a stop condition is met. Each element (chromosome) from the population represents a sequence of variables/parameters. Variable values can be represented in binary form, real-numbers, or even characters.

At each evolution, individuals are evaluated and selected for the next generation. The quality of each individual is determined by a fitness function that depends upon the problem considered. If the chromosome is fitter, it is likely to be selected to reproduce more times [14]. The optimization problem can be to minimize or to maximize the fitness function.

Crossover is applied to the randomly selected parent chromosomes, exchanging information between them and creating new children chromosomes. Some common types of crossovers are: single-point crossover, multi-point crossover and uniform crossover.

Mutation is applied, with some probability, to each chromosome, randomly changing some of the individual's genes. A new evolution starts with these new individuals. Mutation prevents genetic pool from premature convergence (getting

stuck in local maxima/minima). The main purpose of mutation is to bring diversity in population.

As described in [14], a simple GA works as follows:

1. Start with a randomly generated population (the initial population).
2. Compute fitness function for each chromosome in the population.
3. Repeat the following steps until a new generation is created:
   a) Select a pair of parent individuals from the current population (a chromosome can be selected only once to become a parent);
   b) Apply crossover on the current pair to form two offsprings.
   c) Mutate the two offspring chromosomes, with a given probability, and place the resulting individuals in the new population.
4. Selection is applied on the current population along with the new population.
5. Go to step 2.

A *generation* is represented by an iteration of this process. The entire set of generations is called a *run*. Two different runs will produce different behaviors. In order to evaluate the efficacy of a genetic algorithm, we should run it multiple times and analyse the results.

2.4 Search based Testing for EFSM Models

An *extended finite state machine* (EFSM) is a six-tuple (S, s_0, V, I, O, T) [9] where S is the finite set of states, s_0 is the initial state, V is the finite set of context variables, I is the finite set of inputs, O is the finite set of outputs and T is the finite set of transitions. A transition is represented by a start state, an input that may have associated input parameters, a guard (logical expression), a sequential operation (a method with assignments and output statements) and the end state.

A *path* of an EFSM is a sequence of adjacent transitions, $p = S_1 \xrightarrow{f_1[g_1]} S_2 \xrightarrow{f_2[g_2]} \dots S_m \xrightarrow{f_m[g_m]} S_{m+1}$, where S_i represents the state i from that path, f_i is the method executed on the transition i and g_i is the guard of the transition i. A path can be feasible, if there exist values for the input parameters to satisfy guards and to trigger all transitions for that path, and infeasible, otherwise.

There are many approaches that generate values for input parameters for each method f_i from a given path, that satisfy the guard conditions g_i and trigger all transitions. Lefticaru and Ipate investigated in [13] the use of search based techniques for functional testing using state machines. Its purpose is to generate input data for chosen path in a state machine, so that it triggers the transitions, using three search techniques: simulated annealing, genetic algorithms and particle swarm optimization. Kalaji et al. [9] proposed an integrated search based test data generation using EFSMs. The approach has two phases. In the first phase, feasible paths are generated using a GA with a feasibility metric based on dataflow dependence as fitness function, satisfying transition coverage criteria. In the second phase those paths are used as inputs to generate test data that trigger the paths,

using a GA with a fitness function based on the branch distance function and approach level.

The approach proposed by Lefticaru and Ipate [12] is based on the state diagram and uses a genetic algorithm to generate test data. The first step is to find feasible paths to achieve some coverage criteria. The second step is to find, for each path, the input values for parameters, to trigger the transitions. The test data generation problem is converted to an optimization problem, aiming to minimize the fitness function.

Paper [18] generates test data for EFSMs and uses a hybrid genetic algorithm, improving the algorithm presented in [12].

For a particular path in the EFSM, a chromosome (individual, possible solution) is a list of values, $x = (x_1, x_2, \ldots, x_n)$, corresponding to all parameters of the methods, as they appear on that path. A solution is a chromosome with fitness function 0 that triggers transitions between states according to the selected path and validates the guards of each transition.

The fitness function used in this approach is: $fitness = approach\_level + normalized\_branch\_level$ ($f = al + nbl$). $approach\_level$ is calculated by $m-1-p$, where m is the length of the path to be executed and p is the number of nodes executed until the first unsatisfied guard on the path. $normalized\_branch\_level$ is the mapping onto $[0, 1)$ for $branch\_level$. $branch\_level$ computes, for the predicate that is not satisfied, how close the predicate was to being true, using the transformations from Table 1. The normalization function is $norm : [0, 101] \rightarrow [0, 1], norm(d) = 1 - 1.05^{-d}$.

Element	Objective function value obj
Boolean	if TRUE then 0 else K
$a = b$	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 $(b - a) + K$
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
$a\ xor\ b$	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$
$\neg a$	Negation is moved inwards and propagated over a

Table 1. Tracey's objective functions for relational predicates and logical connectives. The value K, $K > 0$, refers to a constant which is always added if the term is not true

The algorithm ends when the stop criteria is reached or when the maximum number of evolutions is exceeded. After the selection step, a new generation is created using recombination, crossover and mutation.

3 Search based Testing for kP systems

Modelling systems specification can be also done by using kP systems. In this paper we will introduce kP systems that behave similar to EFSMs and apply testing approaches based on EFSMs.

We will consider deterministic kP systems with two compartments.

The main compartment will only consist of rewriting rules of the form $Aa \rightarrow Bb\{g\}$, where A, B and a, b belong to two disjoint sets (A, B play the role of the input and output states of a transition of an EFSM, respectively, whereas a, b are input and output of the transition, with g being its guard).At any moment only one rule is applied, i.e. the system is always working in sequential manner.

The other compartment is meant to send symbols to the main compartment that are inputs a for its rules. This compartment is behaving similar to an environment of an EFSM that provides inputs to it. The rules have the form $C \rightarrow D(a, M)$, where C, D and a belong to disjoint sets, as in the case of the main compartment; M is the type of that compartment.

An execution of the system from a given configuration consists in applying a rule from the input compartment, sending the input multiset to the main compartment and applying a corresponding rule from the main compartment.

The testing strategy developed in this paper will refer to the main compartment; the other one will just provide inputs, as presented above.

When we aim to simulate the execution of the main compartment for N steps then the compartment generating inputs should be able to generate N inputs. Execution of each rule $Aa \rightarrow Bb\{g\}$ in the main compartment depends of the availability of Aa and also on guard g that must be true. In order to generate suitable inputs both these conditions have to be fulfilled and this is what happens for a test generation.

We can automatically generate test data for kPSystems using genetic algorithms and the kPWorkbench tool to simulate the evolution of the system.

3.1 kP System Definition

In this subsection we will present the kP system configuration.

The kP system will be denoted $k\pi = (A, \mu, C_1, C_2, i_0)$. The kP system will have the following elements:

$\mu = (V, E)$, where $V = \{cM, cInp\}$, cM = main compartment and cInp = the input compartment, $E = \{(cM, cInp)\}$.

We denote A_{ST} and A_{IO} two disjoint sets and $A = A_{ST} \cup A_{IO}$. A_{ST} denotes symbols that are either corresponding to states in cM or symbols used in $cInp$ for generating inputs. A_{IO} are input and output symbols as well as symbols that appear in guards.

The membrane structure contains two compartments, $cM = (t_M, w_{M0})$, where $t_M = (R_M, \sigma_M)$ is the compartment type and w_{M0} is the initial multiset over A and $cInp = (t_I, w_{I_0})$, where $t_I = (R_I, \sigma_I)$ is the compartment type and w_{I0} is the initial multiset over A.

The main compartment type $t_M = (R_M, \sigma_M)$ consists of a set of evolution rules R_M and an execution strategy σ_M working in sequential manner. R_M contains only rewriting rules:$Aa \to Bb\{g\}$, where $A, B \in A_{ST}$ and $a, b \in A_{IO}$.

In an evolution step and a given configuration, only one rule can be applied.

The input compartment type $t_I = (R_I, \sigma_I)$ consists of a set of evolution rules R_I and the execution strategy σ_I working in sequential manner. R_I contains only rewriting and communication rules $C \to D(a, t_M)$, where $C, D \in A_{ST}$ and $a \in A_{IO}$

The initial configuration contains the initial values from the memory and the output compartment i_0 is represented by the main compartment.

Example 1. Let us consider the kP system $k\Pi_1 = (A, \mu, cM, cInp, i_0)$, where $i_0 = cM$, $A_{ST} = \{A, B, C, D, E, F, A_1, ..., A_6\}$, $A_{IO} = \{a, b.f, d, o, t, x\}$

$$R_M = \begin{cases} r_1 : A, f \to A, a\{< 3a \& = f\} & r_2 : A, f \to E\{= 3a\} \\ r_3 : A, t \to B, a\{< 3a \& = t \& < f\} & r_4 : B, x \to C\{= x\} \\ r_5 : B, d \to D\{= d\} & r_6 : C, b, x \to C\{>= x\} \\ r_7 : D, d \to D, b\{>= d\} & r_8 : C, o \to F\{< x\} \\ r_9 : D, o \to F\{< d\} \end{cases}$$

$$R_I = \begin{cases} r_{10} : A_1 \to A_2, (f, t_M) & r_{11} : A2 \to A3, (f, t_M) \\ r_{12} : A3 \to A4, (t, t_M) & r_{13} : A4 \to A5, (x, t_M) \\ r_{14} : A5 \to A6, (3x, t_M) & r_{15} : A_6 \to A_7, (o, t_M) \end{cases}$$

The inital configuration of $k\pi_1$ is $M_0 = (100b\, A, A_1)$.

The only applicable rule is r_{10} for $cInp$ and $A_1 \Longrightarrow^{r_{10}} A_2, (f, M)$ and f goes to cM. Hence, the next configuration is $M_1 = (100b\, A\, f, A_2)$.

In this configuration we can only apply rule r_1 in cM and rule r_{11} in $cInp$, $A_2 \Longrightarrow^{r_{11}} A_3, (f, M)$ in $cInp$, f goes to cM and $A \Longrightarrow^{r_1} A, a$ in cM and the next configuration is $M_2 = (100b\, A\, a\, f, A_3)$.

The next computational step is $A_3 \Longrightarrow^{r_{12}} A_4, (t, M)$ in $cInp$, t goes to cM and $A, f \Longrightarrow^{r_1} A, a$ in cM and the next configuration is $M_2 = (100b\, A\, 2a\, t, A_4)$.

The next configuration is $M_3 = (100b\ B\ 3a\ x, A_5)$ obtained by applying $A_4 \Longrightarrow^{r_{13}} A_5, (x, M)$ in $cInp$, sending x to cM and applying $A, t \Longrightarrow^{r_3} B, a$ in cM.

After this step, the only applicable rules are $A_5 \Longrightarrow^{r_{14}} A_6, (3x, M)$ in $cInp$ and $B, x \Longrightarrow^{r_4} C$ in cM, sending $3x$ to cM and the next configuration is $M_4 = (100b\, C\, 3a\, 3x, A_6)$.

From this configuration we can apply $A_6 \Longrightarrow^{r_{15}} A_7, (o, M)$ in $cInp$ and $C, b, x \Longrightarrow^{r_6} C$ in cM, sending o to cM and reaching configuration $M_5 = (99b\, C\, 3a\, 2x\, o, A_7)$.

The next computational step contains the rule $C, b, x \Longrightarrow^{r_6} C$ applied in cM, obtaining the configuration $M_6 = (98b\, C\, 3a\, x\, o, A_7)$. In the next step, the same rule is applied identically in cM and $M_7 = (97b\, C\, 3a\, o, A_7)$.

In the last computational step, the applicable rule is $C, o \Longrightarrow^{r_8} F$ in cM and the final configuration is $M_8 = (97b\, F\, 3a, A_7)$.

The evolution steps obtained by this simulation are the following:

- Step 1: rule r_{10}
- Step 2: rules $r_{11}r_1$
- Step 3: rules $r_{12}r_1$
- Step 4: rules $r_{13}r_3$
- Step 5: rules $r_{14}r_4$
- Step 6: rules $r_{15}r_6$
- Step 7: rule r_6
- Step 8: rule r_6
- Step 9: rule r_8

To simulate the execution of a system we use kPWorkbench. kPWorkbench is an integrated software suite aimed to provide support for kP systems. Among other functionallities, kPWorkbench contains tools for modelling, simulating and verifying kP systems. A simulation trace represents the evolution of the system during some computations.

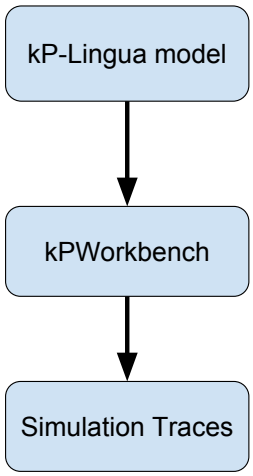

Fig. 1. kPWorkbench simulation steps

3.2 Genetic Algorithm Configuration

The Genetic Algorithm has the following steps:

- create random initial population - length N;

- evaluate the population using the fitness function;
- repeat the following steps until the stopping condition is reached:
  - offspring population ← reproduction(population);
  - evaluate offsprig popultion;
  - population ← reinsertion+selection(population, offspring population).

A chromosome $(x_1, x_2, \ldots, x_n)$ is represented as a list of input symbols corresponding to the input set. A gene represents the input for the corresponding step and consists of a list of strings (input symbols) $x_i = (r_1 s_1, r_2 s_2, \ldots, r_n s_n)$, where s_i is a symbol from the alphabet, and r_i is the number of times the symbol appears in the input set.

The *Crossover Operator* creates two new chromosomes from the two existing parent chromosomes, using one of the two operations, with equal probability.

- exchange only the value for a gene from a random selected point

$$\begin{cases}(x_1, x_2, \ldots, x_n)\\(y_1, y_2, \ldots, y_n)\end{cases} \rightarrow \begin{cases}(x_1, x_2, \ldots, y_i, \ldots, x_n)\\(y_1, y_2, \ldots, x_i, \ldots, y_n)\end{cases}$$

- exchange for a random point only a part of the gene

$$\begin{cases}(x_1, x_2, \ldots, x_i, \ldots, x_n)\\(y_1, y_2, \ldots, y_i, \ldots, y_n)\end{cases} \rightarrow \begin{cases}(x_1, x_2, \ldots, x'_i, \ldots, x_n)\\(y_1, y_2, \ldots, y'_i, \ldots, y_n)\end{cases}$$

  where
  $x_i = (r_1 s_1, \ldots, r_n s_n), y_i = (t_1 s_1, \ldots, t_n s_n)$,
  $x'_i = (r_1 s_1, \ldots, t_i s_i, \ldots, t_n s_n), y'_i = (t_1 s_1, \ldots, r_i s_i, \ldots, r_n s_n)$

A chromosome can be mutated in many different ways. To identify possible mutation operators, we considered the characteristics of a chromosome.We have defined the following different mutation operators, which are all applied with 0.5 probability:

- completly change a gene (an input value)
- remove gene part - for an input value choose randomly a symbol that will not be used ($r_i = 0$)
- for a random gene - replace random symbol number(r_i)
- exchange materials between two genes

We tried to apply the selection operator as it was used in many test data generation approaches. The reinsertion of the offspring population into the new population was made in different ways:

- the new population = the offspring population [12];
- the new population = the offspring population and the fittest individual is kept in the next generation [18];

- apply selection operator and select N chromosomes from the offspring population along with the old population, using different selectors: best chromosome selection, binary tournment selection.

None of these methods worked for our problem. The algorithm was stuck in a local optima. In order to overcome this problem we change the reinsertion method: the best 50% of the current generation and the best 50% of the new offspring are retained. In the next evolution, the crossover operator will use a parent that came from the old population and a parent that came from the offspring population and will create a new individual. This reinsertion method was inspired and adapted from paper [5].

$P = \{x_1, y_1, x_2, y_2, \ldots, x_{25}, y_{25}\}$, where $x_1, x_2, \ldots, x_{25} \in Old_{Population}$ and $y_1, y_2, \ldots, y_{25} \in Offspring_{population}$

To evaluate an individual we need to compute the objective function. To verify if a chromosome is the solution, we need to simulate the system with the corresponding input values and compare the simulation traces with the given steps. The fitness function is based on the *approach level* and the *branch distance*. It checkes if the input steps are exactly the needed ones (representing a solution) or how far is the chromosome from the solution. The approach level records how many steps were not executed by a particular input. The fewer steps executed, the *further away* the input is from executing the steps. The branch distance is computed using the conditions of the guards of the rule at which the evolution diverted away from the current target step.

To compute the fitness function we need to perform a simulation of the system. To simulate the system we use kPWorkbench.

3.3 Experiments

In our experiments we used the kP system presented in Example 1.

Experiment 1

consisted in generating test data for the following evolution steps:

$Steps = \{r_{10}, (r_1, r_{11}), (r_3, r_{12}), (r_4, r_{13}), (r_7, r14), r_7, r_7, r_7, r_7, t_8\}$

The size of the input set is $size = 5$. In this example we have 10 steps, but only the first 6 will receive inputs. The other steps will consume the inputs until the system reaches the final configuration.

The maximum number of evolution is set to 50.

This experiment was made to find the suitable configuration for the algorithm, including the reinsertion method. As described in Subsection 3.2, the first experiments failed. Running 100 times the algorithm for each configuration, we couldn't find a solution. Only for the new reinsertion method the algorithm was successful with a success rate of 75%. Also, the average number of evolutions needed to find a solution was 37.

There are many input sets that create the given scenario. Table 3.3 contains some examples of solutions obtained using during this experiment. The first column shows the number of generations needed to find the solution.

Evolutions	Input1	Input 2	Input 3	Input 4	Input 5
45	$1f\ 2x$	$1t$	$1d$	$1f\ 3x\ 3d$	$2x\ 2d\ 1o$
41	$1f\ 2x$	$1t\ 1d$		$2x\ 2d$	$1f\ 6x\ 3d\ 1o$
33	$1f$	$1f\ 1t\ 5x$	$1d\ 1o$	$5x$	$4d$
32	$1f\ 1t$	$1d$	$1x$	$1f\ 4x\ 5d\ 1o$	$1x$
28	$1f\ 1t\ 1d$	$1o$	$2x$	$3x\ 1d$	$1f\ 1x\ 4d$
27	$1f\ 1t$		$1d\ 1o$	$1t\ 2d$	$1f\ 3d$
43	$1f\ 1t\ 1x$	$3x$	$1d$	$2x\ 1d\ 1o$	$1f\ 4x\ 4d$

Table 2. Example of solutions for Experiment 1

Experiment 2

consisted in generating test data for other examples of evolution steps, using the configuration establish during *Experiment 1*. One of the following evolution steps set we used was:

$Steps = \{r_{10}, (r_1, r_{11}), (r_3, r_{12}), (r_4, r_{13}), (r_6, r14), r_6, r_6, r_6, r_6, t_8\}$

The size of the input set is $size = 5$. In this example we have 10 steps, but only the first 6 will receive inputs. The other steps will consume the inputs until the system reaches the final configuration. The maximum number of evolution is set to 50. Running 100 times the algorithm, we couldn't find a solution. The success rate for this example was 62% and the average number of evolutions needed to find a solution was 36.

4 Conclusions

In conclusion, we used genetic algorithms to generate test data for kP systems. The algorithm input is reperesented by a kP system model and a set of computation steps, the ouput being the set of input sets needed to create the given input scenario. The algorithm uses kP systems that behave similar to EFSMs. We tried to apply directly some algorithm defined for EFSMs, but it wasn't successful. We overcomed this problem by changing the reinsertion method of the population. With this configuration, the algorithm was successful.

As future work, we will extend this algorithm to other kP systems, starting from using different kinds of rules also.

Acknowledgements

This work is supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-III-P4-ID-PCE-2016-0210.

References

1. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierla, L.: Model checking kernel p systems. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing. Lecture Notes in Computer Science, vol. 8340, pp. 151–172. Springer Berlin Heidelberg (2014)
2. Gheorghe, M., Ipate, F., Dragomir, C., Mierla, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P Systems - Version I. Eleventh Brainstorming Week on Membrane Computing (11BWMC) pp. 97–124 (2013)
3. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M.J., Turcanu, A., Valencia-Cabrera, L., García-Quismondo, M., Mierla, L.: 3-col problem modelling using simple kernel P systems. International Journal of Computer Mathematics **90**(4), 816–830 (2013). https://doi.org/10.1080/00207160.2012.743712, https://doi.org/10.1080/00207160.2012.743712
4. Gheorghe, M., Ipate, F., Lefticaru, R., Turlea, A.: Testing identifiable kernel p systems using an x-machine approach. In: International Conference on Membrane Computing. pp. 142–159. Springer (2018)
5. Harman, M., McMinn, P.: A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: Proceedings of the 2007 international symposium on Software testing and analysis. pp. 73–83. ACM (2007)
6. Ipate, F., Gheorghe, M.: Finite state based testing of P systems. Natural Computing **8**(4), 833 (2009). https://doi.org/10.1007/s11047-008-9099-3, https://doi.org/10.1007/s11047-008-9099-3
7. Ipate, F., Gheorghe, M.: Testing non-deterministic stream X-machine models and P systems. Electronic Notes in Theoretical Computer Science **227**, 113–126 (2009). https://doi.org/10.1016/j.entcs.2008.12.107, https://doi.org/10.1016/j.entcs.2008.12.107
8. Ipate, F., Gheorghe, M., Lefticaru, R.: Test generation from P systems using model checking. Journal of Logic and Algebraic Programming **79**(6), 350–362 (2010). https://doi.org/10.1016/j.jlap.2010.03.007, https://doi.org/10.1016/j.jlap.2010.03.007
9. Kalaji, A.S., Hierons, R.M., Swift, S.: An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. Information & Software Technology **53**(12), 1297–1318 (2011)
10. Khan, M.E., Khan, F.: A comparative study of white box, black box and grey box testing techniques. International Journal of Advanced Computer Sciences and Applications **3**(6), 12–1 (2012)
11. Lefticaru, R., Gheorghe, M., Ipate, F.: An empirical evaluation of P system testing techniques. Natural Computing **10**(1), 151–165 (2011). https://doi.org/10.1007/s11047-010-9188-y, https://doi.org/10.1007/s11047-010-9188-y

12. Lefticaru, R., Ipate, F.: Automatic state-based test generation using genetic algorithms. In: Proc. SYNASC'07. pp. 188–195. IEEE Computer Society (2007)
13. Lefticaru, R., Ipate, F.: Functional search-based testing from state machines. In: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008. pp. 525–528 (2008)
14. Mitchell, M.: An introduction to genetic algorithms. MIT Press (1998)
15. Păun, G.: Computing with membranes. Tech. rep., Turku Centre for Computer Science (1998)
16. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693, https://doi.org/10.1006/jcss.1999.1693
17. The P systems website. http://ppage.psystems.eu, [Online; accessed 12/05/2018]
18. Turlea, A., Ipate, F., Lefticaru, R.: A hybrid test generation approach based on extended finite state machines. In: 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, September 24-27, 2016. pp. 173–180 (2016). https://doi.org/10.1109/SYNASC.2016.037, https://doi.org/10.1109/SYNASC.2016.037