Mutation Based Testing of P Systems

Florentin Ipate[1], Marian Gheorghe[2]

[1] Department of Computer Science
Faculty of Mathematics and Computer Science
The University of Pitesti
Str Targu din Vale 1, 110040 Pitesti
`florentin.ipate@ifsoft.ro`

[2] Department of Computer Science, The University of Sheffield
Regent Court, Portobello Street, Sheffield S1 4DP, UK
`M.Gheorghe@dcs.shef.ac.uk`

Summary. Although testing is an essential part of software development, until recently, P system testing has been completely neglected. Mutation testing (mutation analysis) is a structural software testing method which involves modifying the program in small ways. Mutation analysis has been largely used in white-box testing, but only a few tentative attempts to use this idea in black-box testing have been reported in the literature. In this paper, we provide a formal way of generating mutants for systems specified by context-free grammars. Furthermore, the paper shows how the proposed method can be used to construct mutants for a P system specification, thus making a significant progress in the area of P system testing.

1 Introduction

Membrane computing, the research field initiated by Gheorghe Păun in 1998 [12], aims to define computational models, called P systems, which are inspired by the behavior and structure of the living cell. Since its introduction in 1998, the P system model has been intensively studied and developed: many variants of membrane systems have been proposed, a research monograph [13] has been published and regular collective volumes are annually edited – a comprehensive bibliography of P systems can be found at [16]. The most investigated membrane computing topics are related to the computational power of different variants, their capabilities to solve hard problems, like NP-complete ones, decidability, complexity aspects and hierarchies of classes of languages produced by these devices. In the last years there have also been significant developments in using the P systems paradigm to model, simulate and formally verify various systems [2]. Suitable classes of P systems have been associated with some of these applications and software packages have been developed. Of the many variants of P systems that have been defined, in this paper we consider cell-like P systems without priority rules and membrane dissolving rules [13].

Testing is an essential part of software development and all software applications, irrespective of their use and purpose, are tested before being released. Testing is not a replacement for a formal verification procedure, when the former is also present, but rather a complementary mechanism to increase the confidence in software correctness [5]. Although formal verification has been applied to different models based on P systems [1], until recently testing has been completely neglected in this context.

The main testing strategies involve either (1) knowing the specific function or behavior a product is meant to deliver (functional or black-box testing) or (2) knowing the internal structure of the product (structural or white-box testing). In black-box testing, the test generation may be based on a formal specification or model, in which case the process could be automated. There is a large class of formal models used in software specification: finite state machines, Petri nets, process algebras, Z, VDM etc. We can add now P systems as a formal approach [2] to specifying various applications in linguistics, graphics and, more recently, biology, especially for defining signalling pathways.

A number of recent papers devise testing strategies based on rule coverage [4], finite state machine [8] and stream X-machine [7] conformance techniques. In this paper, we propose an approach to P system testing based on mutation analysis.

Mutation testing (mutation analysis) is a structural software testing method which involves modifying the program in small ways [14], [9]. The modified versions of the program are called *mutants*.

Consider, for example, the following fragment of a Java program:

```
if $(x \geq 0) \&\& a$ then
  $y = y + 1$
else
  $y = y + 2$
```

Then mutants for this code fragment can be obtained by either

- substituting && with another logic operator, e.g., ||;
- substituting $\geq$ with another comparison operator, e.g., $>$, $=$;
- substituting $+$ with another arithmetic operators, e.g., $-$;
- substituting one variable (e.g. x) with another one, e.g., y (we assume that the two variables have the same type).

Some (not all) mutants of the above code fragment are given below.

```
if $(x \geq 0) || a$ then
  $y = y + 1$
else
  $y = y + 2$
```

```
if $(x > 0)\&\&a$ then
  $y = y + 1$
else
  $y = y + 2$

if $(x \geq 0)\&\&a$ then
  $y = y - 1$
else
  $y = y + 2$

if $(x \geq 0)\&\&a$ then
  $y = y + 1$
else
  $y = y - 2$

if $(x \geq 0)\&\&a$ then
  $x = y + 1$
else
  $y = y + 2$

if $(x \geq 0)\&\&a$ then
  $y = y + 1$
else
  $x = y + 2$
```

A variety of *mutation operators* (ways of introducing errors into the correct code) for imperative languages are defined in the literature [9], [10] (a few examples are given above). These are called traditional mutation operators. Beside these, there are mutation operators for specialized programming environments, such as object-oriented languages [10]. A popular tool for generating mutants for Java programs is MuJava [15], [10].

The underlying idea behind mutation testing is that, in practice, an erroneous program either differs only in a small way from the correct program or, alternatively, a bigger fault can be expressed as the summation of smaller (basic) faults and so, in order to detect the fault, the appropriate mutants need to be generated. If the test suite is able to detect the fault (i.e., one of the tests fails), then the mutant is said to be killed. Two kinds of mutation have been defined in the literature: *weak mutation* requires the test input to cause different program states for the mutant and the original program; *strong mutation* requires the same condition but also the erroneous state to be propagated at the end of the program (and hence produce an incorrect output). Obviously, the generation of a weak mutation test suite is less complex; on the other hand, a strong mutation test suite is easier to apply as only the program outputs need to be measured.

Mutation analysis has been largely used in white-box testing, but only a few tentative attempts to use this idea in black-box testing have been reported in the literature [11]. Offutt et al. propose a general strategy for developing mutation operators for a grammar based software artefact, but the ideas that outline the proposed strategy for mutation operator development are rather vague and general and no formalization is provided.

In this paper we provide a formal way of generating mutants for systems specified by context-free grammars. Given such a specification, a derivation (or parse) tree can be associated with. Based on it, we formally describe the process of generating the mutants for the given tree. Furthermore, the paper shows how the proposed method can be used to construct mutants for a P system specification.

2 Preliminaries

Before proceeding we introduce the notation to be used in this paper. For an alphabet $V = \{a_1, \ldots, a_p\}$, V^* denotes the set of all strings over V. λ denotes the empty string. For a string $u \in V^*$, $|u|_{a_i}$ denotes the number of a_i occurrences in u. Each string u has an associated vector of non-negative integers $(|u|_{a_1}, \ldots, |u|_{a_p})$. This is denoted by $\Psi_V(u)$.

2.1 Context-free grammars

In this section basic concepts and results related to context-free grammars are introduced. For more details on automata, grammars and languages we refer to a classical textbook [6]. A context-free grammar is 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 $A \rightarrow w$; where A is a single nonterminal symbol, and w is a string of terminals and/or nonterminals;
- S is the start symbol.

For any strings $u, v \in (V \cup T)^*$, we write $u \Longrightarrow v$ if there exists a production rule $A \rightarrow w$ and $\alpha, \beta \in (V \cup T)^*$ such that $u = \alpha A \beta$ and $v = \alpha w \beta$. That is, v is the result of applying the rule $A \rightarrow w$ to u.

The $\Longrightarrow$ relation can be extended to a sequence of zero or more production rules: for any $u, v \in (V \cup T)^*$, we write $u \Longrightarrow^* v$ if there exist $u_1, \cdots, u_k$, $k \geq 1$ such that $u = u_1$, $v = u_k$ and $u_i \Longrightarrow u_{i+1}$, $1 \leq i \leq k-1$. We say that u *derives* v. If the derivation has at least one step (i.e., $k > 1$) then we denote $u \Longrightarrow^+ v$.

The *language* described by the context-free grammar G is the set $L(G) = \{v \in T^* \mid S \Longrightarrow^* v\}$. A language $L \subseteq T^*$ is said to be context-free if there is a context-free grammar G such that $L = L(G)$.

A context-free grammar is said to be proper if

- it has no useless symbols (inaccessible symbols or unproductive symbols), i.e., $\forall A \in V$, $\exists \alpha, \beta \in (V \cup T)^*, v \in T^*$ such that $S \Longrightarrow^* \alpha A \beta$ and $A \Longrightarrow^* v$;
- it has no λ-productions, i.e., $A \rightarrow \lambda$;
- it has no renaming production rules, i.e., $A \rightarrow B$, for $A, B \in V$.

For every context-free language L, if $\lambda \notin L$ then there exists a proper context-free grammar that describes L. For simplicity, in the sequel we consider only proper context-free grammars.

A derivation (parse) tree for a (proper) context-free grammar $G = (V, T, P, S)$ is a tree that satisfies the following conditions:

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

If the root node is labeled by S then the yield of the tree is the string of terminals obtained by concatenating the leaves from left to right. For any string of terminals $w \in T^*$, $S \Longrightarrow^* w$ if and only if w is the yield of some derivation (parse) tree with root S [6]. Consequently, $w \in L(G)$ if and only if w is generated by some parse tree with root S. Parse trees have very high practical value as they are used by compilers to represent the structure of the source code.

A grammar is said to be *ambiguous* if there exists a string and in any leftmost derivation (always the leftmost nonterminal is rewritten) this can be generated by more than one derivation (parse) tree. Usually, ambiguity is a feature of the grammar, not of the language and unambiguous grammars can be found to describe the same context-free language. However, there are certain context-free languages which can only be generated by ambiguous grammars; such languages are called *inherently ambiguous.* An ambiguous grammar presents a practical problem since a string may be associated with more than one parse tree. However, there are well-known techniques for eliminating the causes of ambiguity which are used in compiler construction.

In the sequel we will consider (possibly ambiguous) context-free grammars which describe languages that are not inherently ambiguous. We will tacitly assume that mechanisms for solving the causes of ambiguity exist and so there is a one-to-one mapping between a string and its parse tree.

2.2 P systems

A basic cell-like P system is defined as a hierarchical arrangement of membranes identifying corresponding regions of the system. With each region there are associated a finite multiset of objects and a finite set of rules; both may be empty. A multiset is either denoted by a string $u \in V^*$, where the order is not considered, or by $\Psi_V(u)$. The following definition refers to one of the many variants of P systems, namely cell-like P system, which uses non-cooperative transformation and communication rules [13]. We will call these processing rules. Since now onwards we will refer to this model as simply P system.

Definition 1. *A* P system *is a tuple $\Pi = (V, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n)$, where*

- *V is a finite set, called* alphabet*;*

- *μ defines the membrane structure; a hierarchical arrangement of n compartments called* regions *delimited by* membranes*; these membranes and regions are identified by integers 1 to n;*
- *w_i, $1 \leq i \leq n$, represents the initial multiset occurring in region i;*
- *R_i, $1 \leq i \leq n$, denotes the set of processing rules applied in region i.*

The membrane structure, μ, is denoted by a string of left, [, and right,], brackets, each with the label of the membrane it points to; μ also describes the position of each membrane in the hierarchy. For instance, a structure of three membranes in which membrane 1 contains membranes 2 and 3 can be described by either $[_1[_2]_2[_3]_3]_1$ or $[_1[_3]_3[_2]_2]_1$. The rules in each region have the form $u \rightarrow (a_1, t_1) \ldots (a_m, t_m)$, where u is a multiset of symbols from V, $a_i \in V$, $t_i \in \{in, out, here\}$, $1 \leq i \leq m$. When such a rule is applied to a multiset u in the current region, the symbol a is replaced by the symbols a_i with $t_i = here$; symbols a_i with $t_i = out$ are sent to the outer region or outside the system when the current region is the external compartment and symbols a_i with $t_i = in$ are sent into one of the regions contained in the current one, arbitrarily chosen. In the following definitions and examples all the symbols $(a_i, here)$ are used as a_i. The rules are applied in maximally parallel mode which means that they are used in all the regions in the same time and in each region all the symbols that may be processed, must be.

A configuration of the P system Π, is a tuple $c = (u_1, \ldots, u_n)$, where $u_i \in V^*$, is the multiset associated with region i, $1 \leq i \leq n$. A derivation of a configuration c_1 to c_2 using the maximal parallelism mode is denoted by $c_1 \Longrightarrow c_2$. In the set of all configurations we will distinguish terminal configurations; $c = (u_1, \ldots, u_n)$ is a terminal configuration if there is no region i such that u_i can be further derived.

For the type of P systems we investigate in this paper multi-membranes can be equivalently collapsed into one membrane through properly renaming symbols in a membrane. Thus, for the sake of convenience, in this paper we will only focus on P systems with only one membrane.

2.3 Kripke structures

Definition 2. *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, that is, for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in H$;*
- *$L : S \rightarrow 2^{AP}$ is an interpretation function, that labels each state with the set of atomic propositions true in that state.*

Usually, the Kripke structure representation of a system results by giving values to every variable in each configuration of the system. Suppose $var_1, \ldots, var_n$ are the system variables, Val_i denotes the set of values for var_i and val_i is a value from Val_i, $1 \leq i \leq n$. Then the states of the system are $S = \{(val_1, \ldots, val_n) \mid val_1 \in Val_1, \ldots, val_n \in Val_n\}$, and the set of atomic predicates are $AP = \{(val_i = val) \mid 1 \leq i \leq n, val \in Val_i\}$. Naturally, L will map each state (given by the values of variables) onto the corresponding set of atomic propositions. Additionally, a halt (sink) state is needed when H is not total and an extra atomic proposition, that indicates that the system has reached this state, is added to AP. For convenience, in the sequel AP and L will be omitted from the definition of a Kripke structure.

Definition 3. *An infinite path in a Kripke structure $M = (S, H, I, L)$ from a state $s \in S$ is an infinite sequence of states $\pi = s_0 s_1 \ldots$, such that $s_0 = s$ and $(s_i, s_{i+1}) \in H$ for every $i \geq 0$. A finite path π is a finite prefix of an infinite path.*

3 Mutation Testing from a Context-Free Grammar

In this section we provide a way of constructing mutants for systems specified by context-free grammars. Given the system specification, in the form of a parse tree, we formally describe the generation of mutants for the given specification.

Consider a context-free $G = (V, T, P, S)$ and $L(G)$ the language defined by G. We assume that, for every production rule p of G of the form $A \to X_1 \ldots X_k$, we have defined a set $Mut(p)$, called the *set of mutants* of p. A mutant p' of p is a production rule of the form $A \to X'_1 \ldots X'_n$ such that each symbol $X'_1, \ldots, X'_n$ is either a terminal or is found among $X_1, \ldots, X_k$. Furthermore, p' is either a production rule of G itself or has the form $A \to A$, $A \in V$; this condition ensures that the yield of the mutated tree is syntactically correct.

Among the mutants of p, the following types of mutants can be distinguished:

- A *terminal replacement mutant* is a production rule of the form $A \to X'_1 \ldots X'_k$ if there exists j, $1 \leq j \leq k$, such that $X_j, X'_j \in T$, $X_j \neq X'_j$ and $X'_i = X_i$, $1 \leq i \leq n$, $i \neq j$.
- A *terminal insertion mutant* is a production rule of the form $A \to w$ where w is obtained by inserting one terminal into the string $X_1 \ldots X_k$ (at any position).
- A *string deletion mutant* is a production rule of the form $A \to w$ where w is obtained by removing one or more symbols from $X_1 \ldots X_k$.
- A *string reordering mutant* is a production rule of the form $A \to w$ where w is obtained by reordering the string $X_1 \ldots X_k$.

Given any parse tree Tr for G, the set of mutants of Tr is defined as follows:

- A one-node tree has no mutants.
- Let Tr be the tree with root A and subtrees $Tr_1, \ldots, Tr_k$ having root nodes $X_1, \ldots, X_k$, respectively and $p \in P$ the corresponding production rule of G, of the form $A \to X_1 \ldots X_k$. This is denoted by $Tr = MakeTree(A, Tr_1, \ldots, Tr_k)$. Let Tr' denote a mutant of Tr. Then either

– (**subtree mutation**) $Tr' = MakeTree(A, Tr'_1, \ldots, Tr'_k)$, where there exists j, $1 \leq j \leq k$, such that Tr'_j is mutant of Tr_j and $Tr'_i = Tr_i$, $1 \leq i \leq k$, $i \neq j$, or
– (**rule mutation**) $Tr' = tree(A, Tr'_1, \ldots, Tr'_n)$, where there exists a mutant p' of p of the form $A \to X'_1 \ldots X'_n$ such that for every i, $1 \leq i \leq n$, there exists j, $1 \leq j \leq n$, such that $Tr'_i = Tr_j$.

According to [11] these operations can be made such as to keep the result produced by them in the same language or in a larger one. In the first case a much simpler approach can be considered whereby each rule having a certain nonterminal in the left hand side is replaced by another different rule having the same nonterminal as left hand side. However the above set of operations provide a two stage method which generates mutants by considering first the rule level and then the derivation (parse) tree. If these operations are restricted to produce strings in the same language then we have the following result.

Lemma 1. *Every mutant of a parse tree for G is also a parse tree from G.*

Proof. Follows by induction on the depth of the tree.

Thus, the yield of any mutant constructed as above belongs to the language described by G and so only *syntactically correct* mutants will be generated. Syntactically incorrect mutants are useless (they do not produce test data) and so the complexity of the testing process is reduced by making sure that these are ruled out from the outset.

Example 1. Let $G = (V, T, P, S)$ where

- $V = \{S\}$;
- $T = \{0, \ldots, N\} \cup \{+, -\}$ where N is a fixed upper bound;
- $P = \{p_1, p_2\} \cup \{p_3^i \mid 0 \leq i \leq N\}$, where $p_1 : S \to S + S$, $p_2 : S \to S - S$, $p_3^i : S \to i$, $0 \leq i \leq N$.

Suppose we have the following rule mutants:

- Mutants for $p_1 : S \to S - S$ (terminal replacement), $S \to S$ (string deletion)
- Mutants for $p_2 : S \to S + S$ (terminal replacement), $S \to S$ (string deletion)
- Mutants for p_3^i : $S \to i - 1$ and $S \to i + 1$ if $1 < i < N$, $S \to 1$ if $i = 0$ and $S \to N - 1$ if $i = N$. The mutants of p_3^i are of terminal replacement type and are based on a technique widely used in software testing practice, called boundary value analysis. According to practical experience, many errors tend to lurk close to boundaries; thus, an efficient way to uncover faults is to look at the neighboring values

Consider the string $1 + 2 - 3$ and a parse tree for this string as represented in Figure 1 (leaf nodes are in bold). The construction of mutants for the given parse tree is illustrated in Figures 2, 3 and 4. Thus, the mutated strings are $0 + 2 - 3$, $2 + 2 - 3$, $1 + 1 - 3$, $1 + 3 - 3$, $1 - 3$, $2 - 3$, $1 + 2 - 2$, $1 + 2 - 4$, $1 + 2 + 3$,

$1 - 2 - 3$, $1 + 2$, 3. Some of these produce the same result as the original string; these are called *equivalent mutants*. Since no input value can distinguish these mutants from the correct string, they will not affect the test suite when strong mutation is considered.

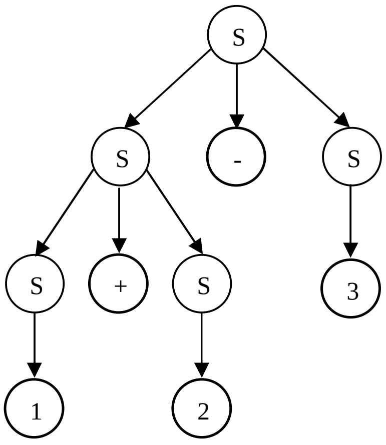

Fig. 1. Example parse tree

4 P System Mutation Testing

Consider a 1-membrane P-system $\Pi = (V, \mu, w, R)$, where $R = \{r_1, \ldots, r_m\}$; each rule r_i, $1 \leq i \leq m$, is of the form $u_i \rightarrow v_i$, where u_i and v_i are multisets over the alphabet V. In the sequel, we treat the multisets as vectors of non-negative integers, that is each multiset u is replaced by $\Psi_V(u) \in \mathbf{N}^k$, where k denotes the number of symbols in V.

In order to keep the number of configuration finite we will assume that each component of a configuration u cannot exceed an established upper bound denoted Max. We denote $u \leq Max$ if $u_i \leq Max$ for every $1 \leq i \leq k$ and $\mathbf{N}^k_{Max} = \{u \in \mathbf{N}^k \mid u \leq Max\}$. Analogously to [3], the system is assumed to crash whenever $u \leq Max$ does not hold (this is different from the normal termination, which occurs when $u \leq Max$ and no rule can be applied). Under these conditions, the 1-membrane P system Π can be described by a Kripke structure.

In order to define the Kripke structure equivalent of Π we use two predicates, $MaxParal$ and *Apply*, defined by: $MaxParal(u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$, $u \in \mathbf{N}^k_{Max}$, $n_1, \ldots, n_m \in \mathbf{N}$ signifies that a derivation of the configuration u in maximally parallel mode is obtained by applying rules $r_1 : u_1 \rightarrow v_1, \ldots, r_m : u_m \rightarrow v_m$ for $n_1, \ldots, n_m$ times, respectively; $Apply(u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$, $u \in$

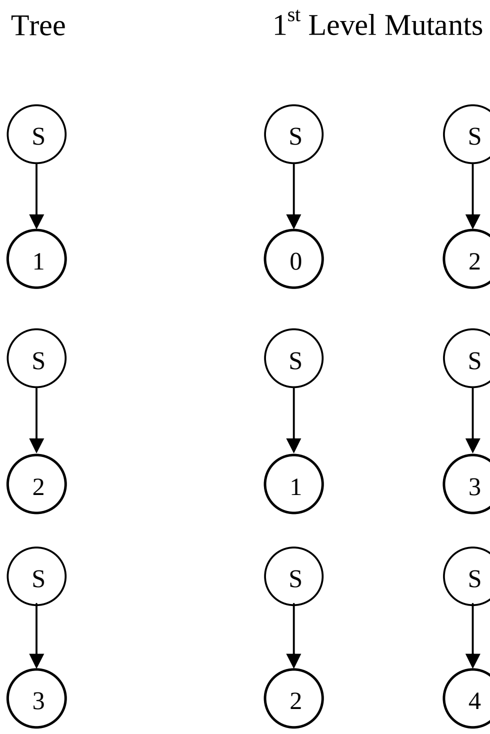

Fig. 2. 1st level mutants

$\mathbf{N}^k_{Max}$, $n_1, \ldots, n_m \in \mathbf{N}$, denotes that v is the result of applying rules $r_1, \ldots, r_m$ for $n_1, \ldots, n_m$ times, respectively.

Then the Kripke structure equivalent $M = (S, H, I, L)$ of Π is defined as follows:

- $S = N^k_{Max} \cup \{Halt, Crash\}$ with $Halt, Crash \notin \mathbf{N}^k_{Max}$, $Halt \neq Crash$;
- $I = w$;
- H is defined by:
  - $(u, v) \in H$, $u, v \in \mathbf{N}^k_{Max}$, if $\exists n_1, \ldots, n_m \in \mathbf{N} \cdot MaxParal(u, u_1, v_1, n_1, \ldots, u_m, v_m, n_m) \wedge Apply(u, v, u_1, v_1, n_1, \ldots, u_m, c_m, n_m)$;
  - $(u, Halt) \in H$, $u \in \mathbf{N}^k_{Max}$, if $\neg\exists v \in \mathbf{N}^k_{Max}, n_1, \ldots, n_m \in \mathbf{N} \cdot Apply(u, v, u_1, v_1, n_1, \ldots, u_m, v_m, n_m)$;
  - $(u, Crash) \in H$ if $\neg\exists v \in \mathbf{N}^k_{Max} \cup \{Halt\} \cdot (u, v) \in H$;
  - $(Halt, Halt) \in H$;
  - $(Crash, Crash) \in H$.

It can be observed that the relation H is total.

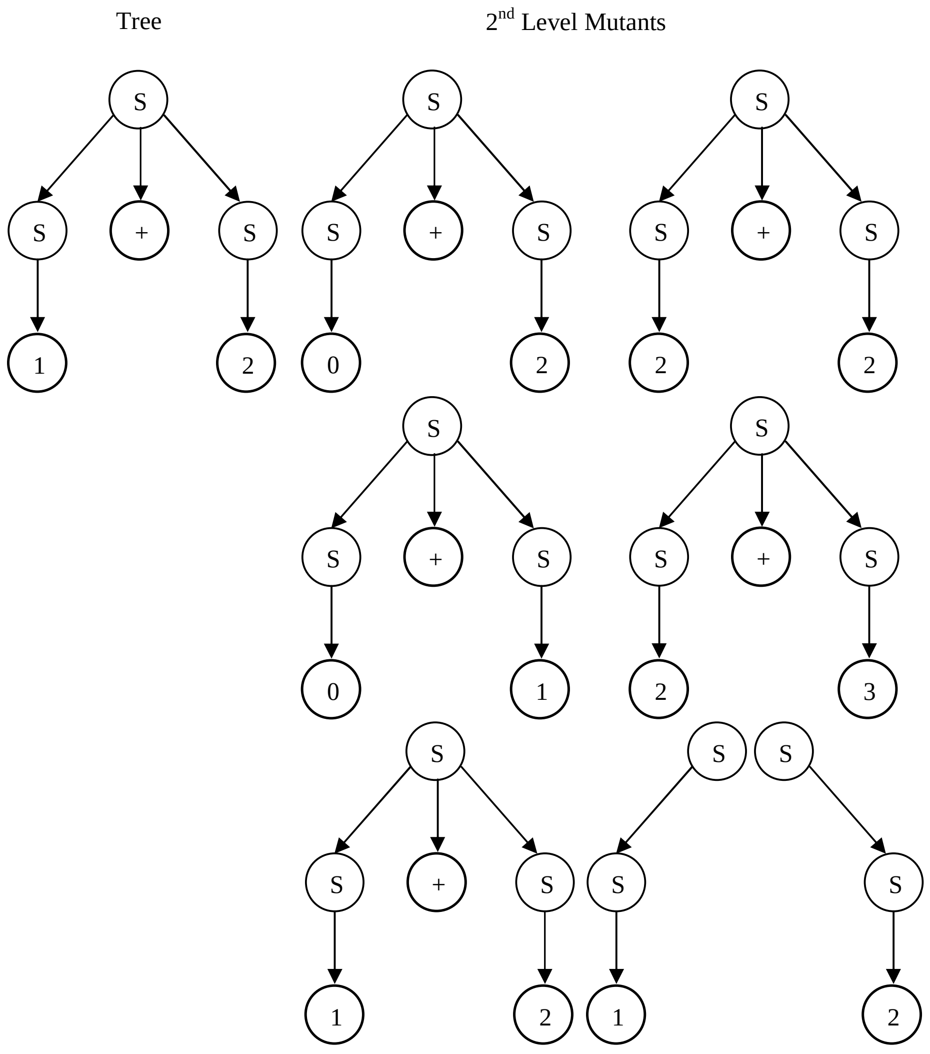

Fig. 3. 2nd level mutants

A multi-membrane P system without dissolving rules can be collapsed into a 1-membrane P system so, without loss of generality we will only consider 1-membrane P systems.

In order to use mutation analysis in P system testing we first have to describe an appropriate context-free grammar, such that the P system specification can be written as a string accepted by this grammar. The parse tree for the string is then generated and the procedure presented in the previous section is used for mutant construction.

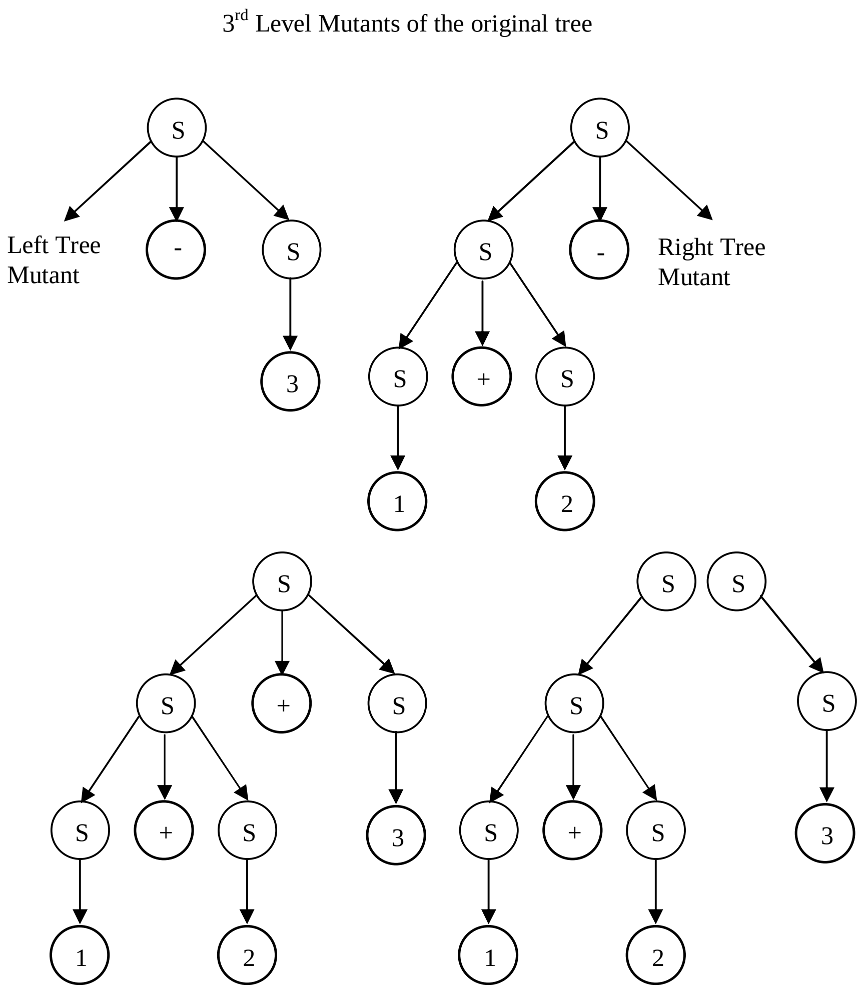

Fig. 4. 3rd level mutants

The grammar definition will depend on the level at which testing is intended to be performed. At a high level (for instance in integration testing) the predicates $MaxParal$ and *Apply* will normally be assumed to be correctly implemented and so they will be presented as terminals in the grammar; obviously, they can be themselves described by context-free grammars and appropriate mutants will be generated in a similar fashion at lower level of testing. On the other hand, it is possible to incorporate the definitions of the two predicates into the definition of the transition relation H; in this case the corresponding grammar will be much more complex and system testing will be performed in one single step. Naturally, for complexity reasons, in practice a multi-phase testing strategy is normally preferred.

The following (simplified) example illustrates the above strategy for high-level testing of P systems.

Example 2. Consider a 1-membrane P-systems with 2 rules $r_1 : u_1 \to v_1$, $r_2 : u_2 \to v_2$. Then the transition of the Kripke structure representation of Π is given by the formulae:

- $(u, v) \in H$, $u, v \in \mathbf{N}^2_{Max}$, if $\exists n_1, n_2 \in N \cdot MaxParal(u, u_1, v_1, n_1, u_2, v_2, n_2) \wedge Apply(u,\ v,\ u_1,\ v_1, n_1, u_2, c_2, n_2)$;
- $(u, Halt) \in H$, $u \in \mathbf{N}^2_{Max}$, if $\neg\exists v \in \mathbf{N}^2_{Max}, n_1, n_2 \in \mathbf{N} \cdot Apply(u, v, u_1, v_1, n_1, u_2, v_2, n_2)$;
- $(u, Crash) \in H$ if $\neg\exists v \in \mathbf{N}^2_{Max} \cup \{Halt\} \cdot (u, v) \in H$;
- $(Halt, Halt) \in H$;
- $(Crash, Crash) \in H$.

Then such a system can be described by a context-free grammar $G = (V, T, P, S)$ where

- $V = \{S, S_1, S_2, U, V, U_1, V_1, U_2, V_2\}$;
- T contains (bounded) vectors from $\mathbf{N}^2$, the additional states Hal and $Crash$, predicates $MaxParal$ and $Apply$, the "true" logical value, logical operators, quantifiers and other symbols, i.e., $T = \mathbf{N}^2_{Max} \cup \{Halt, Crash, MaxParal, Apply, true, \wedge, , \vee, \neg, \exists, \forall, n_1, n_2, \cdot, (,)\}$;
- The production rules are:
  - $p_1 : S \to \neg S$;
  - $p_2 : S \to S \wedge S$;
  - $p_3 : S \to S \vee S$;
  - $p_4 : S \to true$;
  - $p_5 : S \to \exists n_1 \cdot S_1$;
  - $p_6 : S_1 \to \exists n_2 \cdot S_2$;
  - $p_7 : S_2 \to S_2 \wedge S_2$;
  - $p_8 : S_2 \to Apply(U, V, U_1, V_1, n_1, U_2, V_2, n_2)$;
  - $p_9 : S_2 \to MaxParal(U, U_1, V_1, n_1, U_2, V_2, n_2)$;
  - rules that transform nonterminals U, U_1, V_1, U_2, V_2 into vectors from $\mathbf{N}^2$.

Suppose the following rule mutants are defined:

- Mutants for $p_1 : S \to S$;
- Mutants for $p_2 : S \to S \vee S,\ S \to S$;
- Mutants for $p_3 : S \to S \wedge S,\ S \to S$;
- Mutants for $p_4 : S \to \neg true$;
- Mutants for $p_5 : S \to \forall n_1 \cdot S_1$;
- Mutants for $p_6 : S_1 \to \forall n_2 \cdot S_2$;
- Mutants for $p_7 : S_1 \to S \vee S_1,\ S_1 \to S_1$;

- Mutants for p_8 : negate de predicate, change parameters such that the obtained formula is syntactically correct, e.g. switch u and u_1;
- p_9 : negate de predicate, change parameters such that the obtained formula is syntactically correct;
- remaining rules: change each integer value by adding or removing 1.

Now, consider the P system Π with rules $r_1 : a \rightarrow ab$, $r_2 : a \rightarrow c$. Among the P-system mutants generated using the above procedure are the following:

- P systems in which one rules is changed: r_1 can be replaced by any of $\lambda \rightarrow ab$ (this is an invalid rule and the resulting mutant will have no P system equivalent), $aa \rightarrow ab$, $ab \rightarrow ab$, $ac \rightarrow ab$, $a \rightarrow a$, $a \rightarrow b$, $a \rightarrow c$, $a \rightarrow a^2b$, $a \rightarrow ab^2$, $a \rightarrow abc$; r_2 can be substituted in a similar manner. Note that only one rule is mutated at a time.
- r_1 and r_2 are interchanged (this will result in an equivalent mutant).
- A system with rules r_1 and r_2, but which are not applied in a maximal parallel mode (this is obtained by negating the $maxParal$ predicate in the expression of H).
- Other erroneous Kripke systems which may have no P system equivalent; these can be obtained, for example, by negating the *Apply* predicate in the expression of H or by changing one of its "state" parameters (u or v).

Note that in this example we have used a grammar that generates 1-membrane P systems with (at most) two rules and so all generated mutants have this form. More generally, we can use a grammar that describes any 1-membrane P system (with any number of rules). Naturally, in this case the mutant generation process will be much more complex.

5 Conclusions

In many applications based on formal specification methods the test sets are generated directly from the formal models. The same applies to formal models based on grammars. However the approach presented in [11], although novel and with many practical consequences, lacks a rigorous method of defining the process of generating the mutants. In this paper a formal method based rigorously defined operations with rules and subtrees of derivation trees is introduced for context-free grammar formalisms and extended to P systems. Some examples illustrate the approach.

Acknowledgment

Work supported by the CNCSIS grant IDEI 643/2009 (EvoMT).

References

1. F. Bernardini, M. Gheorghe, F.J. Romero-Campero, N. Walkinshaw: A hybrid approach to modelling biological systems. *Workshop on Membrane Computing 2007*, LNCS 4860, 138–159.
2. G. Ciobanu, Gh. Păun, M. J. Pérez-Jiménez, eds.: *Applications of Membrane Computing.* Springer, 2006.
3. Z. Dang, O.H. Ibarra, C. Li, G. Xie: On the decidability of model-checking for P systems. *Journal of Automata, Languages and Combinatorics*, 11, 3 (2006), 279–298.
4. M. Gheorghe, F. Ipate: On testing P systems. *Workshop on Membrane Computing, 2008*, LNCS 5391, 204–216.
5. M. Holcombe, F. Ipate: *Correct Systems: Building a Business Process Solution.* Springer, 1998.
6. J.E. Hopcroft, R. Motwani, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation* (2nd Edition). Addison-Wesley, 2001.
7. F. Ipate, M. Gheorghe: Testing non-determinstic stream X-machine model and P systems. *Electronic Notes in Theoretical Computer Science*, 227 (2009), 113–126.
8. F. Ipate, M. Gheorghe: Finite state based testing of P systems. *Natural Computing*, 2009, to appear.
9. J. Offutt: A practical system for mutation testing: Help for the common programmer. *International Test Conference*, 1994, 824–830.
10. Y.-S. Ma, J. Offutt, Y.R. Kwon: MuJava – An automated class mutation system. *Software Testing, Verification and Reliability*, 15, 2 (2005), 97–133.
11. J. Offutt, P. Ammann, G. Mason, L. (Ling) Liu: Mutation testing implements grammar-based testing. *Proceedings of the Second Workshop on Mutation Analysis*, 2006.
12. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143.
13. Gh. Păun: *Membrane Computing. An Introduction.* Springer, 2002.
14. http://en.wikipedia.org/wiki/Mutation_testing
15. http://cs.gmu.edu/ offutt/mujava/
16. http://ppage.psystems.eu