Design Pattern-Based Solutions for General Membrane System Components

Ionuţ Dincă

University of Pitesti
110040 Pitesti, Targu din Vale, nr.1, Romania
E-mail: ionutdinca_24@yahoo.com

Abstract. The software solutions for membrane computing simulators must be in accordance with the fast evolving of this research area. Therefore, we need solutions for designing extensible and highly maintainable software components. The main objective is to find design pattern-based solutions in the process of designing a general membrane systems framework for easily creating P systems simulators for any existing or future model. There are used design patterns as Abstract Factory, Decorator, Observer, and Strategy. The solutions are presented using Unified Modeling Language diagrams (class diagrams for static structures and sequence diagrams for dynamic aspects).

Keywords: extensible component, design pattern, membrane computing, membrane structure, evolution rule.

1 Introduction

Gh. Păun said about membrane computing (in his book from [4]) the following: "The framework is rather general and versatile - compartmental computation, with the separate "processors" communicating in various ways, synchronized, non-deterministic, parallel, dealing with multisets - and this makes a non-limited range of branches, analogies, cooperation with other areas, and applications possible." The starting point of our approach is coming from his assertion.

The implementation of a P system model involves the same basic steps as the construction of any other application. This includes deciding what the application's key requirements are, designing a solution that will satisfy all the requirements, and developing the application. The current membrane systems implementations are mostly designed for a particular family of membrane systems (see [3, 8] for a list of existing applications). A designer would implement a membrane system simulator based on a set of requirements that is considered to be complete. Later during a refactoring phase it would frequently be the case that requirements are expanded or changed. If the underlying structure of the membrane system model needs to change (like a change in topology of the membrane network, syntax or semantics of the evolution rules, data input format, the computation strategy, or any other variation), the designer has to change the current program source code.

The rapid development and enlargement of the research in membrane computing impose designing of an extensible, flexible and highly maintainable software system. The design should be specific to the problem at hand but also general enough to address future problems and requirements. The key to maximizing extensibility and flexibility of a software system lies in anticipating new requirements and changes to existing requirements, and in designing out system so that they can evolve accordingly. A design that doesn't take changes into account risks major redesign in the future.

Challenges such as those mentioned above occur because all components in a membrane system are functionally very tightly coupled. In such domain problem it is not easy to define levels of abstraction and to separate the system functionality cleanly into those layers. Therefore, extensible design solutions for a general membrane systems framework seems to be useful from the perspective mentioned above.

The main objective of the present paper is to show how can be used object oriented design patterns in the process of designing an effective general membrane systems framework, using the conceptual breakdown as requirements. Such a framework could be used to easily create P systems simulators for any existing model. There are used several design patterns (Abstract Factory, Decorator, Observer, Strategy) for solving specific design problems (see [2] for an introduction to design patterns).

2 Designing for a Large Number of Extensions

When trying to design a solution for solving a real world problem, abstraction serves as a way to model the problem. For us, the world is represented by membrane systems, and the model is a collection of objects. Object-oriented systems are made up of objects. The hard part about object-oriented design is decomposing a system into objects. Object-oriented design methodologies favor many different approaches. We can use the textual analysis of the requirements, single out the nouns and verbs, and create corresponding classes and operations. Or we can focus on the collaborations and responsibilities in our system, and create the so called CRC (Class-Responsibilities-Collaborations) cards. Or we can model the real world and translate the objects found during analysis into design (see [1, 7]).

Many objects in a design come from the analysis model. We presents here an abstraction (see Figure 1) obtained by analyzing the main P Systems ingredients and having in mind two of the most essential principles of extensible object oriented design: *Open-Closed Principle* and *Dependency Inversion Principle*.

The first principle says: *Software entities should be open for extensions but closed for modifications*. The addition of new functionality or the modification of an existing one is called an *extension*. An *extensible component* is a component for which the functionality could be extended with minimal modifications of current implementation. Therefore, the main objective is to design components that conform to this principle. The key to achieve extensibility for a software system is in anticipating the possibly new requirements and changes to existing requirements (finding the possible *extension points* of the requirements). The membrane computing area deals with (at least) the following extensions:

– the topology of membrane structure (a tree as in cell like models, a net of membranes as in Tissue models);
– the type of evolution rules (rewriting rules, communication rules, etc.);
– the type of data input (multiset of symbols, strings);
– the type of membranes (electrically charged or neutral, permeable or not, dissolvable or not, divisible or not etc.);
– the type of output (numbers, vector of numbers, strings, multisets, etc.);
– the strategy of applying the rules (maximal parallelism, nondeterministic, sequential, bounded parallelism, priorities among the rules, the probabilistic choice of the rules, etc.);
– the visual representation of the system (venn diagram, as a tree, etc.);
– the type of target commands (weak target (*in*, *out*), strong *in* target (in_j)).

The second principle offers a solution to obtain the open-closed components:

– A. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
– B. *Abstractions should not depend upon details. Details should depend upon abstractions.*

Otherwise stated, don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface usually defined by an abstract class. All classes derived from an abstract class will share its interface. There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes: (1) Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect. (2) Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class defining the interface. For our problem, a membrane only knows about the general evolution rules (abstract class `GeneralRule`), not about the particular rules (symport rule, rewriting rule, and so on) which exist inside the region delimited by it. The immediate advantage is the extensibility: introducing a new particular form of rules don't affect the system previously developed.

The following sections are concerned with design pattern-based solutions for solving different extensibility design problems.

3 General Creator for General Resources

Our software simulators need to use a variety of different features, in order to achieve as much generality as possible. We want to make the systems flexible enough to use resources without having to recode the application each time a new resource is introduced. An effective way to solve this problem is to define

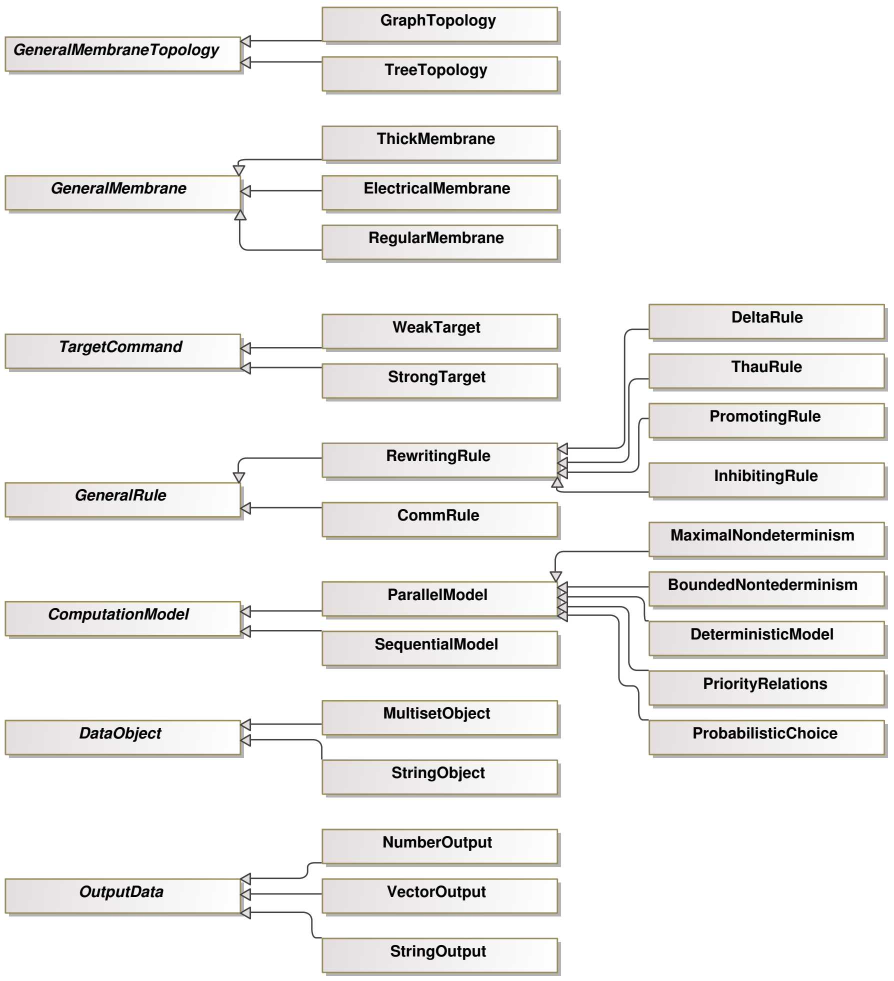

Fig. 1. The abstraction of membrane computing resources.

a general resource creator, the *AbstractFactory* (see Figure 2). The factory has one or more create methods, which can be called to produce generic resources. At runtime, a concrete factory is created and used by application for building a particular class of P Systems (see Figure 3). The `AbstractFactory` class provide a contract for creating families of related or dependent resources without having to specify their concrete classes. This strategy helps to increase the overall extensibility of software systems. The systems can easily integrate new features and resources.

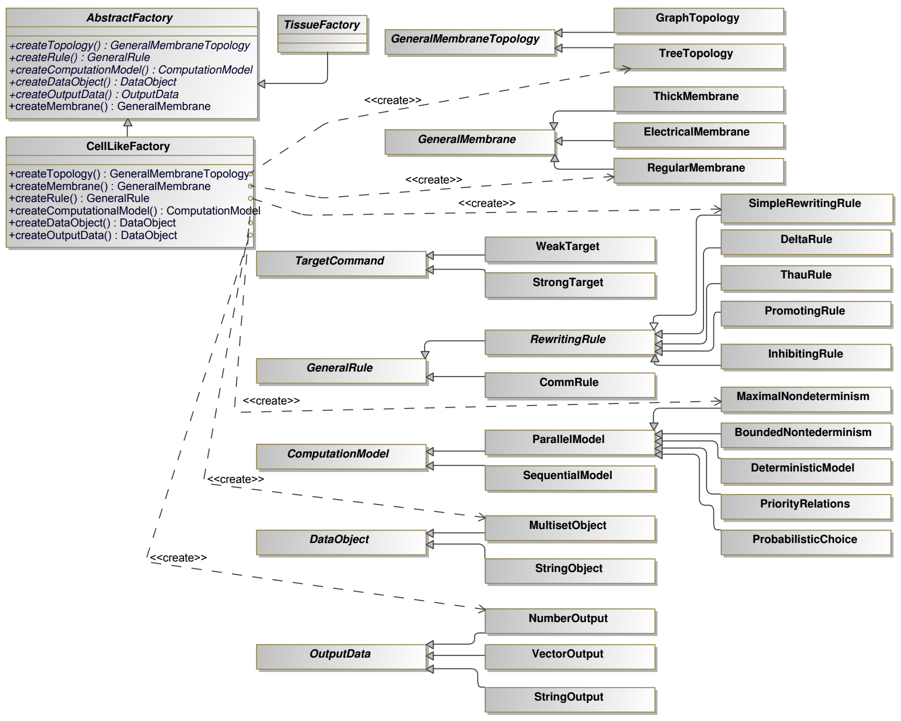

Fig. 2. Factories for creating families of P Systems.

A concrete factory ensures a correct association of resources. For example, a symport/antiport system associates the communication rules with a normal membrane.

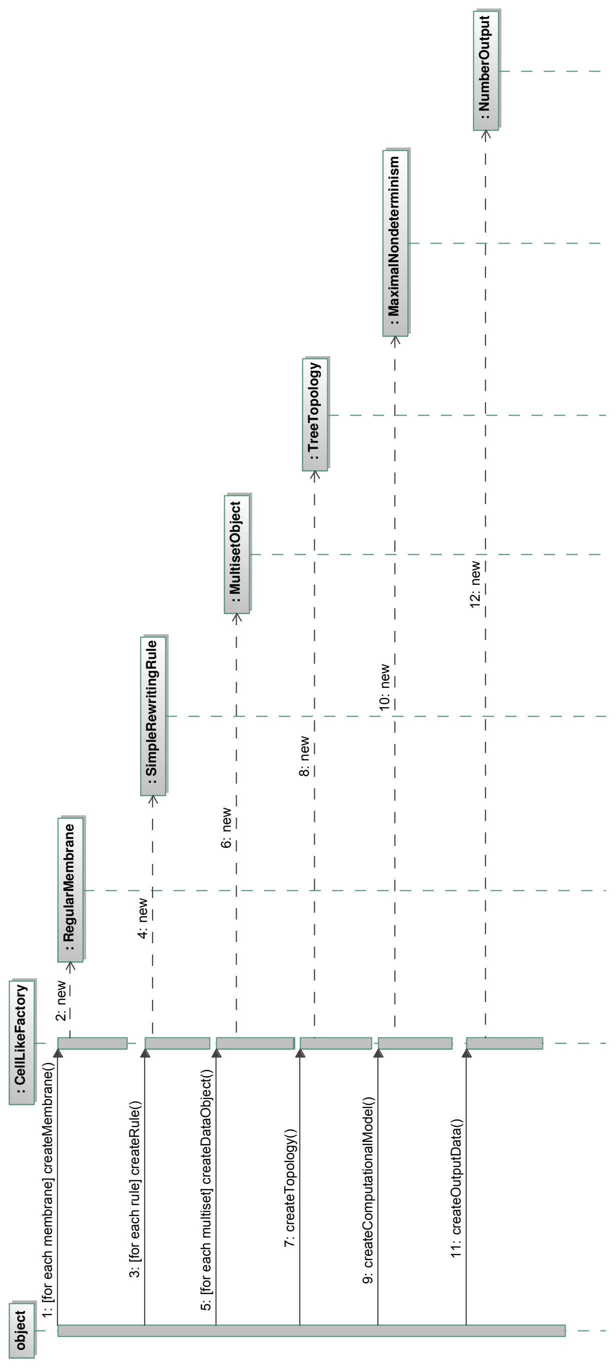

Fig. 3. Collaboration for creating cell-like families of P Systems.

4 Extensible Evolution Engine Component by Extensible Evolution Rules

The evolution rules govern the computations of any membrane system model. The rules and membrane structure topology are both the main ingredients of system. A frequent extension point in membrane computing area consists in changing the syntax and semantic of the evolution rules. Therefore, it is necessary a special attention in the process of designing a general extensible evolution engine component. This section discusses solutions for obtaining component that developers can reconfigure to represent different aspects of execution models with minimal or no code modifications.

4.1 Decorating Rules With New Responsibilities

The functionality of a rewriting rule can be extended, for example, by considering the *membrane dissolving* action. This action is denoted by the symbol δ which may be added to the rules of a system. That is, the rules can be of the form $u \rightarrow v$, or of the form $u \rightarrow v\delta$ (the symbol is appended to the right-side multiset of the rule). The application of the rule $u \rightarrow v\delta$ in a region i means to use $u \rightarrow v$ in the usual way, then to dissolve membrane i;

Not all the rewriting rules are decorated by symbol δ. We want to dynamically add such extensions to individual rules, not to all the rules. One way to add new responsibilities to an object is with inheritance. This is inflexible because the choice of δ-action is made statically. A client can't control how and when to decorate the rewriting rule with the membrane dissolving capability. Decorator Design Patterns could be used as an extensible and flexible solution for the above design problem (see Figure 4). The intention of this pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to inheritance for extending functionality.

The pattern approach is to enclose the usual rewriting rule object in another object that adds the δ-action. The new object is called a *decorator*. The decorator conforms to the interface of the enclosed object so that its presence is transparent to the clients (for us, the membranes of the system). Transparency lets us to add an unlimited number of new responsibilities.

For example, a `SimpleRewritingRule` object implements an usual rule $u \rightarrow v$. The usual rule is extended with dissolving functionality by `DeltaRule` decorator which adds the δ-action. The application of the rule $u \rightarrow v\delta$ is performed by decorator as in Figure 5.

4.2 Different Strategies for Computation Engine

Membrane computations can be simulated by implementing corresponding algorithms which describe the evolution strategy of the systems. The computations are sequences of configurations. In each time unit a transformation of a configuration of the system takes place by applying the rules in each membrane. There

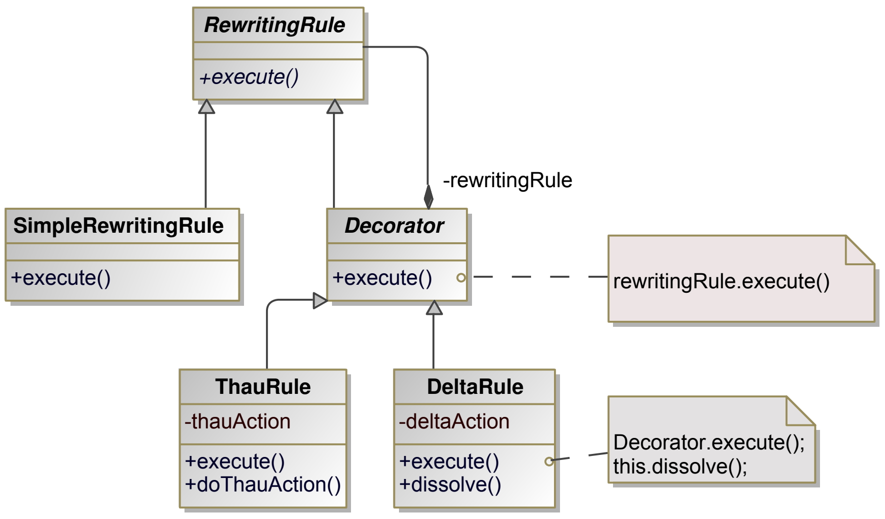

Fig. 4. Extending Rule Functionality by Decorator Pattern.

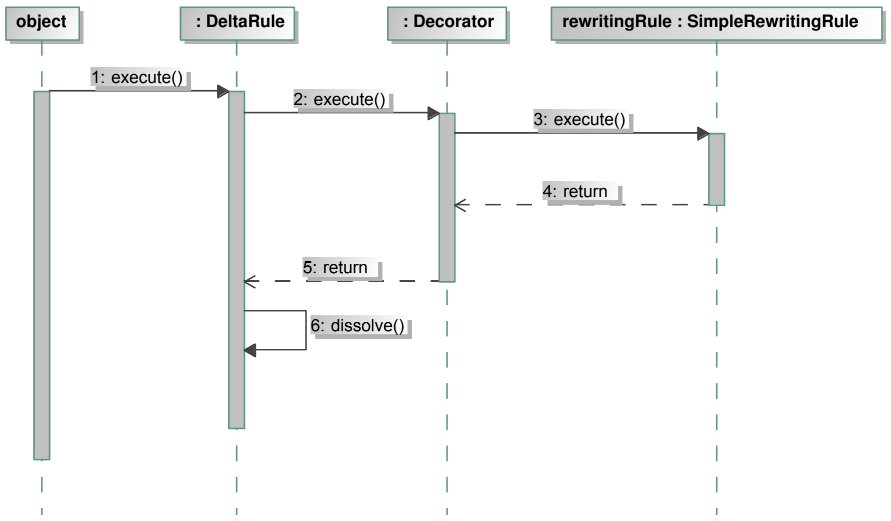

Fig. 5. Collaboration for executing δ-action functionality.

are different strategies of applying the evolution rules. The rules can be used in the maximally parallel manner, sequentially (one rule in the whole system, or in each region), with a bounded parallelism (k rules in the whole system, or in each region), with a minimal parallelism (at least one rule is used in each region where a rule can be used) and so on.

In such cases, *Strategy* pattern helps us to design the applications so that the new strategies of applying the rules can be easily added. The pattern represents the strategies in a separate class hierarchy (see the `ComputationStrategy` class and its subclasses from Figure 6). It provides a way to configure `ComputationEngine` with one of many concrete strategies. Therefore, the computation engine is an extensible component from this point of view.

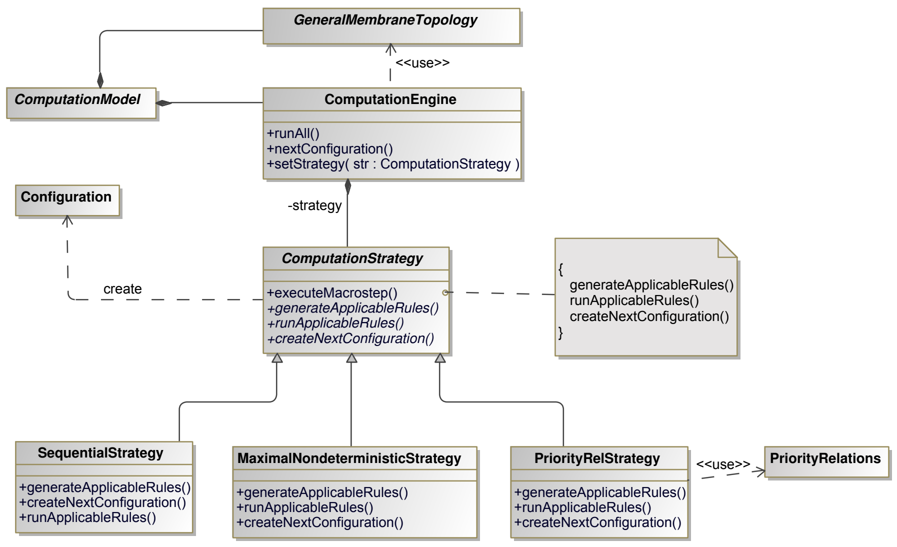
GeneralMembraneTopology
<<use>>
ComputationModel
ComputationEngine
+runAll()
+nextConfiguration()
+setStrategy(str : ComputationStrategy)
Configuration
-strategy
create
ComputationStrategy
+executeMacrostep()
+generateApplicableRules()
+runApplicableRules()
+createNextConfiguration()
{ generateApplicableRules() runApplicableRules() createNextConfiguration() }
SequentialStrategy
+generateApplicableRules()
+createNextConfiguration()
+runApplicableRules()
MaximalNondeterministicStrategy
+generateApplicableRules()
+runApplicableRules()
+createNextConfiguration()
PriorityRelStrategy
+generateApplicableRules()
+runApplicableRules()
+createNextConfiguration()
<<use>>
PriorityRelations

Fig. 6. Extensibility for the computation strategy.

5 Representations for Membrane Structure Topology with Observers

One of the most important aspects to handle in application design is a one-to-many relationship between components. This situation occurs when a group of objects (called `observers`) all depend on the state of a central component (called the `subject`). If any change occurs in the subject, each of the observers needs to be updated. The subject keeps a reference to each of dependants and then merely updates these observers when the need arises. This is fine for a static system with few dependants, as the functionality to update change these objects will not overly complicate the subject and will not change often. There are shortcomings to this approach:

- If more objects need to be added to the list of observers, code changes in the subject requires.
- The code to update observers will eventually start having a negative effect on the subject's performance if the list of observers becomes large.
- All the dependants are active all the time if the update code is added to the subject. It can be very tedious or even impossible to change the code to include only certain dependants at any given stage.

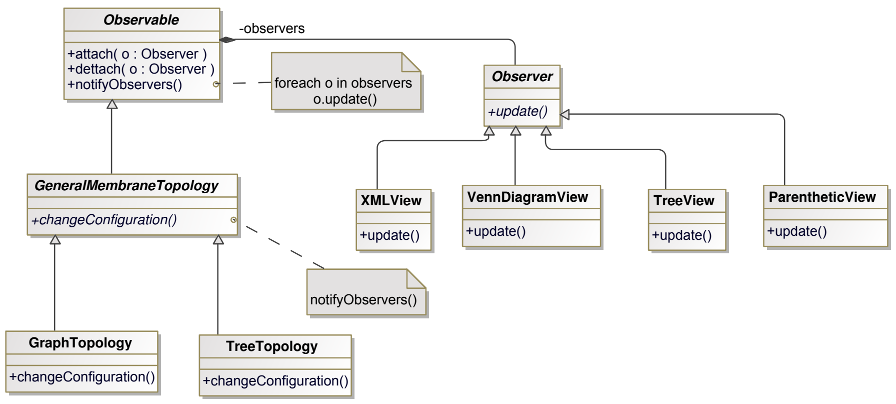

Fig. 7. Observers for topology component.

The Observer pattern (see Figure 7) allows a developer to extract the functionality of updating dependants. The only functionality required in the subject is a `notifyObservers()` method. When any change that might affect dependants occurs in the subject, this method needs to be called as well. This method iterates over a list of registered observers (`XMLView`, `VennDiagarmView`, `TreeView`, `ParentheticView`) and inform each observer of the update and also allows the observer to take the corrective action (see Figure 8). This allows any number of observers to be dynamically registered with the subject, without complicating the subject's code.

The membrane structure topology component is the heart of any P systems implementation. It is suggestive for users to use various representations of this component: as an XML structure, pictorially by a Venn diagram, a tree, a graph or a string of matching parenthesis. Therefore, this different representations are all dependants of the topology subject. The Observer pattern is a flexible and extensible solution for this design problem.

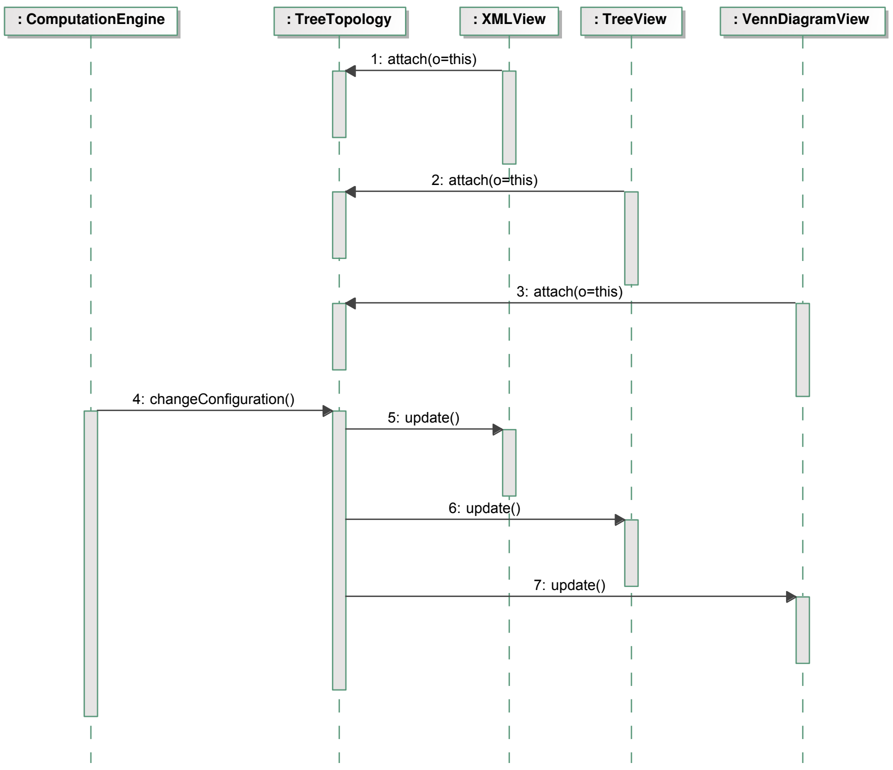

Fig. 8. Collaborations for updating various representations of topology.

6 Conclusions

The flexibility and the versatility of the formalism of membrane computing impose designing extensible software components which are capable to handle future problems and requirements. The software solutions for membrane computing simulators must be in accordance with the fast evolving of this research area. We considered here some of the features of membrane systems and how could be used object oriented design patterns as design solutions. The research from this paper is a beginning. The question of considering pattern-based solutions for other design problems remains to be investigated. For instance, how could be efficiently extend the design for the P systems with active membrane or P systems with a probabilistic way of using the rules. Another important objective of the future research consists in using of a software engineering tool (for example, Rational Rose generates the implementation's skeleton using the UML specification of design) for accelerating the process of implementing the software components in two different programming languages (Java and C#).

References

1. B. E. Wampler, *The Essence of Object Oriented Programming with Java and UML*, Addison Wesley, 2001.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
3. G. Ciobanu, Gh. Paun, M.J. Perez-Jimenez, eds., *Applications of Membrane Computing*, Springer-Verlag, 2006.
4. Gh. Păun, *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.
5. Gh. Paun, G. Rozenberg, A. Salomaa, eds., *The Oxford Handbook of Membrane Computing*, Oxford Univ. Press, 2010.
6. I. Dincă, *Software Design Patterns for Membrane Systems Simulation*, Scientific Bulletin, University of Pitesti, Mathematics and Computer Science Series, No. 12, 2006, ISSN 1453-116x, pp. 53-64.
7. S. Stelting, O. Maassen, *Applied Java Patterns*, Prentice Hall, 2001.
8. The P Systems Web Page: http://ppage.psystems.eu/.