Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. IV (2009), No. 3, pp. 253-262

# Mutation Based Testing of P Systems

Florentin Ipate, Marian Gheorghe

*Florentin Ipate*
The University of Pitesti
Department of Computer Science, Faculty of Mathematics and Computer Science
Str Targu din Vale 1, 110040 Pitesti
E-mail: florentin.ipate@ifsoft.ro

*Marian Gheorghe*
The University of Sheffield
Department of Computer Science
Regent Court, Portobello Street, Sheffield S1 4DP, UK
E-mail: M.Gheorghe@dcs.shef.ac.uk

Received: April 5, 2009
Accepted: May 30, 2009

**Abstract:** 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. 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.
**Keywords:** mutation testing, P systems, Kripke structures, context-free grammars

## 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 behaviour 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. Furthermore, a comprehensive bibliography of P systems can be found at [16]. Of the many variants of P systems that have been defined, in this paper we consider cell-like P systems without priority 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 behaviour 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 is based on a formal specification or model, in which case the process could be automated. A number of recent papers devise black-box testing strategies for P systems 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 $(x \geq 0\&\&a)$ $y = y+1$; else $y = y+2$;

Then mutants for this code fragment can be obtained by substituting: (i) && with another logic operator, e.g., ||; (ii) $\geq$ with another comparison operator, e.g., $>$, $=$; (iii) $+$ with another arithmetic operators, e.g., $-$; (iv) 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 $(x \geq 0||a)$ $y = y+1$; else $y = y+2$;
if $(x > 0\&\&a)$ $y = y+1$; else $y = y+2$;
if $(x \geq 0\&\&a)$ $y = y-1$; else $y = y+2$;
if $(x \geq 0\&\&a)$ $y = y+1$; else $y = y-2$;
if $(x \geq 0\&\&a)$ $x = y+1$; else $y = y+2$;
if $(x \geq 0\&\&a)$ $y = y+1$; 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 specialised 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.

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 formalisation 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 it. Based on the tree, we formally describe the process of generating the mutants for the given specification. Furthermore, the paper shows how the proposed method can be used to construct mutants for a P system specification.

## 2 Preliminaries

For an alphabet $V = \{a_1, ..., a_p\}$, $V^*$ denotes the set of all strings over $V$; $\lambda$ 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 $(|u|_{a_1}, ..., |u|_{a_p})$. This is denoted by $\Psi_V(u)$.

The concept of context-free grammar is assumed to be known, for details we refer to a classical textbook [6]. Only proper context-free grammar, i.e., with no useless symbols and no $\lambda$ or renaming productions, will be used in this paper. For any derivation from the start symbol to a string of terminal symbols, $w$, a derivation (or parse) tree with the yield, the string of terminals obtained by concatening the leaves from left to right, $w$, is associated. The set of terminal strings derived from the start symbol is called the language generated by the language. 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. In the sequel possibly ambiguous grammars will be considered.

## 2.1 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 systems, 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 $\Pi = (V, \mu, w_1, ..., w_n, R_1, ..., R_n)$, where $V$ is a finite set, called *alphabet*; $\mu$ defines the membrane structure, which is 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, $\mu$, is denoted by a string of left and right brackets ([, and ]), each with the label of the membrane it points to; $\mu$ also describes the position of each membrane in the hierarchy. The rules in each region have the form $u \longrightarrow (a_1, t_1)...(a_m, t_m)$, 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, $u$ 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 $\Pi$, is a tuple $c = (u_1, ..., u_n)$, 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; $c = (u_1, ..., u_n)$ 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, subsequently we will only focus on P systems with only one membrane.

## 2.2 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 left-total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $(s, s') \in H$; $L : S \longrightarrow 2^{AP}$ 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 $AP = \{(var_i = val_i) \mid 1 \leq i \leq n, val \in Val_i\}$. 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 left-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.

## 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 grammar $G = (V,T,P,S)$ and $L(G)$ the language defined by $G$. We assume that, for every production rule $p : A \longrightarrow X_1 \ldots X_k$, 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 $A \longrightarrow X_1' \ldots X_n'$ 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 $A \longrightarrow A$, $A \in V$; 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 $A \longrightarrow X_1' \ldots X_k'$ 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 $A \longrightarrow w$ 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 $A \longrightarrow w$ 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 $A \longrightarrow w$ 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 roots, nodes $X_1, \ldots, X_k$, respectively and $p \in P$ the corresponding production rule of $G$, of the form $A \longrightarrow X_1 \ldots X_k$. 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**) $Tr' = MakeTree(A, Tr_1', \ldots, Tr_n')$, where there exists a mutant $p'$ of $p$ of the form $A \longrightarrow X_1' \ldots X_n'$ such that for every $i$, $1 \leq i \leq n$, there exists $j_i$, $1 \leq j_i \leq k$, such that $Tr_i' = Tr_{j_i}$.

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 3.** *Every mutant of a parse tree from 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.

Let us consider the grammar $G = (V,T,P,S)$ where $V = \{S\}$; $T = \{0,\dots,N\} \cup \{+,-\}$, with $N$ a fixed upper bound; $P = \{p_1, p_2\} \cup \{p_3^i \mid 0 \le i \le N\}$, with $p_1 : S \longrightarrow S+S$, $p_2 : S \longrightarrow S-S$, $p_3^i : S \longrightarrow i$, $0 \le i \le N$. Suppose we have the following rule mutants:

- for $p_1 : S \longrightarrow S-S$ (terminal replacement), $S \longrightarrow S$ (string deletion)
- for $p_2 : S \longrightarrow S+S$ (terminal replacement), $S \longrightarrow S$ (string deletion)
- for $p_3^i$ : $S \longrightarrow i-1$ and $S \longrightarrow i+1$ if $1 < i < N$, $S \longrightarrow 1$ if $i = 0$ and $S \longrightarrow N-1$ 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 neighbouring 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.

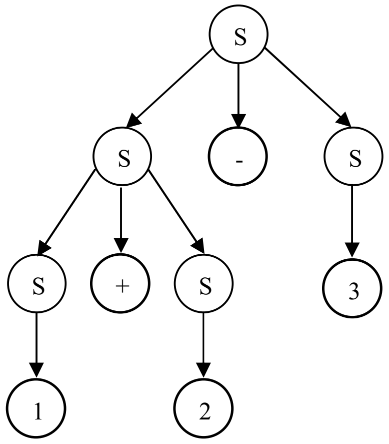

Figure 1: Example parse tree

## 4 P system mutation testing

Consider a 1-membrane P system $\Pi = (V, \mu, w, R)$, where $R = \{r_1, \dots, r_m\}$; each rule $r_i$, $1 \le i \le m$, is of the form $u_i \longrightarrow v_i$, 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 $\Psi_V(u) \in \boldsymbol{N}^k$, where $k$ denotes the number of symbols in $V$. In order to keep the number of configurations finite we will assume that each component of a configuration $u$ cannot exceed an established upper bound denoted $Max$. We denote $u \le Max$ if $u_i \le Max$ for every $1 \le i \le k$ and $N_{Max}^k = \{u \in \boldsymbol{N}^k \mid u \le Max\}$. Analogously to [3], the system is assumed to crash whenever $u \le Max$ does not hold (this is different from the normal termination, which occurs when $u \le Max$ and no rule can be applied). Under these conditions, the 1-membrane P system $\Pi$ can be described by a Kripke structure. In order to define the Kripke structure equivalent of $\Pi$ we use two predicates, *MaxParal* and *Apply*, defined by:

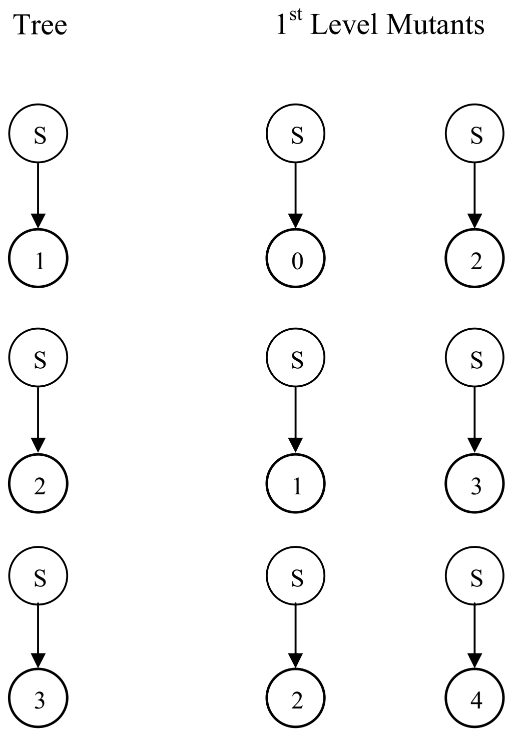

Figure 2: 1st level mutants

$MaxParal(u,u_1,v_1,n_1,\ldots,u_m,v_m,n_m)$, $u \in N^k_{Max}$, $n_1,\ldots,n_m \in \boldsymbol{N}$ signifies that a derivation of the configuration $u$ in maximally parallel mode is obtained by applying rules $r_1 : u_1 \longrightarrow v_1,\ldots,r_m : u_m \longrightarrow v_m$, $n_1,\ldots,n_m$ times, respectively; $Apply(u,v,u_1,v_1,n_1,\ldots,u_m,v_m,n_m)$, $u \in N^k_{Max}$, $n_1,\ldots,n_m \in \boldsymbol{N}$, denotes that $v$ is the result of applying rules $r_1,\ldots,r_m$, $n_1,\ldots,n_m$ times, respectively.

Then the Kripke structure equivalent $M = (S,H,I,L)$ of $\Pi$ is defined as follows: $S = N^k_{Max} \cup \{Halt, Crash\}$ with $Halt, Crash \notin N^k_{Max}$, $Halt \neq Crash$; $I = w$; $H$ is defined by:

- $(u,v) \in H$, $u,v \in N^k_{Max}$, if $\exists n_1,\ldots,n_m \in \boldsymbol{N} \cdot MaxParal(u,u_1,v_1,n_1,\ldots,u_m,v_m,n_m) \wedge$ $Apply(u,v,u_1,v_1,n_1,\ldots,u_m,c_m,n_m)$;
- $(u,Halt) \in H$, $u \in N^k_{Max}$, if $\neg\exists v \in N^k_{Max}, n_1,\ldots,n_m \in \boldsymbol{N} \cdot$ $Apply(u,v,u_1,v_1,n_1,\ldots,u_m,v_m,n_m)$;
- $(u,Crash) \in H$ if $\neg\exists v \in N^k_{Max} \cup \{Halt\} \cdot (u,v) \in H$;
- $(Halt,Halt) \in H$, $(Crash,Crash) \in H$.

It can be observed that the relation $H$ is left-total.

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.

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. On the other hand, it is possible to incorporate the definitions of the two predicates

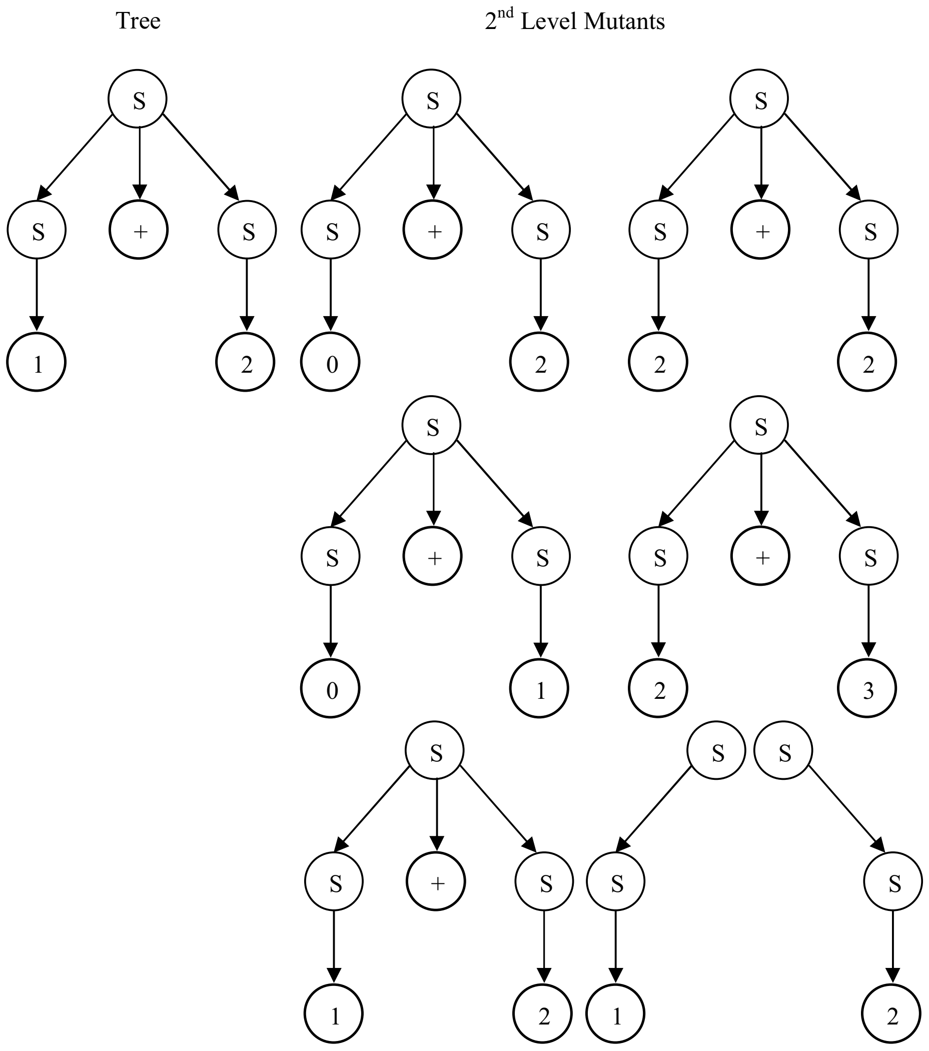

Figure 3: 2nd level mutants

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.

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

**Example 4.** *Consider a 1-membrane P systems with 2 rules $r_1 : u_1 \longrightarrow v_1$, $r_2 : u_2 \longrightarrow v_2$. Then the transition of the Kripke structure representation of $\Pi$ is given by the formulae:*

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

*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 $\boldsymbol{N}^2$, the additional states Halt and*

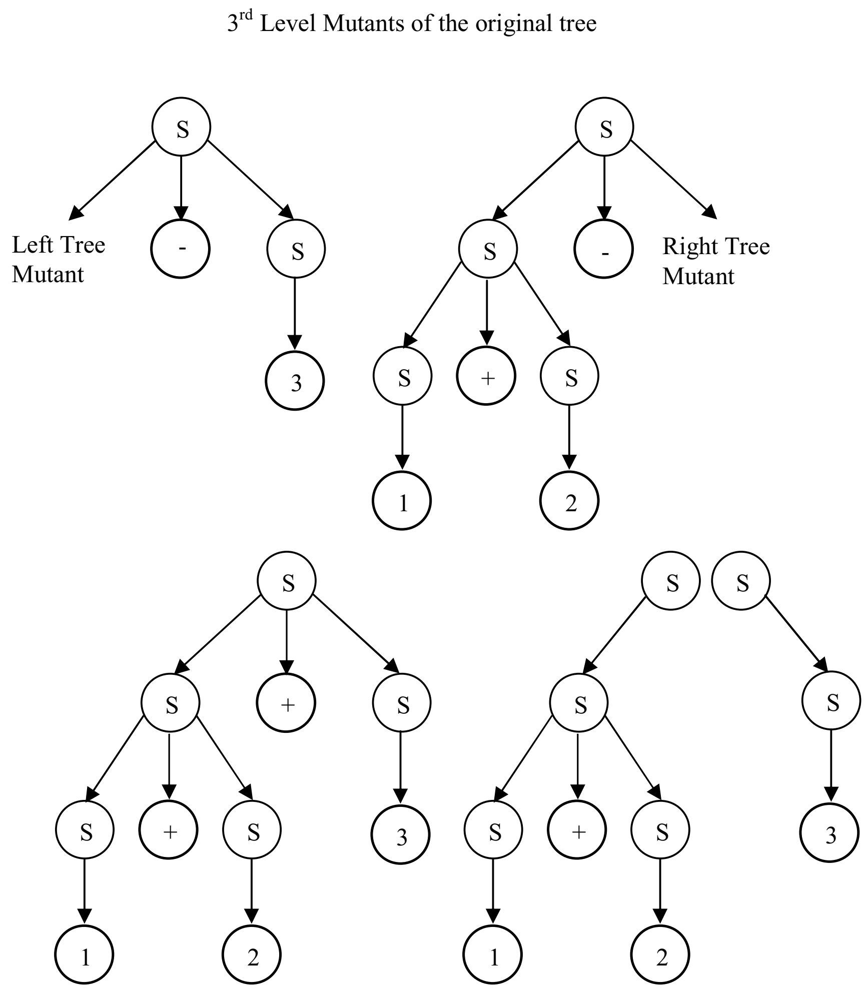

Figure 4: 3rd level mutants

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

*The following mutants can be defined for the rules* $p_1$ *to* $p_7$: $p'_1 : S \longrightarrow S$; $p'_2 : S \longrightarrow S \vee S$, $p''_2 : S \longrightarrow S$; $p'_3 : S \longrightarrow S \wedge S$, $p''_3 : S \longrightarrow S$; $p'_4 : S \longrightarrow \neg true$; $p'_5 : S \longrightarrow \forall n_1 \cdot S_1$; $p'_6 : S_1 \longrightarrow \forall n_2 \cdot S_2$. $p'_7 : S_1 \longrightarrow S \vee S_1$, $p''_7 : S_1 \longrightarrow S_1$. *For* $p_8$ *mutants can be defined by negating de predicate, changing parameters such that the obtained formula is syntactically correct, e.g., switch u and* $u_1$. *Similarly, mutants for* $p_9$ *are obtained by negating de predicate, changing parameters such that the obtained formula is syntactically correct. For the remaining rules mutants are generated by adding* 1 *to or subtracting* 1 *from each integer value.*

## 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 is introduced to rigorously define operations with rules and subtrees of derivation trees for context-free grammar formalisms. This is then extended to P systems and some examples are provided to illustrate the approach. In this paper, the mutation operators are applied to the Kripke structure equivalent of the P system rather than to the P system itself. The advantage of this approach is that test values can be simply generated using a model checking tool (these are the counterexamples returned by the tool). Future work may investigate the application of the mutation operators directly to the P system and the associated test generation process.

**Acknowledgment**. This work is supported by the CNCSIS grant IDEI 643/2009 (EvoMT). The authors are grateful to reviewers for their comments.

## Bibliography

[1] F. Bernardini, M. Gheorghe, J. J. Romero-Campero, N. Walkinshaw, A Hybrid Approach to Modelling Biological Systems, Workshop on Membrane Computing 2007, *Lecture Notes in Computer Science,* Vol. 4860, pp. 138–159, 2007.

[2] G. Ciobanu, Gh. Păun, M. J. Pérez-Jiménez (eds.), *Applications od 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,* Vol. 11, pp. 279–298, 2006.

[4] M. Gheorghe, F. Ipate, On Testing P Systems, Workshop on Membrane Computing, *Lecture Notes in Computer Science,* Vol. 5391, pp. 204–216, 2008.

[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,* Vol. 227, pp. 113–126, 2009.

[8] F. Ipate, M. Gheorghe, Finite State based Testing of P Systems, *Natural Computing,* to appear, 2009.

[9] J. Offutt, A Practical System for Mutation Testing: Help for the Common Programmer, *International Test Conference,* pp. 824–830, 1994.

[10] Y.-S. Ma, J. Offutt, Y. R. Kwon, MuJava: An Automated Class Mutation System, *Software Testing, Verification and Reliability,* Vol. 15, pp. 97–133, 2005.

[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,* Vol. 61, pp. 108–143, 2000.

[13] Gh. Păun, *Membrane Computing: An Introduction,* Springer-Verlag, Berlin, 2002.

[14] http://en.wikipedia.org/wiki/Mutation_testing

[15] http://cs.gmu.edu/ offutt/mujava/

[16] http://ppage.psystems.eu

**Florentin Ipate** was born on 4th December 1967 in Constanta. FI holds a PhD and MSc degrees with the University of Sheffield and a BSc with Politehnica University of Bucharest, all in Computer Science. He is now a professor of Computer Science and PhD supervisor with the University of Pitesti. He has been awarded In Hoc Signo Vinces Prize for research and publications, by the National Research Council for Higher Education, Romania, 2002 and COPYRO Publishing Prize for Computer Science, Romania, 2000. FI's research interests are in specification and model based testing, formal specification languages for software systems, agile modelling and testing, modelling and testing biology-inspired computing systems. His main research results have been published in a research monograph with Springer and in high profile journals.

**Marian Gheorghe** was born on 2nd February 1953 in Bucharest. MG holds a PhD and a BSc with the University of Bucharest. He is now Senior Lecturer with the University of Sheffield and head of the Verification and Testing Group. MG's research interests are in formal computational models, verification and testing, modelling biological systems, agent technologies, artificial life, empirical software engineering. He has published in important international journals and is featured in the main computer science publications database, DBLP, with around 60 items.