Design for a P System Program - A P System Editor and Simulator

Ren Tristan A. de la Cruz: SN#2008-35070

June 22, 2020

1 Preliminaries

1.1 Membrane Computing

Membrane computing is a field of theoretical computer science that studies different models of computation known as *P systems*. The term '*P systems*' refers to a family of models of computation which are inspired by biological processes. P system models use abstractions of biological processes as computational operations. For example, different types of rules (operations) used by most P system variants are abstractions of processes like *chemical reaction* and *ion transport* that occur inside biological cells. Most P system variants use *object symbols* as the objects of computation. One can think of these object symbols as abstraction of physical *molecules* or *ions*. P systems store *multisets* of these object symbols inside regions enclosed by *membranes*. A P system has a collection of these membranes with multisets of objects symbols inside. The membranes can be 'connected' to each other to form a *membrane structure*.

1.2 Components of a P System

Currently, there are tens, if not hundreds, of P system variants. The following sections describe the components that are common to most (if not all) P system variants.

1.2.1 Membrane Structure

A P system has set of membranes that can be connected to each other. These connected membranes form a *membrane structure*. Early P systems are described as *cell-like* since their membrane structures are inspired by nested membranes inside a cell. Figure **??** shows an example of a cell-like membrane structure.

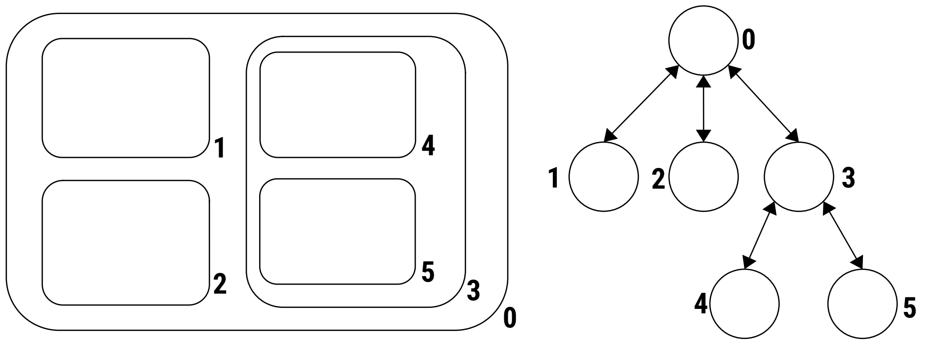


Figure 1: Cell-like Membrane Structure

In Figure **??** you can see two representations of a 6-membrane cell-like membrane structure. The left diagram shows the nesting of the membranes while right diagram shows the tree topology of the membranes. The outer most membrane, membrane 0, is the root node of the tree while membranes that do not contain internal membranes (membranes 1,2,4,5) are the leaf nodes of the tree.

Other P system variants use the more general graph membrane structure. P systems described as *tissue-like* or *neural-like* use graph membrane structures. In *tissue-like* P systems, the membranes are called *cells* and the connected cells form a '*tissue*'. In *neural-like* P systems, the membranes are called *neurons* and the connected neurons form a *neural network*. Figure **??** shows an example of a graph membrane structure with 7 membranes.

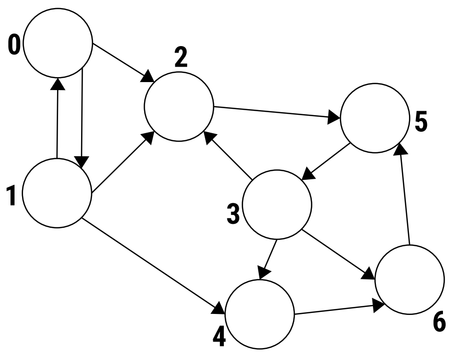


Figure 2: Graph Membrane Structure

1.2.2 Multisets of Object Symbols

As mentioned in Section **??**, the objects of computation in P systems are abstract object symbols. Any P system has a fixed set of object symbols or an *alphabet*. We denote this alphabet of object symbols as V. A multiset over alphabet V is simply a set of object symbols from V in which multiple instances of the same object symbols are allowed. For example, given the alphabet $V = \{a, b, c, d\}$, the following are some multisets over V:

- $\{a, a, b\}$
- $\{a, a, b, c, c, d\}$
- $\{c, d, d\}$
- $\{a, c, c, c, d, d\}$
- $\{a, a, b, b, b, c, c, c, c, d, d, d, d, d\}$

Multisets can be written as strings. For example:

- $\{a, a, b\} = aab = a^2b$
- $\{a, a, b, c, c, d\} = aabccd = a^2bc^2d$
- $\{c, d, d\} = cdd = cd^2$
- $\{a, c, c, c, d, d\} = acccdd = ac^3d^2$
- $\{a, a, b, b, b, c, c, c, c, d, d, d, d, d\} = aabbbccccddddd = a^2b^3c^4d^5$

In a P system, regions enclosed by membranes store multisets of object symbols. Figure **??** shows two examples of membrane structures (cell-like and graph) with multisets of symbols in the regions.

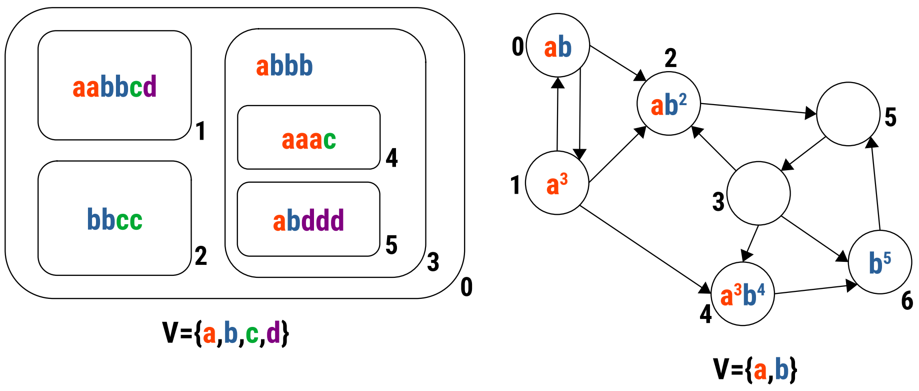


Figure 3: Multisets in Membrane Structures

The left diagram in Figure **??** shows the membrane structure from Figure **??** with multisets over alphabet $V = \{a, b, c, d\}$. Membrane 0 is empty (or contains the empty multiset). Membrane 1 contains the multiset *aabbcd*. Membrane 2 contains the multiset *bbcc*. Membrane 3 contains the multiset *abbb*, etc. The right digram in Figure **??** shows the membrane structure from Figure **??** with multisets over alphabet $V = \{a, b\}$. Cell 0 contains the multiset ab. Cell 1 contains the multiset a^3. Cell 2 contains the multiset ab^2. Cell 3 is empty, etc.

1.2.3 Rules of Different Types

A P system uses a set of rules to perform computation. Different P system variants use different types of rules. In general, a rule transforms the multisets inside the membranes and/or transforms the membrane structure itself. Static structure P systems only use rules that primarily transform the multisets inside the membranes and do not change the structure of the system. Dynamic structure P systems have rules that can add new membranes, delete existing membranes, and change the connections between membranes.

Rules for transforming multisets inside the membranes are based on *multiset rewriting rules*. A *multiset rewriting rule* is written as $M \rightarrow M'$ where M and M' are both multisets over the same alphabet. The rule rewrites the multiset M to the multiset M'.

For example: Alphabet is $V = \{a, b, c, d\}$

- `Rule 1`: $a^2b^2 \rightarrow cd$
  - Applying `Rule 1` once to multiset $a^4b^5c^3d^2$ will transform it to $a^2b^3c^4d^3$.
  - Applying `Rule 1` twice to multiset $a^4b^5c^3d^2$ will transform it to bc^5d^4.
- `Rule 2`: $ad \rightarrow ab^2c^3$
  - Applying `Rule 2` once to multiset $a^4b^5c^3d^2$ will transform it to $a^4b^7c^6d$.
  - Applying `Rule 2` twice to multiset $a^4b^5c^3d^2$ will transform it to $a^4b^9c^9$.

The first P system, known as *transition P system*, is a cell-like P system that uses modified multiset rewriting rules. Rules in transition P systems are multiset rewriting rules that use the membrane structures of the systems. A rule is associated with the region enclosed by a specific membrane. A rule will 'consume' a multiset of object symbols in its region then it can produce multisets of object symbols in the same (*here*) region, in the region outside (*out*) its membrane, and/or in the regions enclosed by membranes inside (*in*) the rule's own membrane. Figure **??** shows an example of a rule being applied.

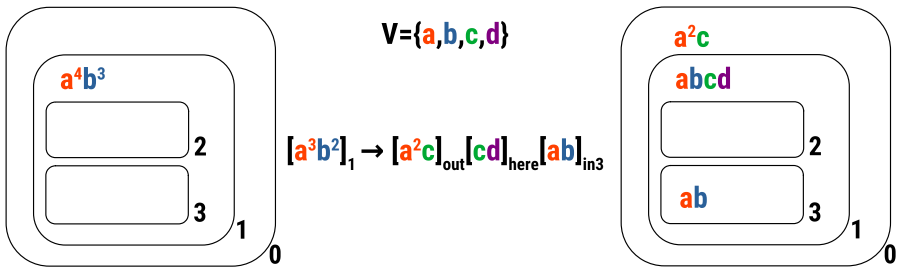


Figure 4: Application of Transition Rule $[a^3b^2]_1 \rightarrow [a^2c]_{out}[cd]_{here}[ab]_{in_3}$

The left diagram on Figure **??** shows the *configuration* of the system before rule application. Membrane 1 contains the multiset a^4b^3 while the other membranes are empty. The rule $[a^3b^2]_1 \rightarrow [a^2c]_{out}[cd]_{here}[ab]_{in_3}$ is associated with the region enclosed by membrane 1. The rule consumes a^3b^2 from membrane 1, produces a^2c outside membrane 1 (in membrane 0), and produces ab inside membrane 3. The right diagram on Figure **??** shows the configuration of the system after the rule is applied.

It can also be specified in a rule in a transition P system that the membrane associated with the rule be *dissolved* after the rule was applied. Figure **??** shows how a rule can *dissolve* a membrane. Rule $[a^2b^2]_1 \rightarrow [ab]_{in_2}[cd]_{in_3}\delta$ consumes multiset a^2b^2 from membrane 1, produces multiset ab in membrane 2, produces multiset cd in membrane 3, and *dissolves* membrane 1 (this is specified by δ at the end of the rule). The remaining multiset in membrane 1 will go to its parent membrane, membrane 0.

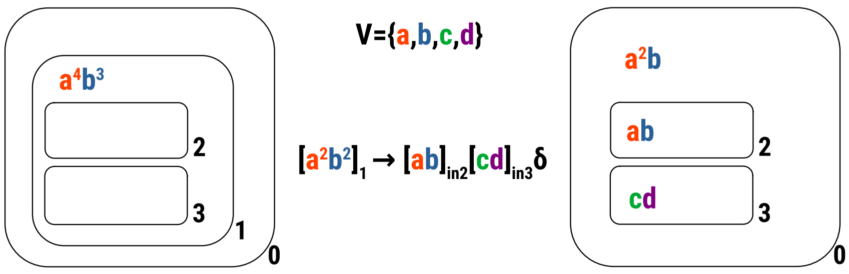


Figure 5: Application of Transition Rule $[a^2b^2]_1 \to [ab]_{in_2}[cd]_{in_3}\delta$

Tissue P systems' rules are very similar to rules in transition P systems. The difference is that tissue P systems' membrane structures form graphs that do not have directionality or orientation unlike the tree membrane structures of transition P systems. When you are in the node of a tree in the membrane structure of a transition P system, the rule can send produced multisets *out* in the direction of the parent node, in the same node (*here*), or in the direction of the children nodes (*in*). In tissue P systems, a rule can send produced multisets in the same node (*here*) or to the adjacent nodes (*out*). Aside from this rule convention, rules in tissue P systems also have multiple different semantic meanings.

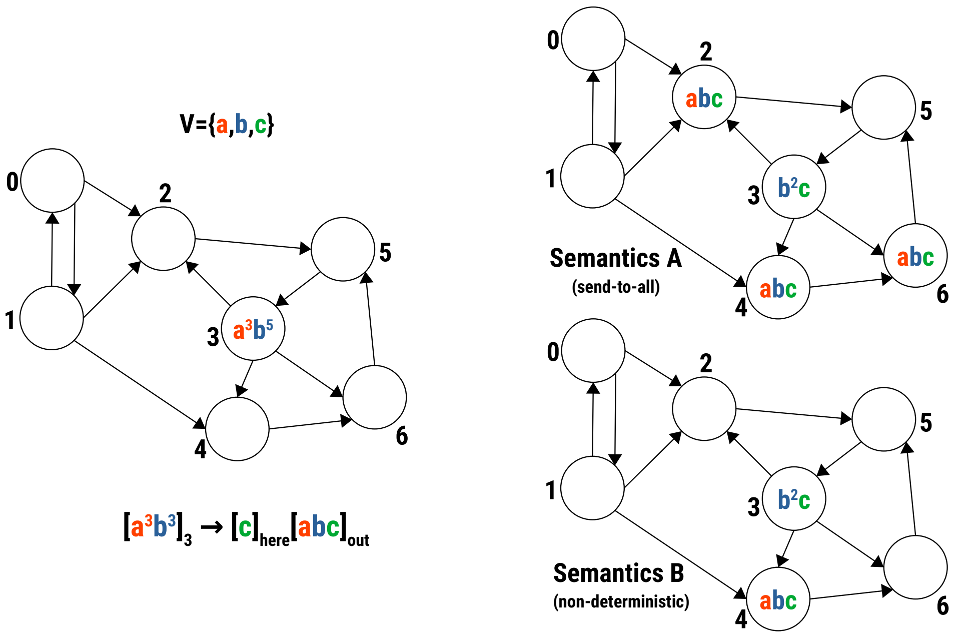


Figure 6: Application of Tissue P system Rule $[a^3b^3]_3 \to [c]_{here}[abc]_{out}$

Figure **??** shows two interpretations (Semantics A and B) for the rule $[a^3b^3]_3 \to [c]_{here}[abc]_{out}$. For semantics A, the rule consumes multiset a^3b^3 from cell 3, produces multiset c in cell 3, and sends multiset abc to all adjacent cells (cells 2,4,6). For semantics B, the rule consumes multiset a^3b^3 from cell 3, produces multiset c in cell 3, and *nondeterministically* sends multiset abc to one of the cells adjacent to cell 3 (either cell 2,4, or 6).

P systems *with active membranes* introduced a type of rule that allows creation of new membranes. Figure **??** shows a rule of this type. The rule $[a]_y \to [b]_y[c]_y$ in Figure **??** has a very similar syntax to previous rules but its semantics is different. The rule consumes the object symbol a in membrane labeled y. The rule will then duplicate the membrane copying all the remaining object symbols to both membrane copies. Both membrane copies are also labeled y. In this P system variant, membrane labels are not membrane ids so multiple membranes can have the same label. In one of the copies, object symbol b is produce while in the other copy object symbol c is produced. Figure **??** shows the rule being applied twice in two steps.

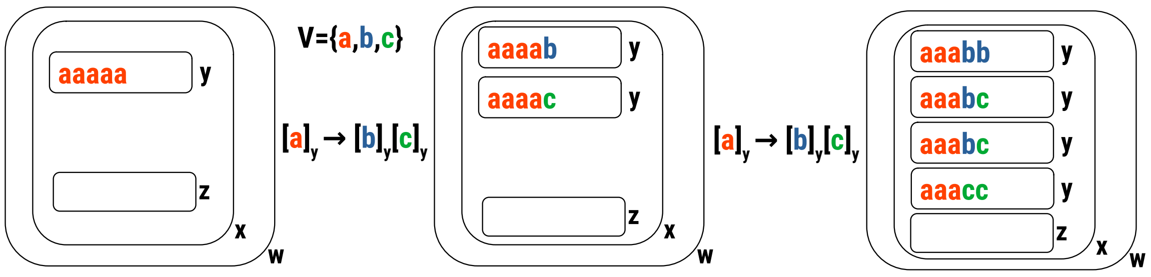


Figure 7: Application of the Active Membrane Rule $[a]_y \rightarrow [b]_y[c]_y$

The types of rules described above form a small set of samples of all possible types of rules. A large portion of the other types of rules that are not discussed are variations of the rules above. Some of them may have a slightly different semantics to the rules above.

1.2.4 P System Derivation Modes

P systems are parallel computing models. A P system can apply multiple rules at the same time. Different P systems can have different conventions that say which combinations of rules can be applied at the same time. This convention is known as the *derivation mode* of the system. A common derivation mode is called *maximally parallel* derivation mode. In maximally parallel mode, the system will apply a combination of rules such that you can no longer add instances of rules to this *maximal* combination of rules. For example, you have:

- `Rule 1:` $[ab]_1 \rightarrow [xy]_{here}$
- `Rule 2:` $[b^2]_1 \rightarrow [xyz]_{here}$

`Rule 1` consumes ab from membrane 1 while `Rule 2` consumes b^2 from membrane 1. If membrane 1 contains the multiset a^3b^6, one maximal combination of rules is 3×`Rule 1` + `Rule 2`. This combination will consume the multiset a^3b^5. With the multiset b remaining no additional rules can be added to the combination making the combination maximal. Another maximal combination of rules is 3×`Rule 2`. This combination will consume the multiset b^6. With the multiset a^3 remaining no additional rules can be added. Another maximal combination is 2×`(Rule 1 + Rule 2)` consuming the multiset a^2b^6. In maximally parallel mode, only a maximal combination of rules can be applied. For example, the system can not apply the combination with one instance of `Rule 1`, if the system can still apply additional rules then it will add these rules to the combination. In this mode, the system *maximizes* the rules it can apply.

Minimally parallel mode is often used in *neural-like* P system variants. In the minimally parallel mode, at most one applicable rule is applied per membrane. The 'parallel' part is that multiple membranes can apply a rule at the same time. The 'minimal' part is that at most one rule can be applied per membrane.

There are other derivations modes. Some P system variants have the same rule types and membrane structure but only differ in the derivation modes they use.

1.3 Purpose of a User-friendly General P Systems Viewer and Simulator

Manually designing (using pen and paper) a P system is easy for smaller systems (with fewer than 20 membranes). It gets increasing difficult the larger the system is. You have to keep track of the multisets of object symbols and the rule set per membrane and the connections between the membranes. Manually simulating P systems is much more difficult than designing one even for smaller systems because the components of the system work in parallel (membranes applying rules at the same time). Additionally, non-deterministic systems are much more difficult to simulate manually since you have to keep track of branching computations.

P-lingua is well-known P system simulator but it is not particularly user-friendly and its support for P system variants is limited to the few popular variants. It is a command-line based simulator and has no built-in P system visualizer. It gives the simulation as text. When creating a system for the P-lingua simulator, the user has to explicitly type out the system definition in a file.

The purpose of a user-friendly general P system simulator and viewer program is to simplify the process of designing and simulating P systems. The program should help the users design P systems by providing a user-friendly/intuitive interface that they can use to create and modify P systems. The program should also be able to perform simulation of the system. Additionally, the program should be general enough to accommodate most P system variants.

2 Design of the P System Program

2.1 P System Program - Editor and Simulator

To avoid confusion, we are going to use ‘program’ for the system being designed while the term ‘systems’ will refer to P systems (the objects of simulation). The P system program will be a standalone P system editor and simulator. The user is the only one interacting with the program. There are no external programs that interact with the P system program. The purpose for designing the program will to be self-contained/standalone is to make it portable and easy to use for membrane computing researchers.

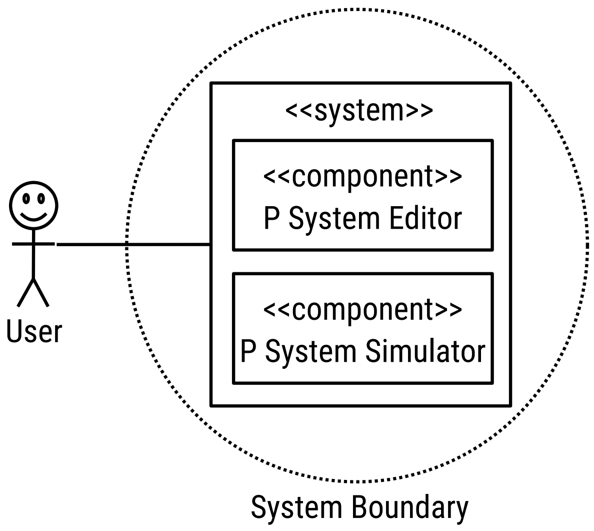


Figure 8: Context for the P System Program

The program will have two main components: the editor and the simulator. The P system editor contains the user interface to control the program while the P system simulator contains the functions needed to generate the P system’s computation tree.

2.2 Representing P Systems

We have already described components of a P system in Section ??. In this section, we construct different classes to represent the components of the P system. We also show how these components (and classes) are related to each other.

1. `Membrane` class represents membranes in the P system (see Section ??). A membrane has three components, the membrane’s *id*, the membrane’s *label*, and the membrane’s *content* which is a multiset over a given alphabet. The `Membrane` class has `id` (integer), `label` (string), and `content` (multiset) attributes and the additional `coordinate` (pair of integers) attribute.

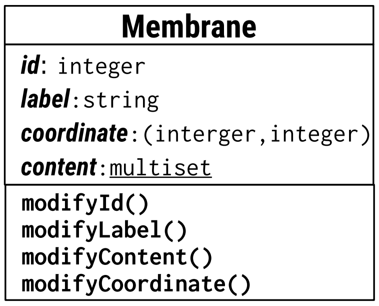


Figure 9: Membrane Class

The `id` attribute is for identifying the membrane. The `label` attribute is a string label of the membrane. The `content` attribute is a multiset type. If we are using a multiset over an alphabet of size n, then a multiset is simply an array of integers of size n. For example, for an alphabet $V = \{a, b, c\}$, a multiset over V is an array of (non-negative) integers of size 3. The multiset `[3,2,0]` refers to the multiset $a^3b^2c^0$.

The `coordinate` attribute will be used when visualizing the membrane. The membrane will be drawn in 2-dimensional space so a 2D coordinate is needed to specify a membrane's location.

The four methods of the class, `modifyId()`, `modifyLabel()`, `modifyContent()`, `modifyCoordinate()`, are used to modify the values of the corresponding attributes.

2. `Configuration` class represents the *configuration* (or state) of the P system (see Sections **??** and **??**). The *configuration* of a P system is its network of membranes. The `configuration` class has the following attributes: `id` (integer), `time` (integer), `membranes` (set of `membranes`), `connections` (set of `membrane` id pairs).

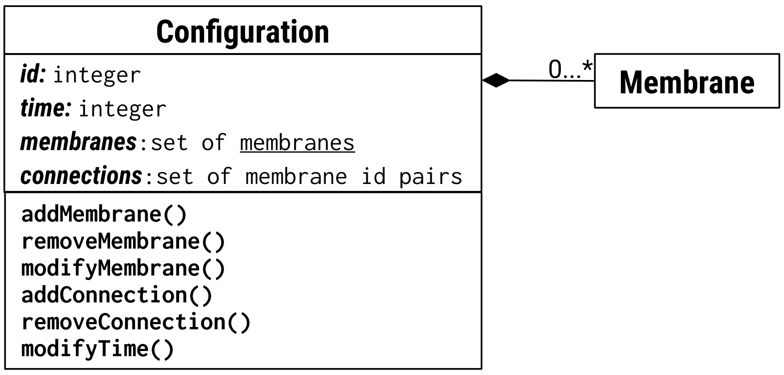


Figure 10: Configuration Class

The `id` attribute is for configuration identification. The `time` attribute is the time step when the P system had the configuration. The initial configuration of the P system has the `time` attribute set to `0`. The next configurations will have the `time` attribute set to `1`. The `membranes` attribute is the set of `membrane` objects while the `connections` attribute is the set of pairs of `membrane` object `ids` specifying the connections between membranes.

Method `addMembrane()` is used to add a `membrane` to the `membranes` attribute of the `configuration` while method `removeMembrane()` is used to remove a `membrane`. Method `modifyMembrane()` is used to modify attributes of a `membrane`. Methods `addConnection()`, `removeConnection()` are used to add or remove pairs of `membrane` ids. Method `modifyTime()` sets the `time` attribute of the configuration.

3. `Rule` class represents P system rules (see Section **??**). The class has the following attributes: `id` (integer), `condition` (`configuration` type), `action` (string).

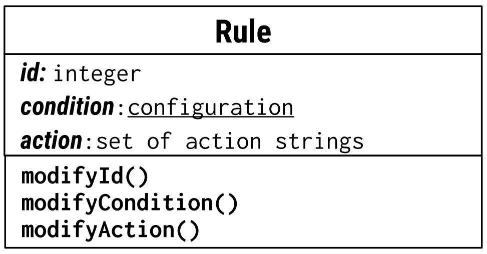


Figure 11: Rule Class

The `id` attribute is for rule identification. The `condition` attribute is a `configuration` type object. A configuration is basically a graph/network of membranes. The `condition` attribute is a subgraph of membranes that the rule will look for in order for it to be applicable. For example, given a reference `configuration A`, a rule is applicable with respect to `configuration A` only if the rule's `condition`, `configuration B`, is a subgraph of `configuration A`. The `action` attribute of the rule is a set of strings that specify the modifications/transformations it will perform on a given `configuration`. e.g. The string `rc-1-3` will call `removeConnection()` of a `configuration` to remove the connection `(1,3)` from the `configuration`. The string `am-<id>-<lb>-1-2-3`, assuming the alphabet $V = \{a, b, c\}$, will call `addMembrane()` of a `configuration` to add a membrane with `id=<id>`, `label=<lb>`, and `content=[1,2,3]` (or $a^1b^2c^3$).

The methods `modifyId(), modifyCondition(), modifyAction()` are used to modify the attributes of the `rule`.

4. `Rule set` class simply aggregates `rules` into a set. Its only attribute is `set` which is a set of `rules`. Methods `addARule(), removeARule(), modifyARule()` are used to modify the `rule set`.

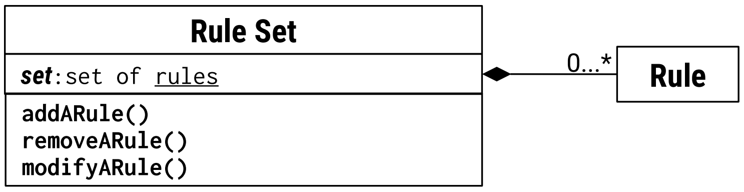


Figure 12: Rule Set Class

5. `P system` class represents P systems (see Section ??). The class has the following attributes: `id` (integer), `system_configuration` (`configuration`), `alphabet` (set of symbols), `system_rule_set` (`rule set`), `derivation_mode` (string).

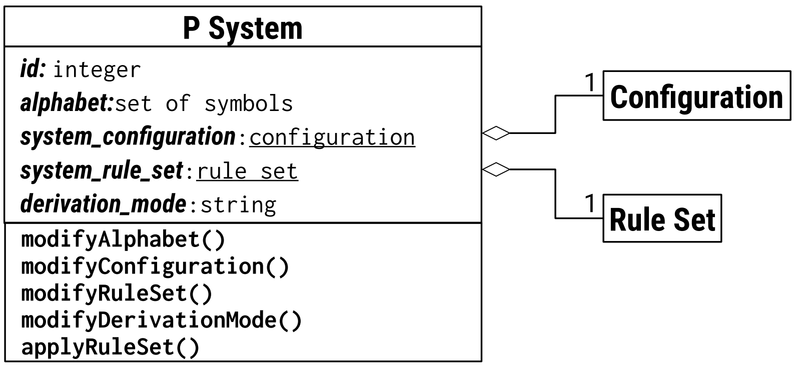


Figure 13: P System Class

The `id` attribute is for P system identification. The `alphabet` attribute is a set of symbols representing the types of objects in the P system. The `system_configuration` attribute is the initial *configuration* of the P system. The `system_rule_set` attribute is the set of rules of the P system. The `derivation_mode` attribute is a string that defines the allowable combination of rule that the P system can apply (see Section ??).

Methods `modifyAlphabet(), modifyConfiguration(), modifyRuleSet(), modifyDeriviationMode()` are used to modify the attributes of the `P system`. The `applyRuleSet()` takes allowed combination of applicable `rules` from the P system's `system_rule_set` and use the `rules`' `actions` to modify the P system's `system_configuration`.

6. `Transition` class represents transitions from one *configuration* to another *configuration* via application of combinations of applicable rules. The class has the following attributes: `id` (integer), `pre_transition_id` (integer), `post_transition_id` (integer), `used_rules` (set of integer).

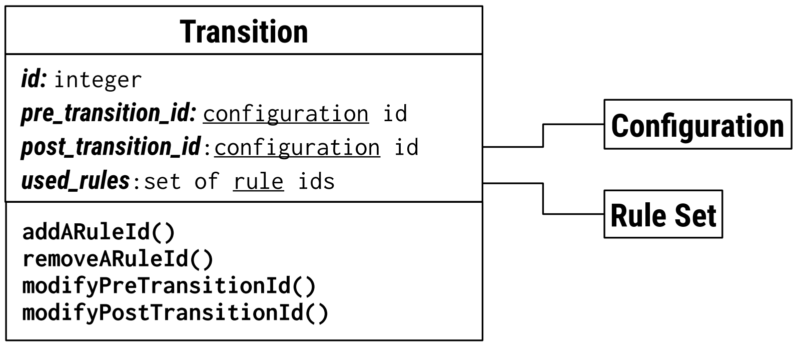


Figure 14: Transition Class

The `id` attribute is for transition identification. The `pre_transition_id` attributed is an integer that is the `id` of the configuration before transition (before rules are applied) while the `post_transition_id` is an integer that is the `id` of the configuration after transition. The `used_rules` attribute is the set of integers which are the `ids` of the rules used in the transition.

Methods `addARuleId(), removeARuleId(), modifyPreTransitionId(), modifyPostTransitionId()` are used to modify the attributes of the `transition`.

7. `Computation` class represents the *computation tree* of the P system. The *computation tree* is a tree-like network of *configurations* the root of which is the initial configuration of the P system. Starting from the root (initial) configuration, new configuration nodes are generated by applying allowed combinations of applicable rules to the root configuration then adding the resulting new configurations to the tree. Branching will occur in the computation tree if the P system is non-deterministic such that it is possible to apply more than one combination of rules to a given configuration. If multiple valid combinations of rules are applied to a configuration resulting to multiple next configurations, then these are 'branches' in the computation tree.

   The class has the following attributes: `id` (integer), `configuration_set` (set of `configurations`), `transition_set` (set of `transitions`).

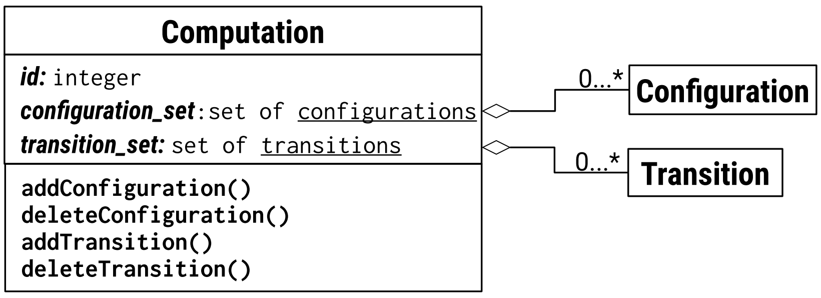


Figure 15: Computation Class

The `id` attribute is for computation identification. The `configuration_set` attribute is the set of `configurations` that are the nodes of the computation tree. The `transition_set` is the set of `transitions` between nodes of the computation tree.

Methods `addConfiguration()`, `deleteConfiguration()`, `addTransition()`, `deleteTransition()` are used to modify attributes of the `computation`.

8. `PSystem-Computation` class simply combines a `P system` and its `computation`. It has the `system` attribute for the *P system* part and the `computation` attribute for the *computation* part. Methods `modifySystem()` and `modifyComputation()` are used to modify the attributes of the `system-computation`.

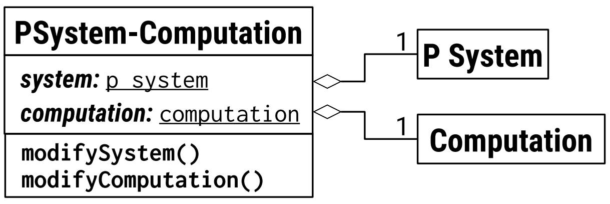


Figure 16: Simulation Class

2.3 User and P System Program Interactions

The P system program's input and output is a `PSystem-Computation` object that will stored in a file that we will call a *.psy* file. The user can use two general functionalities of the P system program: the editor functionality and the simulator functionality. As a P system editor, the user use the P system program to open/create a *.psy* file and edit the `P system` stored in the file using a graphical user interface. As a P system simulator, the user can open a *.psy* file and *simulate* (generate the computation tree) the `P system` stored in the file. More specific use-cases for the P system program is shown in Figure **??**.

In the following sections, the term *simulation* will also refer to the computation tree or the `computation`-type object.

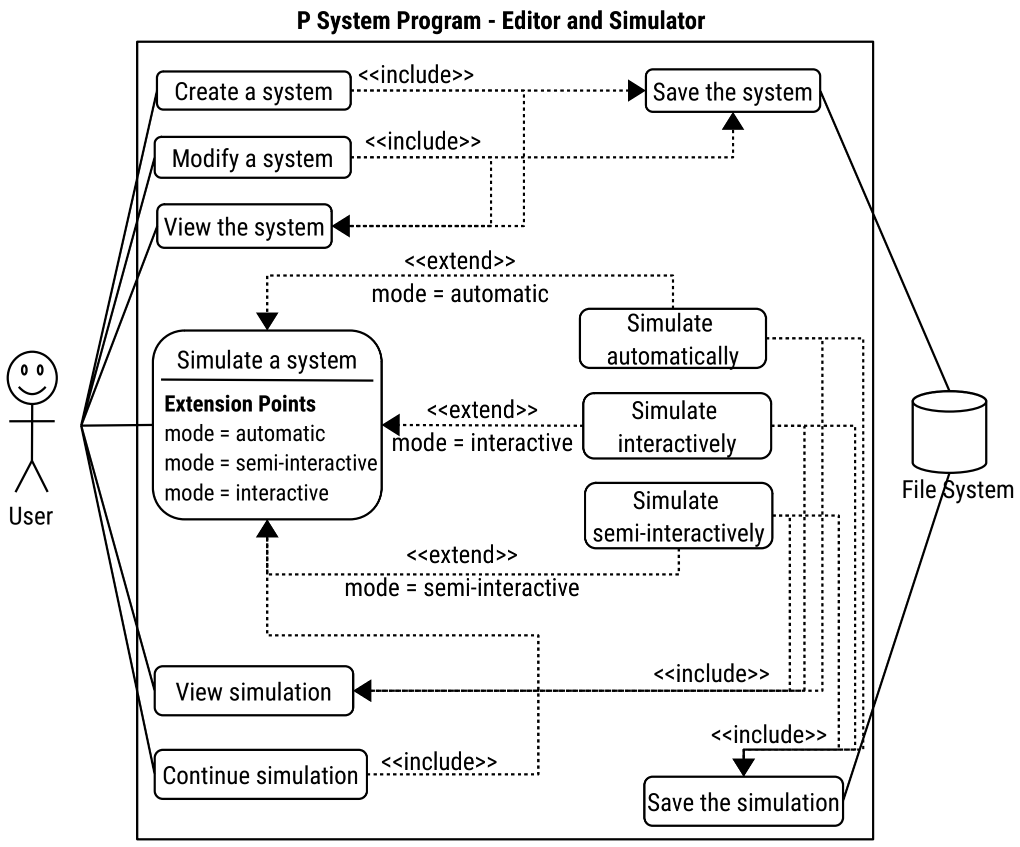


Figure 17: Use-cases for the P System Program

The P system program has the following six primary (user-facing) use-cases:

1. `Create a System:` The user can create a P system from scratch and save the system in a `PSystem-computation` type object stored in a *.psy* file.
2. `View a System:` The user can open an existing *.psy* file and view the P system stored in the file.
3. `Modify a System:` The user can open an existing *.psy* file and modify the P system stored in the file.

4. `Simulate a System:` The user can open/create a *.psy* file then use the P system stored in the file to generate a computation tree. The `computation` attribute of the `PSystem-computation` object will store the simulation while the `system` attribute will store the P system. There are 3 modes of simulation: *automatic*, *interactive*, and *semi-interactive*.

   - In `mode=automatic`, the user will choose if the computation will be *linear* or *branching*. The user will also specify simulation halting criteria which include the *maximum computation depth* and *maximum number of configurations*. In this mode, the simulator part of the program will generate the configurations in the computation tree until one of the halting criteria is met.
   - In `mode=interactive`, the simulator will generate configurations one step at a time at user's control. If, in a given time step, there are multiple combinations of rules possible, the user will be prompted to select one of the combinations to apply. The simulation will then proceed to the next configuration.
   - In `mode=semi-interactive`, the simulator will generate the configurations automatically as long as there are no branching involve. If there is branching in the computation, the user will be prompted to select one of the possible combinations of rules to apply.

   After simulation the computation can then be saved in the same *.psy* file.

5. `View Simulation:` The user can open a *.psy* file that contains simulation/computation of the system and view the simulation using the P system editor (GUI).

6. `Continue Simulation:` The user can open a *.psy* file that contains computation of the system and then continue the simulation in either of the 3 modes.

Figure **??** shows an example GUI for the P system program. The GUI contains the main *View Box* that shows the configuration of the P system. It has a *Rules Box* that lists the rules of the system. It has an *Add Box* that contains thing you can add to the P system. It has a *Edit Box* that shows selected elements which can then be modified. Figure **??** also shows some menu options.

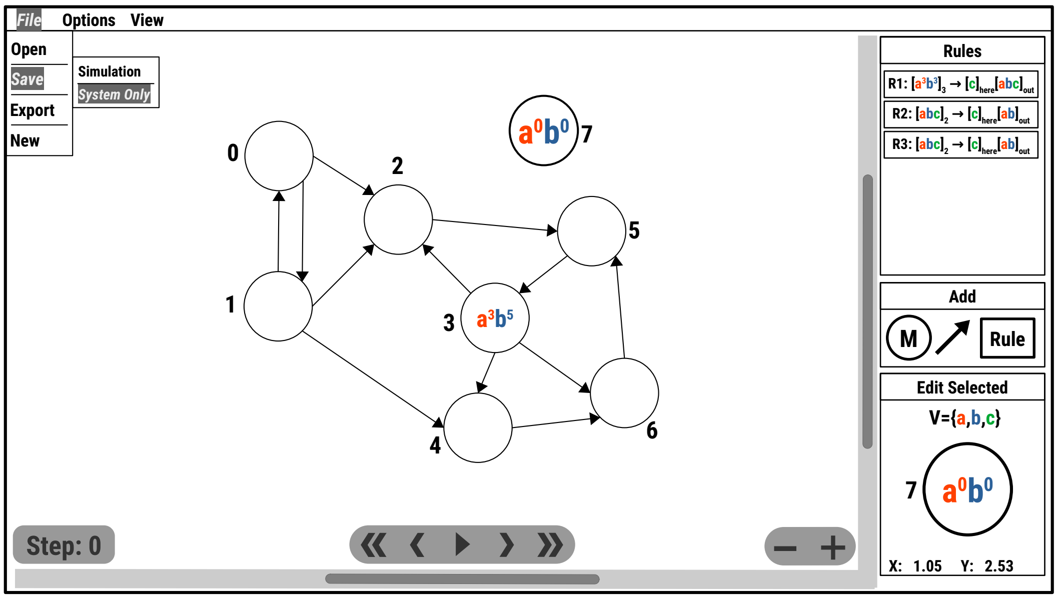


Figure 18: User Interface

Figure **??** shows the menu option for the different simulation mode.

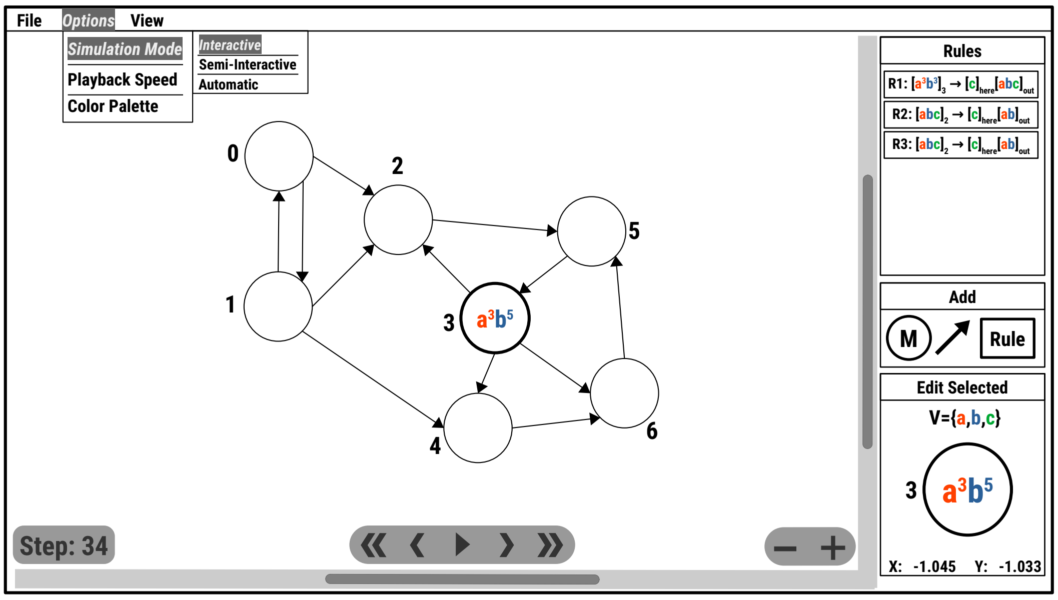


Figure 19: User Interface

Figure ?? shows the *computation tree* view. The *view box* in the computation tree view will show parts (some configuration) of the computation tree instead of showing a selected configuration. A preview of the selected configuration will be shown in the *edit box*.

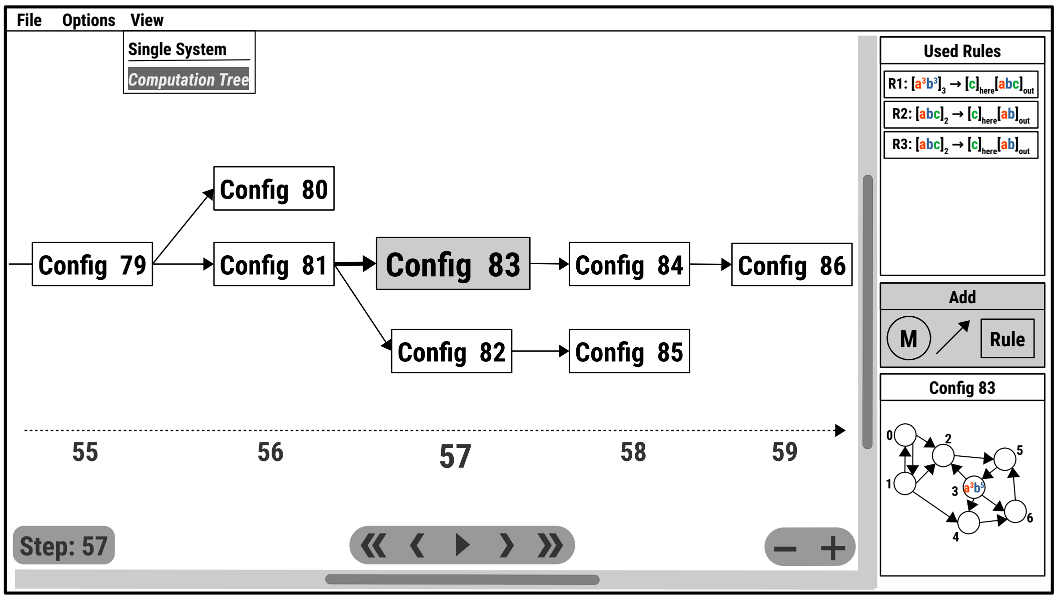


Figure 20: User Interface

The interface has the `(- +)` control for zooming in and out of the P system in the *single system* view. The interface also has scroll bars for changing the position of the *view box*. The ⟨⟨ ⟨ ▶ ⟩ ⟩⟩ controls are for the simulation.

2.4 Interactions between User, P System Editor, and P System Viewer

Figure ?? shows the interactions mainly between the user and the P system editor (GUI) for the `Create a System` use-case.

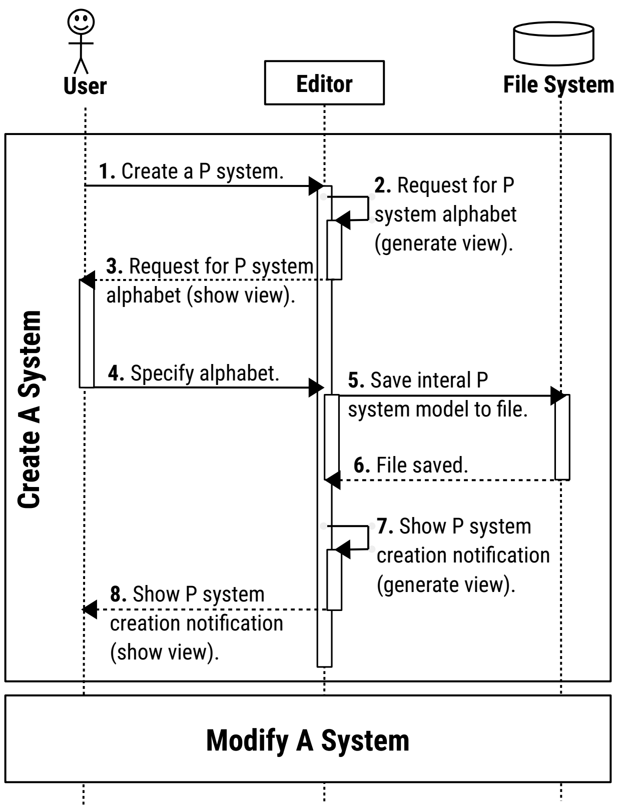
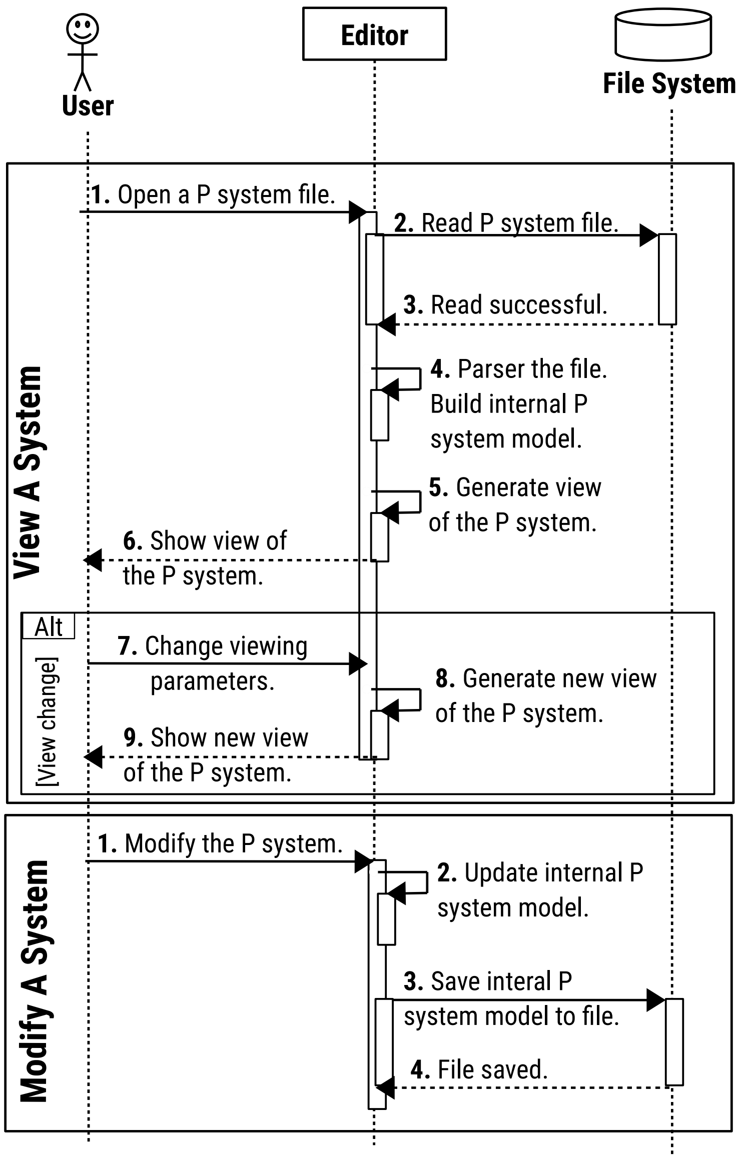

User
Editor
File System
Create A System
1. Create a P system.
2. Request for P system alphabet (generate view).
3. Request for P system alphabet (show view).
4. Specify alphabet.
5. Save interal P system model to file.
6. File saved.
7. Show P system creation notification (generate view).
8. Show P system creation notification (show view).
Modify A System

Figure 21: Sequence Diagram for `Create a system` Use-case

Creation of a new system will start with the user selecting `File > New` in the menu options (see Figure **??**). The P system editor will need to ask the user for the alphabet of the system. The editor will then generate an updated view (by view we mean the general interface). The user will be prompted, by the updated view, to specify the P system's alphabet. After specifying the alphabet the editor can now create an internal `PSystem-Computation` object and create the initial template for the `system` attribute (P system) of the object. The object will then be stored to a *.psy* file in the file system. The editor will generate an updated view with creation notification and show this view to the user.

Figure **??** shows the sequence diagram for the `View a System` and `Modify a System` use-cases.



User
Editor
File System
View A System
1. Open a P system file.
2. Read P system file.
3. Read successful.
4. Parser the file. Build internal P system model.
5. Generate view of the P system.
6. Show view of the P system.
Alt
[View change]
7. Change viewing parameters.
8. Generate new view of the P system.
9. Show new view of the P system.
Modify A System
1. Modify the P system.
2. Update internal P system model.
3. Save interal P system model to file.
4. File saved.

Figure 22: Sequence Diagram for `View a System` and `Modify a System` Use-cases

The user will open a *.psy* file that contains the `PSystem-Computation` object with the P system definition in its `system` attribute. The editor will read and parse the file then generate the view of the P system in is *view box*. The user can then view the system and change the view box by scrolling or changing the zoom value. The user can also modify the P system by adding component to it using controls in the *add box*. If there are any changes to the P system, the editor will update its internal copy of the P system then it will write any updates to the *.psy* file. `View simulation` use-case a similar sequence diagram to `View a System` use-case. The only difference is the editor has specific procedure for parsing computation and has a specific procedure for generating its view.

Figure **??** shows the sequence diagram for the `Simulate a System` use-case.

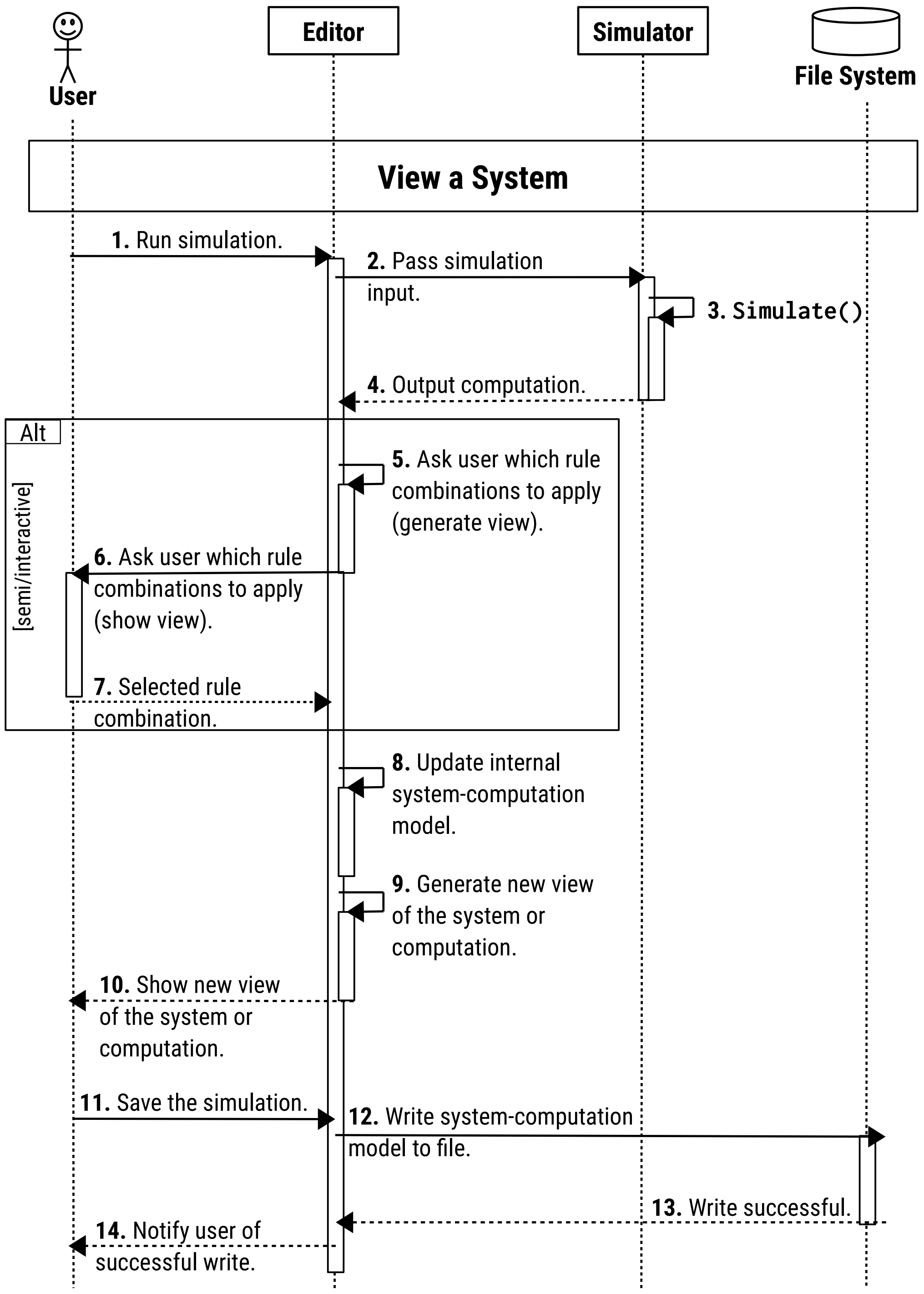

User
Editor
Simulator
File System
View a System
1. Run simulation.
2. Pass simulation input.
3. Simulate()
4. Output computation.
Alt
[semi/interactive]
5. Ask user which rule combinations to apply (generate view).
6. Ask user which rule combinations to apply (show view).
7. Selected rule combination.
8. Update internal system-computation model.
9. Generate new view of the system or computation.
10. Show new view of the system or computation.
11. Save the simulation.
12. Write system-computation model to file.
13. Write successful.
14. Notify user of successful write.

Figure 23: Sequence Diagram for `Simulate a System` Use-case

First, the user needs to open a *.psy* file. The editor will use the procedure for `View a System` use-case. The editor will then pass all simulation input to the P system simulator. The simulation input includes the initial configuration of the system, the rule set and derivation mode of the system, and the simulation parameters like simulation mode and halting criteria. The simulator will then perform the `Simulator()` procedure (to be described in Section ??). The `Simulator()` procedure will return an output *computation*. If the output computation contains more than one new configuration and if the mode is either *semi-interactive* or *interactive*, the editor will ask the user to select which of the valid combination of rules to apply. The editor will then update its internal copy of the computation tree by adding the new configuration(s) and transition(s) to the tree. Then the editor will generate a new view to reflect the changes and it will display this new view to the user. If the user chooses to save the simulation, the internal `computation`/simulation copy of the editor will be written to the *.psy* file.

2.5 Internal Operations of the `Simulate()` Operation

`Simulate()` is the main function of the P system simulator. This function is invoked by the P system editor if the user triggers the simulation of the system.

The `transition()` function is one of the main function used by `Simulate()`. This function takes a `configuration`, a `rule set`, a `derivation mode`, and simulation parameters then generate all possible valid next `configurations` and the `transition` objects related to each of these `configurations`. Figure ?? shows the data flow diagram inside `transition()` function.

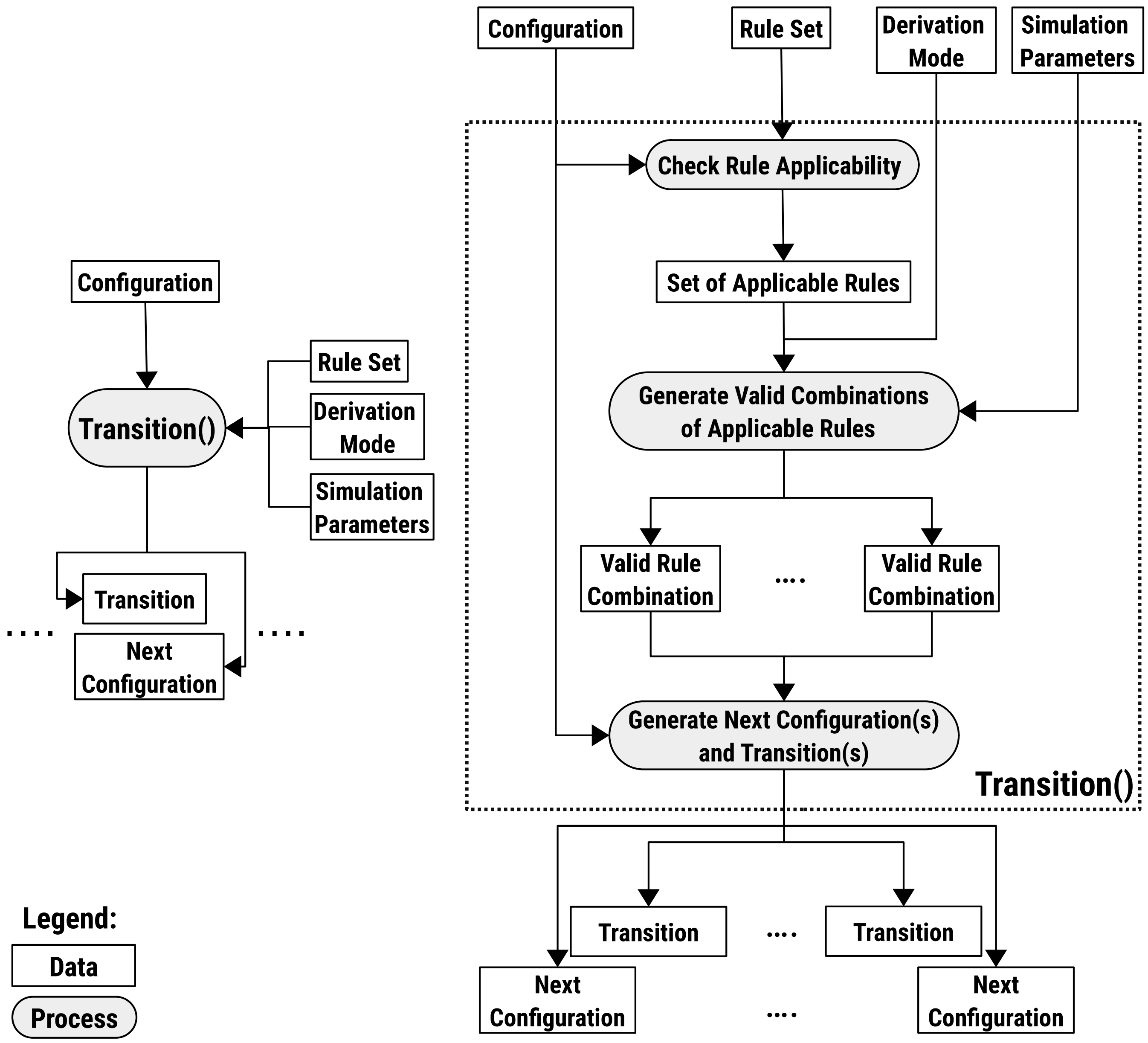


Figure 24: Data Flow of `Transition()` Operation for Generating Next Configuration(s) and Transition(s)

`Transition()` will first check which of the rules are applicable with respect to the given configuration. Using the set of applicable rules, the derivation mode, and the simulation parameters, valid combinations of applicable rules will be generated. Each valid rule combination will be applied to the input configuration resulting to multiple next configurations and the transition objects for each of these new configurations.

Figure ?? shows the data flow of function `Simulate()` while Figure ?? shows `Simulate()`'s activity diagram. `Simulate()` uses the `Transition()` function.

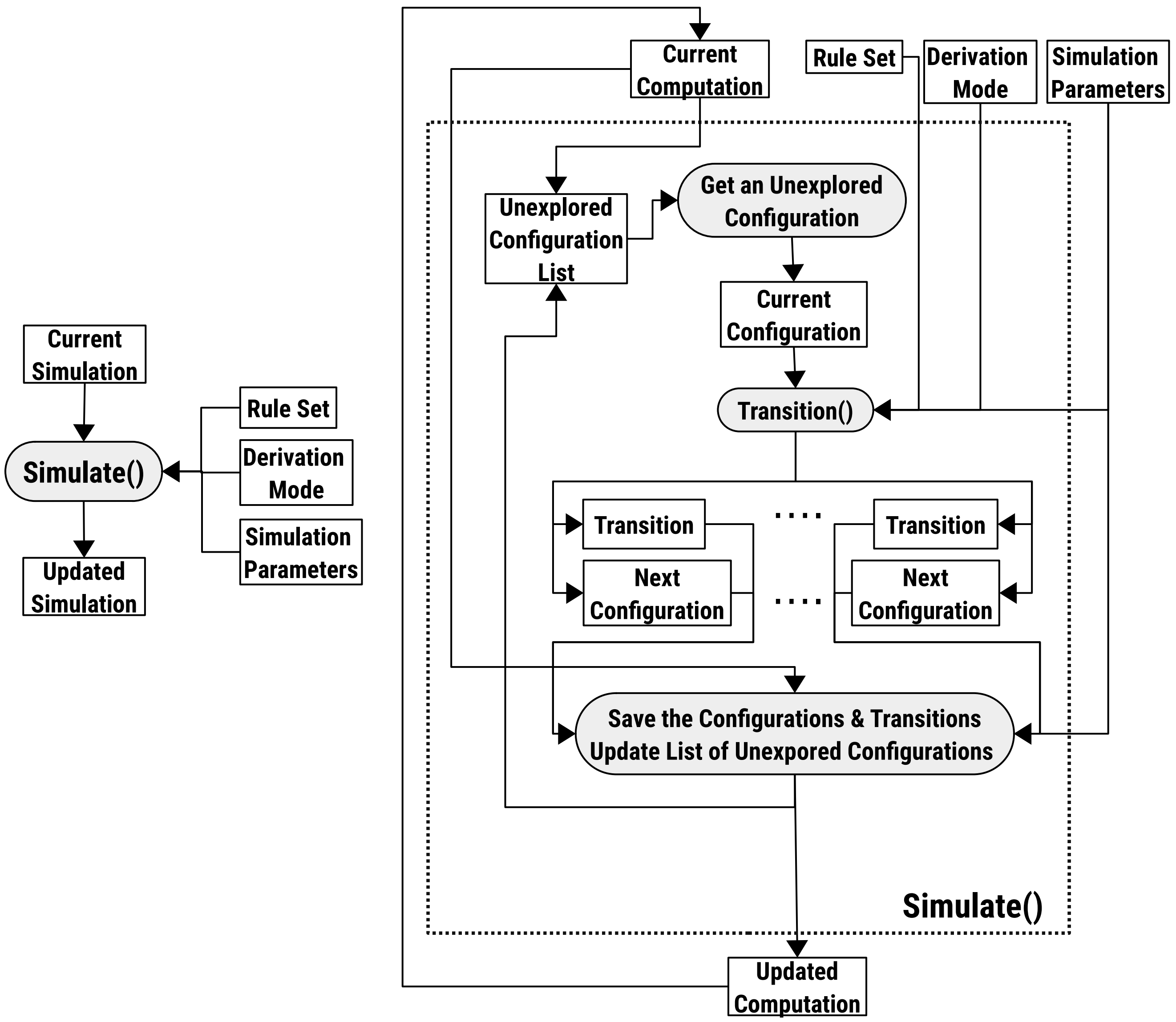


Figure 25: Data Flow of `Simulate()` Operation for Generating Updated Simulation

The input of `Simulate()` are the current computation (specifically the unexplored configurations of the computation tree), the P system's rule set and derivation mode, and the simulation parameters. All unexplored configurations from the input computation tree will be added to the *unexplored configuration list.* The function will get the first unexplored configuration in the list. That configuration, we will call *current configuration*, will be the input to the `Transition()` function along with the rule set, derivation mode, and simulation parameters. The output transition(s) and next configuration(s) will be added to the *unexplored configuration list* and to the copy of the input computation. If the simulation parameters specify a depth-first search simulation, then the new configurations will be inserted at the beginning of the unexplored configuration list. Otherwise, if the simulation should be breadth-first the new configurations will be added at the back of the unexplored configuration list. `Simulate()` will continue generating new configurations and transitions until one of the halt conditions, specified in the simulation parameter, is met.

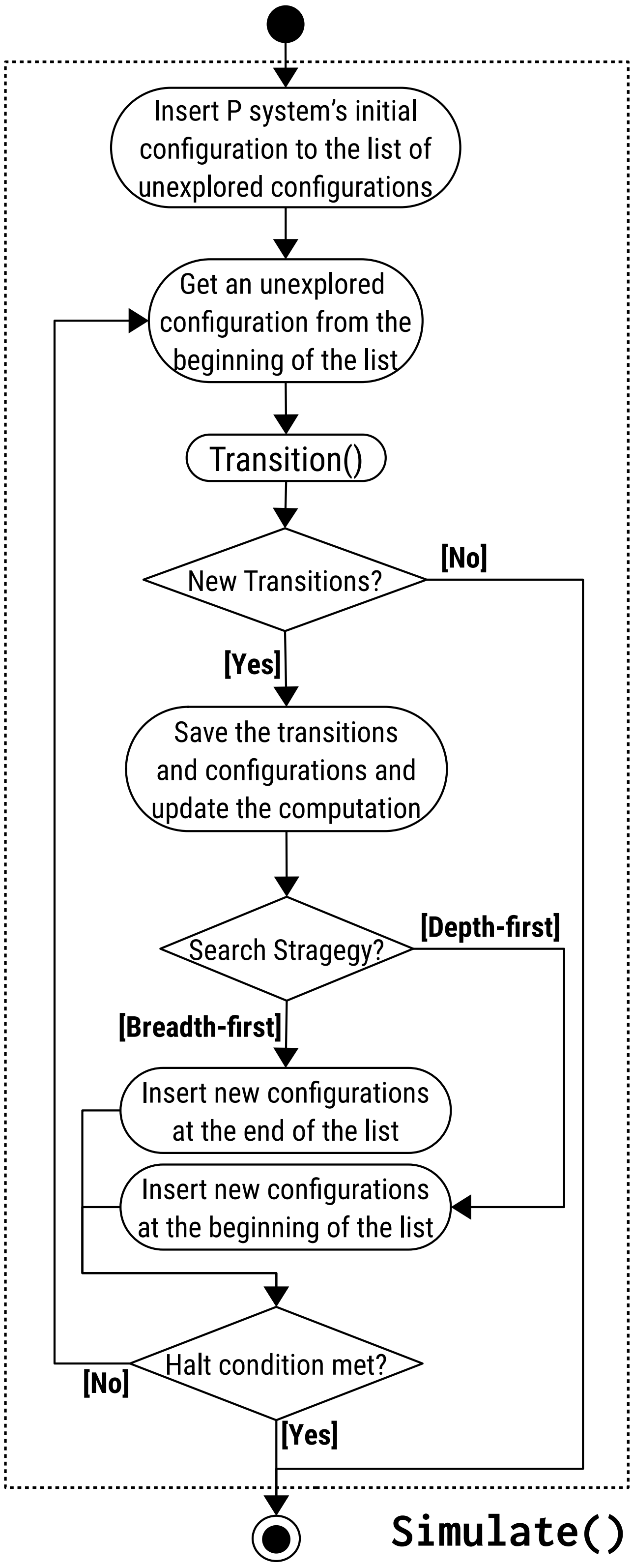

Insert P system's initial configuration to the list of unexplored configurations
Get an unexplored configuration from the beginning of the list
Transition()
New Transitions?
[No]
[Yes]
Save the transitions and configurations and update the computation
Search Stragegy?
[Depth-first]
[Breadth-first]
Insert new configurations at the end of the list
Insert new configurations at the beginning of the list
Halt condition met?
[No]
[Yes]
Simulate()

Figure 26: Activity Diagram for `Simulate()`

2.6 Remarks

This design document shows a high to mid-level design for the P system program. Low-level design, which is very close to implementation, is not included in this document. Examples of low-level designs are implementation designs of procedures/functions inside `Transition()` procedure. e.g. Checking for Rule Applicability, Generating Valid Combination of Rules, Generating Next Configuration(s) (see Figure **??**).