UML Modeling of Finite State Machines and Molecular Machines

Ken Webb

Primordion
Ottawa, Canada
www.primordion.com
ken@primordion.com

Abstract

Biological systems at the molecular level are composed of hierarchically structured objects that continuously interact and influence each other by directly altering each others' composition. The software in many technology systems is hierarchically constructed using objects that pass messages to each other which then trigger transitions in finite state machines. On the surface these appear to be quite different interaction mechanisms. This paper takes a detailed look at these two mechanisms, and, using concepts from UML with models executed using the Xholon framework, argues that these are just points on a continuum. A progression of UML models are developed, as a starting point for a future more comprehensive exploration of this space that encompasses both finite state machines and biological machinery at the molecular level.

Introduction

In this paper I introduce a simple biological control system in which an enzyme continuously transforms glycogen into a more readily usable sugar. The activity of this enzyme is closely regulated by two other enzymes. I then introduce aspects of the Unified Modeling Language (UML), a popular standard used by software developers to graphically design executable models. I also present Xholon, a research tool and runtime environment that can execute UML models.

The bulk of the paper works through a progression of models designed and executed with UML and Xholon. Each is a model of the same biological control system. Model 1 is a symbolic finite state machine (FSM). Model 2 is a more physical simulation using UML FSM objects. Model 3 uses more biologically plausible objects. Model 4 attempts to remove any remaining hidden symbols to produce a system in which each object's behavior is only dependent on what other objects it is composed of or attached to. Model 5 is a version that can be integrated into an existing much larger more complex simulation, to explore how scalable the ideas presented here are. Model 6 is a fully physical realization of the system using Lego blocks, to reinforce the importance of taking a physical perspective.

All models described in this paper, as well as the Xholon framework and a UML viewer, are available as free downloads (Primordion, 2006; No Magic, 2006).

For the latest information on how to obtain these, please visit my symposium site at http://www.primordion.com/symCSE07.html.

This paper uses abbreviations rather than the full biological names. These are summarized in Table 1.

Abbreviation used in this paper	Biological term
Aden	adenosine
Adp	adenosine diphosphate
Atp	adenosine triphosphate
G1P	glucose-1-phosphate
Glc	glucose
Gly	glycogen
GPase	glycogen phosphorylase
GPaseA	glycogen phosphorylase a
GPaseB	glycogen phosphorylase b
GSynthase	glycogen synthase
PGrp	phosphoryl group
PKinase	phosphorylase kinase
PPhosphatase	phosphorylase phosphatase

Table 1: These abbreviations are used throughout the paper.

The Biology System

Biological cells (Alberts, 2000) include numerous examples of active objects whose behaviors are regulated, often at multiple levels. One such system (Becker, *et al*., 1996) involves the enzyme GPase. When you eat, another molecular system, involving the enzyme GSynthase, transforms individual molecules of Glc (sugar) found in your food, into chains of Gly for storage. When your muscle cells need energy, GPase slices long chains of Gly into single units of readily usable Glc in the form of G1P, as shown in Illustration 1.

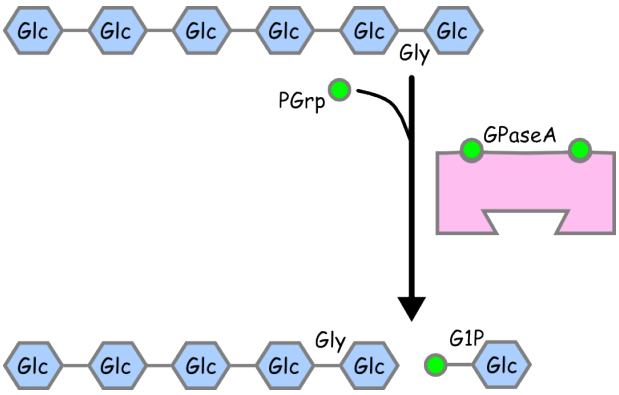

Illustration 1: GPaseA converts long chains of Gly into units of G1P. As part of this biochemical reaction, small PGrp molecules are added on to the end of the Glc units to form G1P.

Cells closely regulate all such enzyme systems. GPase can readily be converted between an inactive *b* form and an active *a* form. As shown in Illustration 2, the enzyme PKinase catalyzes the *b* to *a* reaction, while PPhosphatase catalyzes *a* to *b*. Only when GPase is in the active *a* form does it actually slice Gly chains into individual G1P molecules. In its *b* form it does nothing.

All of these reactions involve physical changes at the molecular level. GPase slices a long chain into individual units by physically binding to the chain at its cleavage site. PKinase subtly changes the shape of GPase by adding two PGrp so that it can bind to Gly chains. This shape change is called a conformational change. PPhosphatase reverses the conformational change by removing the two PGrp. The addition and removal of PGrp molecules are very common processes in biochemistry, called *phosphorylation* and *dephosphorylation*.

PKinase and PPhosphatase are in turn regulated, at least in part, by the quantity of G1P in the cell. GPase itself is also secondarily regulated by the quantity of its G1P product. GPase is part of a much larger system than is described here.

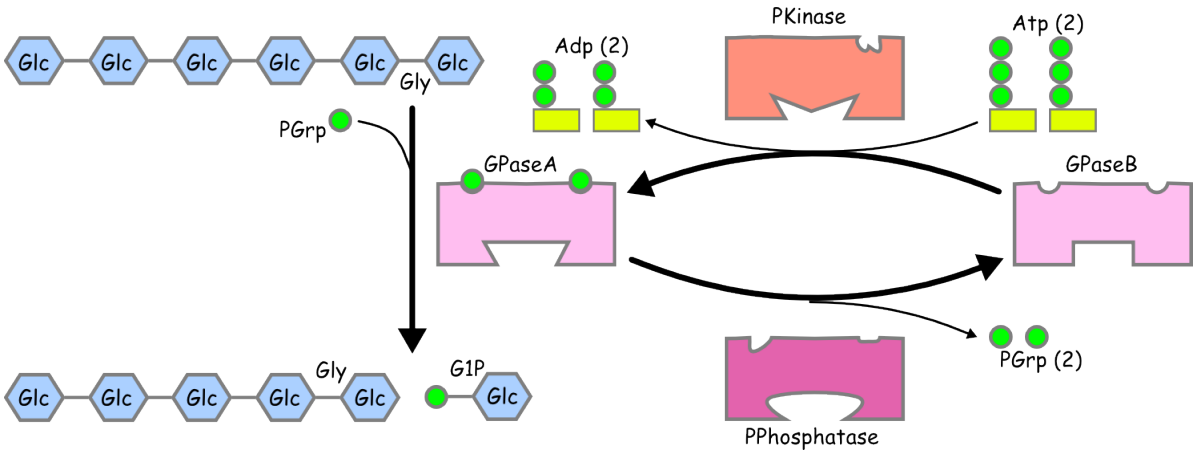

Illustration 2: PKinase and PPhosphatase regulate the activity of the GPase enzyme by transforming it physically between the active GPaseA form and the inactive GPaseB. The process of adding PGrp units to an enzyme is called phosphorylation.

UML 2

The Unified Modeling Language (UML) (OMG, 2006b; Rumbaugh et al., 2005) is a set of software engineering best practices for the development of object-oriented systems. It came into being in the mid-1990's when Grady Booch, Jim Rumbaugh and Ivar Jacobson merged their own software design methodologies, and has since become the de facto industry standard. UML has been standardized by the Object Management Group (OMG), which has recently released version 2 of the standard (UML 2). It is common practice in the software industry to present analysis, design, and implementation models of systems, using the UML common visual notation.

UML 2 offers 13 different diagrams, three of which will be used in this paper. The *class diagram* identifies classes of objects that may appear in the system being developed. These classes may be organized into a hierarchical relationship, in which one class may be a subclass of another one. For example, there could be a class called SmallMolecule with subclasses called Glc, PGrp, and Atp. The *composite structure diagram* (Selic et al., 1994) specifies a composition hierarchy, in which objects are composed of other objects which may in turn be composed of yet other objects, to any arbitrary level of nesting. For example, Gly is composed of a variable number of units of Glc, and Glc is composed of 6 Carbon, 12 Hydrogen and 6 Oxygen atoms. Composite structure diagrams also specify ports that provide the potential for interaction between objects. The *state diagram* (Harel, 1987) specifies the behavior of a class of objects, using states and transitions.

A class diagram specifies the types of parts. A composite structure diagram specifies how the parts are summed up into higher level structures, and, through the use of interactions through ports, how the system is more than just the sum of these parts. A state diagram specifies one way of representing the runtime dynamics as the different parts interact with each other.

Many vendors provide UML tools. The systems described here, and all of the UML diagrams, were developed using MagicDraw UML (No Magic, 2006).

The Systems Modeling Language (SysML) (OMG, 2006a; SysML.org, 2006; Burkhart, 2006) is a subset of UML 2 with extensions to support hardware as well as software. SysML uses *block definition diagrams* in place of class diagrams, *internal block diagrams* in place of composite structure diagrams, and state diagrams. It could be used instead of UML 2 as the basis for this paper. The basic unit of structure in SysML is the *block*, which is very similar to the concept of *xholon* described here.

Xholon

The Xholon (Primordion, 2006) modeling tool and runtime framework is closely aligned with UML 2. As part of its multi-paradigm modeling (Vangheluwe *et al.*, 2002) repertoire, it can execute UML finite state machines (FSM), can model and execute simulations of molecular machines (MM), and can handle hybrids of these two. A complete Xholon application can be generated from a UML 2 model.

A Xholon model represents a UML model as a set of Extensible Markup Language (XML) (W3C, 2006) files, rather than as diagrams. One file contains the class inheritance hierarchy (a tree), another specifies the composite structure (another tree), and a third XML file specifies ports and the potential lateral interactions between objects (a network overlaid on top of the composite structure tree). Every Xholon application, whether in a technology or biology domain, uses the same basic structural mechanisms, all derived from UML 2.

A Xholon model also includes a Java (Sun Microsystems, 2006) class with programming code that specifies the detailed behavior that may be executed at runtime by active objects in the composite structure tree. These activities are often only a few lines of code. They are invoked during transitions between states in an FSM, or at each time step in an MM. Each FSM and MM consists of objects and in turn is contained within a larger object.

Xholon is both a research project, and a tool for developing and executing event-driven applications such as embedded systems (ex: control software for automobiles, microwave ovens, cell phones, etc.). As a research project, it aims to:

- Identify existing formal mechanisms from technology domains, such as finite state machines, petri nets, system dynamics, and many more.
- Identify mechanisms already formalized from the natural world, such as genetic algorithms, genetic programming, neural networks, membrane computing, and more.
- Identify additional mechanisms from the natural world, especially those found in biology, and formalize these.
- Provide a framework and vocabulary that can encompass all of these, to allow software developers to freely use these while developing applications.
- Systematically explore the space that is common to both complex technology and biology systems.

The following are typical steps in developing a Xholon application:

1. Specify a set of object types to be used. Identify parts and types of parts.
2. Specify an inheritance hierarchy for those object types. Organize the part types. Illustration 3 is a UML class diagram showing types and the inheritance relationship between these types. Each of these types will be used in one or more of the various models described in the next section.

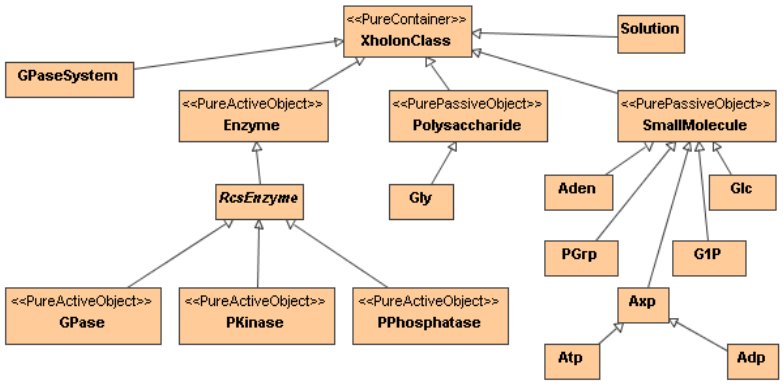

Illustration 3: Types and the inheritance relationships between these types. Some of these object types are just containers for other objects. Some are passive objects that will be acted on. Others are active objects that have defined behavior and that will execute programming code at runtime.

3. Specify a composite structure for instances of those types. The final composite structure tree is "the sum of the parts".
4. Specify ports through which objects can interact with other objects. Ports allow the application to potentially be more than just the sum of its parts.
5. Specify connections between objects using ports. Connections enable the application to fulfill the potential.
6. Define the behavior of objects that will be active at runtime, using finite state machines, molecular mechanisms, or other mechanisms.
7. Write snippets of programming code that will be executed by these active objects at runtime.
8. Iterate.

A progression of models

The next four sections will work through a series of UML models and their Xholon implementations. Each model is a somewhat different representation of the GPase system. The models progress gradually from a traditional symbol-based finite state machine to an equivalent non-symbolic molecular machine.

Note that the use in this paper of the term *molecular machine* differs from the normal biology and nanotechnology usage. Biologists would more likely interpret it to mean a large multi-protein complex such as a polymerase or ribosome. In this paper it loosely means any structured collaboration between molecules that accomplishes some function.

Model 1 – Symbolic finite state machine

Illustration 1 provides the domain vocabulary for the system when the GPase enzyme is in its active *a* form. GPase splits the Gly substrate into units of G1P product. Illustration 2 shows that GPase activity is regulated by two other enzymes, PKinase and PPhosphatase.

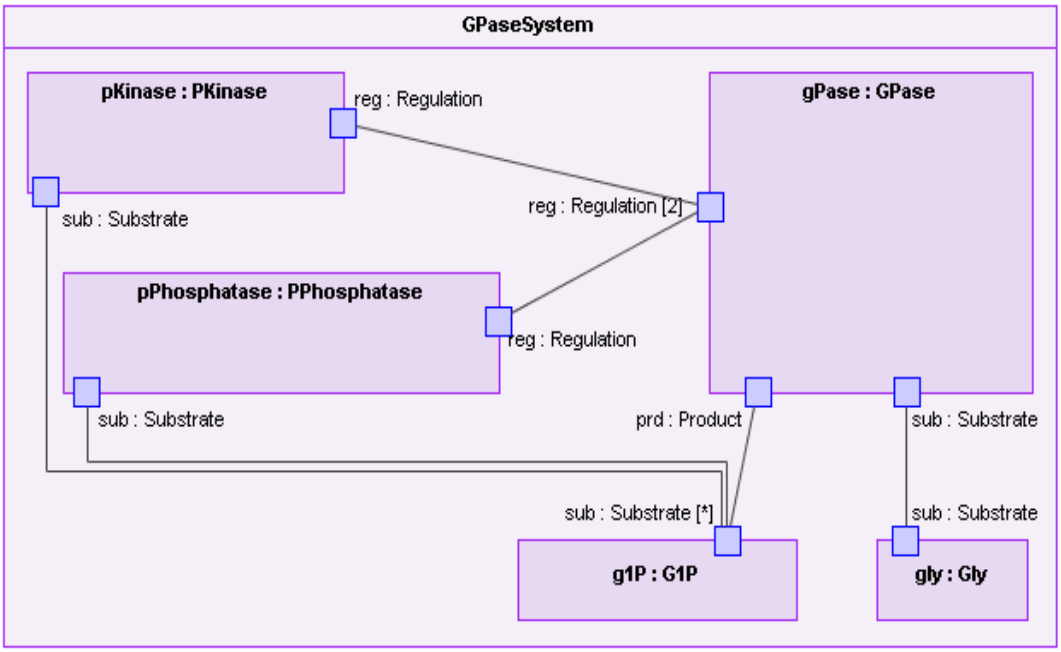

Illustration 4: Model 1 and Model 2 composite structure diagram.

Illustration 4, a UML composite structure diagram, is one way of representing this biological system. There are six types of objects in the figure. The GPaseSystem container class contains five parts. Each part is a single instance of a UML class – gPase:GPase, gly:Gly, g1P:G1P, pKinase:PKinase, and pPhosphatase:PPhosphatase. Each of the five boxes in the figure includes the role name (a lowercase abbreviated name such as g1P), the ":" separator, and the class name. Each object plays a defined role within the context of the GPaseSystem container.

Each part has one or more ports, shown as small squares along the edge of the part. Ports are points of potential interaction with the world outside of the part. Ports are connected using connector lines. A connector between two ports indicates that, at runtime, the parts that own those ports, may either send messages to each other, or one or both may call functions of the other. Ports are labeled with a role name and the name of the port class (ex: sub:Substrate). Ports with multiple instances are shown with brackets. For example, the GPase *reg* port has two instances (reg:Regulation[2]), which allows two different enzymes to interact with it. G1P can interact with a variable number of other parts (sub:Substrate[*]).

The intended meaning of this model is that, at each time step, GPase will split off a molecule from the Gly chain through its *sub* port, and will release it as a separate G1P object through its *prd* port. It implements this by decrementing a variable in Gly and incrementing another variable in G1P. In this model, GPase is an active object, while Gly and G1P are passive.

PKinase and PPhosphatase are two other active objects. Each time step both of these enzymes check the current quantity of G1P, and probabilistically and independently, determine whether or not to regulate the activity of GPase through their *reg* ports.

The behavior of objects can be expressed using UML finite state machines (FSM). Illustration 5 shows a UML FSM representation of the internal behavior of GPase. GPase is always in one of two states – Inactive or Active. Initially, it is inactive, as shown by the arrow exiting the filled-in circle (UML initial pseudostate) at the top left of the figure. If GPase receives an S_ACTIVATE message through its *reg* port while in the Inactive state, it will transition to the Active state. If it receives S_DEACTIVATE, then it will transition back to the Inactive state. In the model, PKinase is responsible for sending S_ACTIVATE messages through its *reg* port, while PPhosphatase sends S_DEACTIVATE. When active, at each time step, GPase executes a UML do activity during which it slices Gly.

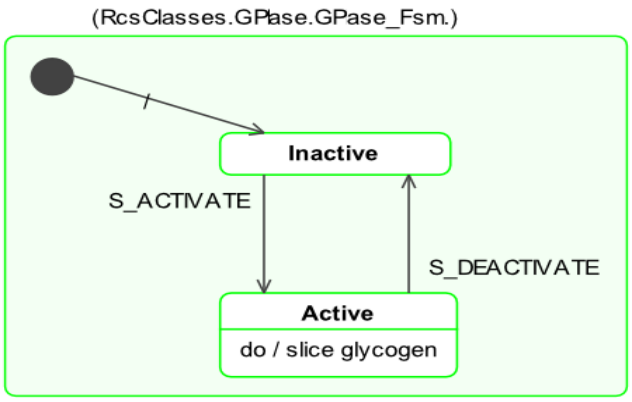

Illustration 5: Model 1 and Model 2 GPase finite state machine (FSM).

The behavior of the PKinase and PPhosphatase active objects can also be modeled as FSMs, but each of these will only contain a single state with an associated activity that they do each time step.

State machines have traditionally been implemented in software by using two variables, both of which can be single integers. In this model, a variable called *state* can take on one of two values, Inactive (0) and Active (1). A second integer variable, called *event*, can be either S_ACTIVATE (100) or S_DEACTIVATE (110), where the S_ stands for SIGNAL_ . The Java programming code in Illustration 6 shows how this FSM has been implemented in Model 1. This code is taken from a manually coded Xholon application.

```
public int state = STATE_INACTIVE;

public void processReceivedMessage(Message msg)
{
  int event = msg.getSignal();
  if (xhClass.hasAncestor("RcsEnzyme")) {
    switch (state) {
    case STATE_INACTIVE:
      switch(event) {
      case S_ACTIVATE:
        state = STATE_ACTIVE;
        break;
      default:
        break;
      }
      break;
    case STATE_ACTIVE:
      switch(event) {
      case S_DEACTIVATE:
        state = STATE_INACTIVE;
        break;
      default:
        break;
      }
      break;
    default:
      break;
    }
  }
}
```

Illustration 6: This traditional implementation of a finite state machine (FSM) uses symbols to represent the current state and event.

Note that *state* and *event* are both symbols. They are simple variables that can take on various values, but they

are not domain-level objects. Symbols are not a very biologically plausible way of representing what actually happens in a biological cell. What we need are simulated objects that have a more physical presence.

I will define *symbol* and *physical* object operationally. Inside a computer, a *symbol* and a *physical* object both require memory. A *symbol* takes on different values by modifying the contents of a memory location. A *physical* object remains the same throughout the simulation, but takes on different relationships with other objects. In model 1, *state* is a symbol, a permanent attribute of GPase. It exists in the computer's memory as a 32 bit (4 byte) integer, that can take on legal values of INACTIVE (0 decimal, 00000000 00000000 00000000 00000000 binary) or ACTIVE (1 decimal, 00000000 00000000 00000000 00000001 binary). In model 3, *PGrp* is a physical object that always has the same type. During part of the simulation, it is contained within an Atp molecule, while at other times it is attached to a GPase molecule. A *symbol-based system* becomes a *physical-based system* when each symbol is replaced by a physical object.

Model 2 – Physical finite state machine

Internally in UML, a FSM is stored as a composite structure. Just as in the structure shown in Illustration 4, a FSM has an outermost object, with other objects nested inside of it. *State*, *transition* and *activity* objects connect with each other, and collaborate at runtime to produce the desired behavior.

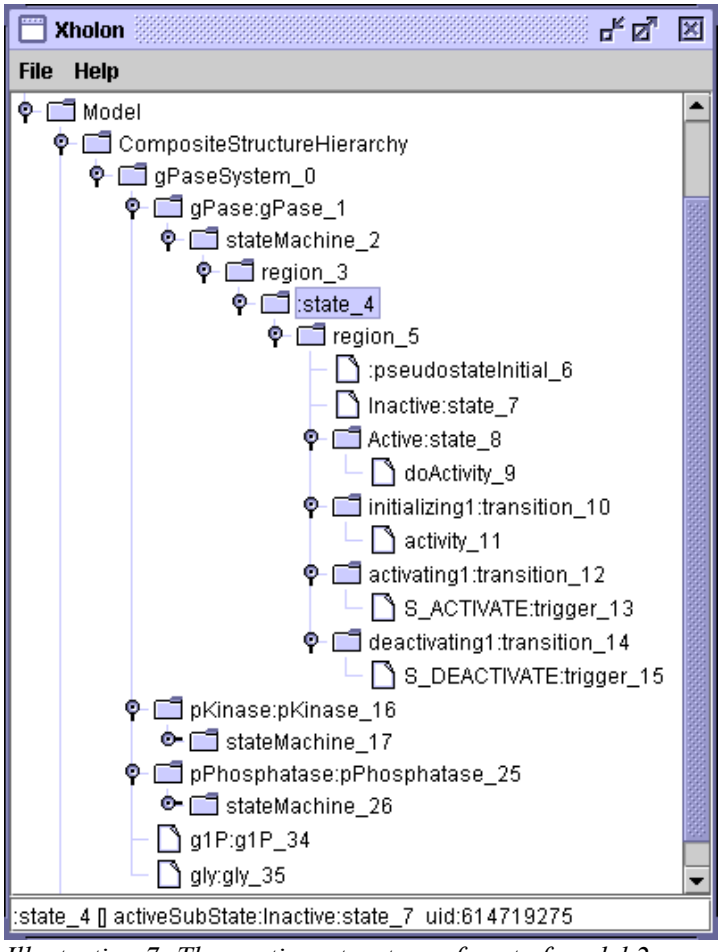

Illustration 7: The runtime structure of part of model 2, as represented within the Xholon runtime framework. Note especially the composite structure of stateMachine_2, the runtime behavior of the active GPase object.

The Xholon code generator reproduces this UML structure when generating the XML composite structure of a Xholon application. This runtime structure is shown in Illustration 7. Compare this with Illustration 5. The stateMachine_2 object is the part of gPase:gPase_1 that implements its behavior. The region_3 and region_5 objects are artifacts of the UML standard that can be ignored here. The state_4 object is the outermost rounded rectangle in Illustration 5.

Instead of symbols, two values of the *state* variable, Model 2 uses two separate objects. This is a more *physical* realization of the model. However, there are still some hidden symbols. Close inspection of the code will show that state_4 has a variable called *activeState* that references either Inactive:state_7 or Active:state_8.

Model 2 is interesting because it represents the behavior of GPase as a collaboration of objects, but a very different set of object types from those used by biology.

Model 3 – Toward a physical molecular machine

Model 3 replaces the FSM representation with a set of objects that are more biologically plausible. Instead of using technology-based objects called *state*, *transition*, *pseudostateInitial*, *trigger* and *activity*, we would like to employ objects closer to the domain with additional names taken from Illustration 2, such as GPaseA, GPaseB, Atp, Adp and PGrp.

Illustration 8 shows this new model. GPase is now initially of type GPaseB, the inactive form. At runtime, the GPase object will be converted back and forth between the inactive *b* form and active *a* form. PKinase and PPhosphatase regulate the current form of GPase, by moving PGrp units between two Atp objects and the GPase object. When GPase contains two PGrp units, it becomes type GPaseA, the active form.

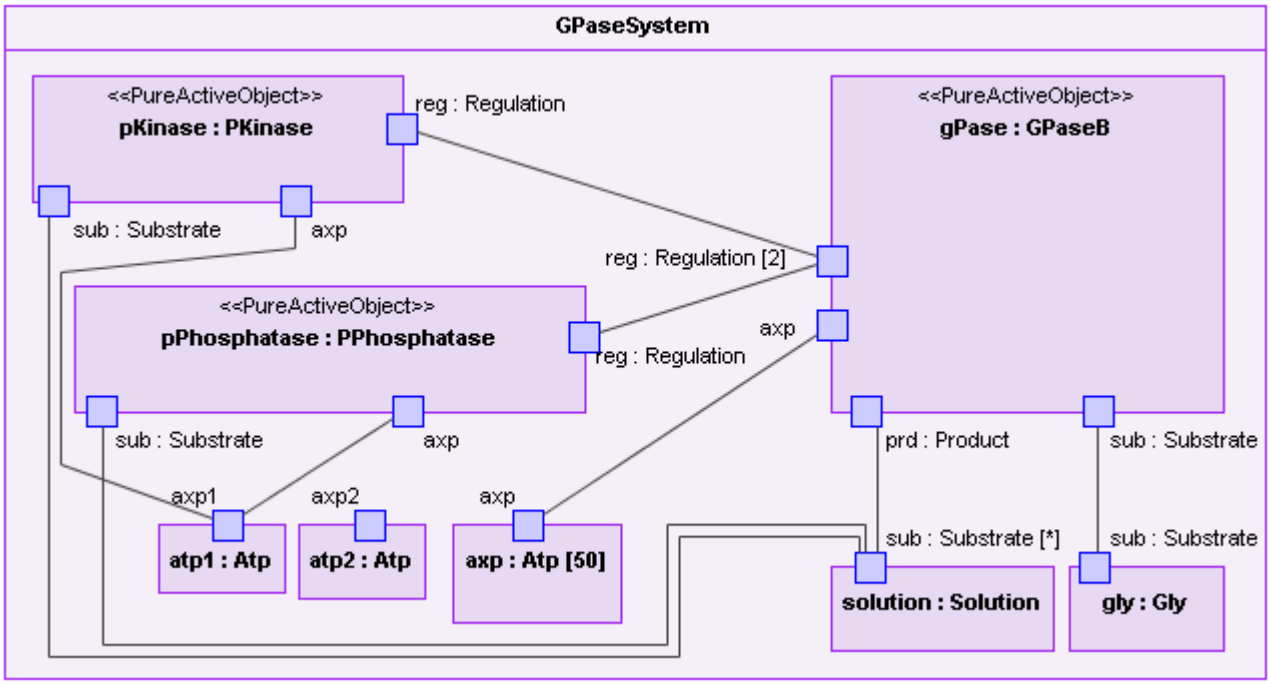

Illustration 8: Model 3 composite structure diagram. Compare this with Illustration 4. Note especially the addition of the Atp objects.

When in its active *a* form, GPase probabilistically splits a Glc from the Gly chain and combines it with a PGrp taken from one of the 50 axp:Atp molecules, to produce a unit of G1P which is added to the solution. All of these actions are done directly by the three active objects. PKinase and PPhosphatase act directly on GPase,

adding and removing PGrp, rather than by sending messages to it's state machine. Illustration 10 shows the initial internal contents of the Gly chain, while Illustration 9 shows the composition of Atp.

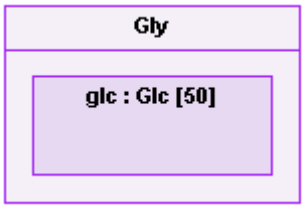

Illustration 10: Gly, in this model, is initially composed of 50 molecules of Glc.

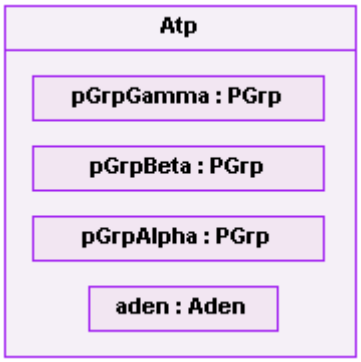

Illustration 9: Atp is composed of three units of PGrp, and one Aden. Adp has a similar structure, but only two units of PGrp.

Model 4 – A more physical molecular machine

While the two are equivalent in overall functionality, model 2 uses finite state machines (FSM) with message passing, while model 3 uses molecular machines (MM) with more direct interaction between objects. Instead of using the technology vocabulary of states and transitions, model 3 uses PGrp, Atp and Adp.

But model 3 is still not as biologically plausible as it could be. In a biological system, an enzyme does not act the way it does because of its current state (ex: Inactive or Active) or because of its current type (ex: GPaseA or GPaseB), but because of what it is composed of or associated with (ex: whether or not it has two PGrp attached to it).

Model 4 only has one type of GPase, one type of Axp (which subsumes Atp and Adp), and one type of glucose (which subsumes Glc and G1P). At runtime, the role played by these objects depends on what they are currently composed of, rather than by which of two types they happen to be at that moment. Model 4 eliminates a hidden symbolic dependency, *type*, which is just an integer in the model, and replaces it with simulated objects.

Model 5 - Integration into a more complex simulation

The GPase system is a small part of a much larger biological system. Because a cell contains many billions of molecules, it's not practical to model each molecule as a separate software object, as is done in model 4.

The GPase processes can be integrated into a larger more complex simulation (Webb & White, 2005; Webb & White, 2006), if the small molecules are converted from individuals into symbols (counters). This has been done to demonstrate that the ideas of composite structure and runtime interaction can be scaled upwards to produce much larger and more complex systems. The resulting models are available at the Xholon project website (Primordion, 2006).

Model 6 – A Lego blocks physical realization

Children's Lego blocks (LEGO Group, 2006) can be useful in determining if a system represents attributes symbolically or physically. It can be difficult for software developers to make this distinction, and it helps to put a system to a test, which is presented here in a tentative form.

A system is *physical*:

- if it initially contains some specified number of blocks (objects) and block composites, each of which has a predesignated type, and if no new blocks can be introduced during the course of the simulation,
- if only the following operations can be performed on blocks (objects), and if there is some predesignated active object block responsible for each such action:
  - a block can be moved from being attached to A to being attached to B,
  - a block cannot gratuitously change its state or quality such as by changing its color, shape or size. For example, a block cannot change color from red to green, it cannot change from a 2 x 3 shape to 2 x 4, and it cannot change from a 2 x 2 size to 4 x 4.
  - an active block can only perform one action at a time, only when it is snapped into place with or adjacent to the block it is interacting with.

Otherwise, a system is *symbolic*.

Illustration 11, drawn using the LDraw (LDraw.org, 2006) tools MLCad and LDview, is roughly the Lego block equivalent of the biological system in Illustration 2 and the UML model of Illustration 8. Illustration 12 shows how the system might appear after a number of simulation steps. Glc is combined with PGrp and moved from the Gly container to the Solution container. PGrp is moved from its association with Aden (Atp and Adp), to an association with GPase.

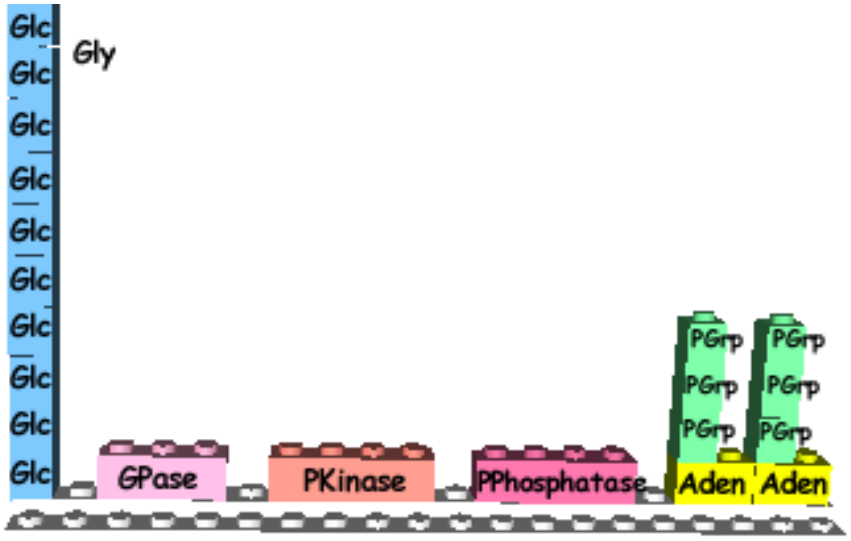

Illustration 11: Initial configuration of a Lego block representation of models 3 and 4. Compare this with Illustration 8 and Illustration 4.

As an example, *state* in model 1 would be a symbolic attribute with two possible values (Inactive, Active), because it would involve some gratuitous change such as changing the color of GPase from red (Inactive) to green (Active) and back again. The only way to legally change its state in a physical model is to attach other blocks to it, such as by adding two PGrp blocks, and this can only be done if there is a block (PKinase) within the simulation that has been predesignated as having this action as part of its behavior.

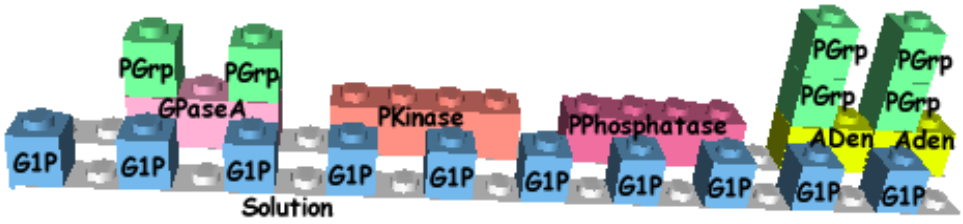

Illustration 12: Lego block representation of models 3 and 4 when the simulation has stopped.

The system could initially have one red 2 x 4 block representing GPaseA, and one green 2 x 4 block representing GPaseB. The red block would continuously slice Gly into G1P, but there would be no legal way for it to be deactivated by turning itself into the green block.

Note that this test is not intended to be a formally defined system with complete and unambiguous rules. It is merely a device to help in reasoning about the types of systems discussed in this paper.

It is possible to perform block simulations by doing the following:

- Specify one type of block or one composition of block types to represent each type of object in the model.
- Specify the precise legal behavior of each block or block composite.
- Select from a box of Lego blocks the required number of blocks to represent the initial configuration. This is the complete set of blocks that can be used during the course of the simulation. Snap blocks together to form any initial composites. Snap all the blocks and block composites into a base.
- Iterate through a sequence of time steps. During each time step, each active object can perform one action in accord with its pre-specified behavior.

Discussion

There is a reduction in the number of symbols in the progression from model 1 to model 4. Symbols are replaced by simulated *physical* objects. Model 6 is the least symbolic of the models in that it uses real physical blocks.

There is also a progression in the use of vocabulary from the biology domain. All models use domain terminology such as GPase and Gly for purely structural elements. Models 1 and 2 use state machine vocabulary for behavioral aspects, while models 3 and 4 use domain vocabulary for both structural and behavioral aspects.

Where models 1 and 2 describe behavior in terms of state and transition, model 3 and especially model 4 implement it using Atp, Adp and PGrp. A PGrp is a small molecule that is the active part of both Atp and Adp, which are themselves relatively small. In a biological cell, Atp is often described as the unit of energy currency. It energizes many of the reactions that continuously occur in our bodies. It would seem appropriate that simulated Atp can somehow take the place of states, transitions and other elements of a finite state machine. Atp in a cell provides energy to enable dynamic behavior, while states and transitions in software also enable dynamic behavior.

Abbott (2006) reviews and discusses some connections between complex systems and systems engineering. In the next few paragraphs, I will briefly touch on some of these as they relate to the models that have been developed in the present paper.

Xholon systems are *agent-based*, in which Xholon active objects are agents. Xholon can model and simulate both artificially engineered and natural systems. The GPase system shows a type of *bricolage* or exaptation, where the Atp molecule, commonly used as a currency of energy in the cell, does extra duty as an agent of conformational change. In models 3 and 4 of the GPase system, the behavior of the system depends on all of its

components, rather than only on the FSM components as in models 1 and 2. It uses *distributed control*. *Energy* is an important part of the models, especially models 3 and 4. Atp is a high-grade energy currency, Gly is stored energy, and Glc is a readily available form of energy. In biological systems, enzymes and other proteins are the direct targets of *evolution*. They are the agents in the GPase system. There are possibly *market* forces at work in this system, given that there are only finite quantities of the resources available and nothing can be created once the simulation starts.

The quantity of Gly and G1P can be graphed over time. The resulting graphs *emerge* out of the interactions of the parts of the system. Each enzyme has its own small set of simple rules. GPase could operate on its own without being regulated. In this case, if GPase starts in an inactive state, then the final emergent graph will be a horizontal line for both Gly and G1P. If GPase is initially active, then the graphs will be straight diagonal lines. When regulated, curved lines emerge. The engineering *requirement* of the GPase system is that it construct this set of curves.

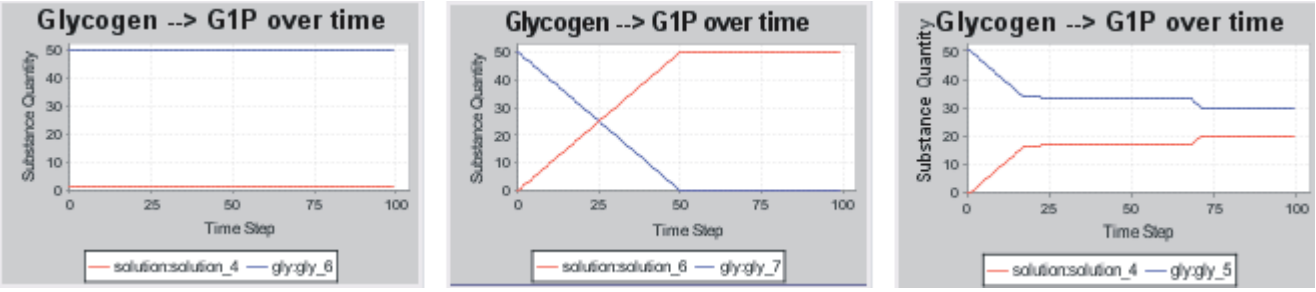

Illustration 13: The activity of GPase over a period of 100 time steps. The blue (top) lines show the size of the Gly chain over time. The red (bottom) lines show the amount of G1P in solution over time. In the leftmost graph, unregulated GPase is initially inactive and therefore there is no change in the levels. In the middle graph, unregulated GPase is initially active and there is therefore constant change until the Gly chain reaches a size of 0. In the rightmost graph, regulated GPase is probabilistically active or inactive, and the two regulating enzymes are in turn regulated by the amount of free G1P currently in solution. The levels therefore change at an ever slower rate.

There is some *stigmergy* in this system, indirect interaction or communication amongst agents through the environment, if the small molecules and Gly are thought of as constituting the environment. There is a feedback loop that involves GPase creating G1P in solution, and PKinase and PPhosphatase basing their actions partly on how much G1P there currently is.

The GPase system is only one of many thousands of biological systems that involve an enzyme interacting with a constellation of small molecules and regulatory enzymes. Within a given biological compartment, many such systems intermesh with each other. Parts of one system can be readily reused by other systems. Novel unexpected systems can emerge out of this. The whole becomes a *system of systems*. Models 3 and 4 can intermesh with each other more readily than models 1 and 2. With a FSM, the behavioral parts are encapsulated and therefore hidden from other parts and systems. They can't be reused in novel ways. The parts of an MM are more distributed and can be reused as parts of other systems.

In a paper that focuses on robotics and neural network controllers, Lipson (2005, p.3) states that “many systems, including robotic systems in particular, are often viewed as comprising two major parts: The morphology, and the controller. The morphology is the physical structure of the system, and the controller is a separate unit that governs the behavior of the morphology ...”. Lipson goes on to say that the distinction in practice is often blurred, but that this distinction is still “pedagogically useful”. Models 1 and 2 in the present paper have distinct morphology and controller parts, while in the later models there is no such clear-cut distinction. The finite state machine controller contained within GPase in model 2, becomes a more distributed set of morphology parts that collaborate less formally to produce the control behavior in model 4. Both approaches can be modeled in UML because morphology (structure) and controller (behavior) parts are both represented in diagrams as nodes with

connectors between nodes.

Future work

Systematically explore the space described in this paper. Re-implement my Protein-Based Neural Network (PBNN) MSc dissertation (Webb, 2004) using Xholon. This is a complex molecular machine system. Implement a technology system using molecular machines, such as an Elevator controller that is currently implemented in Xholon using finite state machines and message passing.

Arguably, biological genes are to UML classes, as proteins are to objects. The present paper has dealt in part with the correspondence between enzymes (all of which are proteins coded for by genes) and xholon active objects (all of which have behavior specified by their xholon class). I would like to make xholon classes, which currently exist in the Xholon tool as real physical objects at runtime, more gene-like. Their expression (creation of object instances) could be controlled as in real gene regulatory networks. Through differential regulation in different compartments of an application, the same set of xholon classes might produce different emergent behaviors, analogous to the different cell types in a biological organism. Xholon classes could be manipulated using evolutionary computation techniques such as genetic algorithms (GA) and genetic programming (GP), to evolve new behaviors. The GA/GP tool ECJ (Luke, 2006) has already been integrated into Xholon as an option. The goal would be for Xholon to merge its present UML sense of class with the biological concept of gene, while still allowing it to run a full range of applications from technology controller to biology simulation.

Conclusion

The same approach can be used to develop technology systems and to model executable biological systems. There is no obvious dividing line between the two types of systems. UML is a flexible collection of mechanisms, derived mostly from the technology side, but also applicable in the biology domain. Biology systems can be modeled using technology mechanisms such as finite state machines. Technology systems can be developed using newer ideas inspired by biology. Xholon is an attempt to gradually bring these two worlds together, to allow for a greater choice of mechanisms when building large complex systems.

Xholon is a tool that can model and execute a wide range of event-driven reactive systems, from both domains. It's a research tool to explore the differences, and even more so the similarities, between what may initially appear to be quite different paradigms.

The Xholon research program is to take ideas or constructs from biology and from technology, and gradually merge these into a single space of modeling concepts and mechanisms that will be recognizable and useful to both biologists and software developers. The particular focus in this paper is on incorporating specific concepts from biochemistry and UML 2 into the same modeling space. The progression of models presented here suggests that there is no fixed demarcation between finite state machines (FSM) and molecular machines (MM). The Xholon modeling tool depends equally on software engineering best practices as captured in UML 2 and on biology for inspiration.

By gradually progressing from a symbolic FSM representation to a more fully *physical* representation, this paper has demonstrated that these paradigms can be encompassed within the same UML-based concepts and tools.

UML provides an adequate set of constructs to enable the modeling of both technology and biology systems, of arbitrary size and complexity. These models, that occupy a continuous space, can be executed using the Xholon runtime framework. The set of UML constructs includes composite structure, ports, message passing and direct access, finite state machines and molecular machines (both of which are specializations of composite structure).

References

Abbott, R., 2006. Complex Systems + Systems Engineering = Complex Systems Engineering. http://arxiv.org/ftp/cs/papers/0603/0603127.pdf

Alberts, B., *et al*, 2000. Molecular Biology of the Cell, 4th ed. Garland Publishing.

Becker, W., Reece, J., Poenie, M., 1996. The World of the Cell. Benjamin/Cummings, Menlo Park.

Burkhart, R., 2006. Modeling System Structure and Dynamics with SysML Blocks. Frontiers in Design & Simulation Research 2006 Georgia Institute of Technology March 16, 2006. http://www.pslm.gatech.edu/events/frontiers2006/proceedings/2006-03-16-Frontiers2006-Burkhart-2.pdf

Harel, D., 1987. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 231-274.

LDraw.org, 2006. Centralized LDraw Resources. http://www.ldraw.org/.

LEGO Group, 2006. LEGO blocks. http://www.lego.com.

Lipson, H., 2005. Evolutionary Robotics and Open-Ended Design Automation. http://ccsl.mae.cornell.edu/papers/Biomimetics05_Lipson.pdf.

Luke, S., et al., 2006. ECJ - A Java-based Evolutionary Computation Research System. http://cs.gmu.edu/~eclab/projects/ecj/.

No Magic, 2006. MagicDraw UML. http://www.magicdraw.com/.

OMG, 2006a. Systems Modeling Language (SysML). http://www.omgsysml.org/.

OMG, 2006b. Unified Modeling Language (UML). http://www.omg.org/uml.

Primordion, 2006. Xholon project. http://www.primordion.com/Xholon/.

Rumbaugh, J., Jacobson, I., Booch, G. (2005). The Unified Modeling Language Reference Manual, 2nd ed. Addison-Wesley, Boston.

Selic, B., Gullekson, G., Ward, P., 1994. Real-time Object-Oriented Modeling. John Wiley & Sons, New York.

Sun Microsystems, 2006. Java programming language. http://java.sun.com/.

SysML.org, 2006. Systems Modeling Language (SysML). http://www.sysml.org/.

Vangheluwe, H., de Lara, J., Mosterman, J, 2002. An introduction to multi-paradigm modelling and simulation. In Fernando Barros and Norbert Giambiasi, editors, Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems), pages 9 - 20, April 2002. Lisboa, Portugal. http://www.cs.mcgill.ca/~hv/publications/02.AIS.campam.pdf

W3C, 2006. Extensible Markup Language (XML). http://www.w3.org/XML/.

Webb, K., 2004. Protein-Based Neural Networks and Their Potential Applications in Building Adaptive Systems. Brighton, UK: University of Sussex, MSc Dissertation. http://www.primordion.com/pub/mscCoursework/Ksw_Dissertation_2n.pdf.

Webb, K., White, T., 2004a. Combining Analysis and Synthesis in a Model of a Biological Cell. Symposium on Applied Computing (SAC 2004), 14-17 March 2004. Nicosia, Cyprus. http://www.primordion.com/pub/publishedPapers/Sac2004_WebbWhite_Bioin.pdf

Webb, K., White, T., 2004b Cell Modeling using Agent-based Formalisms *AAMAS 2004*, New York. http://www.primordion.com/pub/publishedPapers/Aamas2004_WebbWhite.pdf

Webb, K., White, T., 2005. UML as a cell and biochemistry modeling language. BioSystems 80, 283-302. http://www.primordion.com/pub/publishedPapers/BioSys2005_WebbWhite_CogSci2003-05.pdf

Webb, K., White, T., 2006. Cell modeling with reusable agent-based formalisms. Applied Intelligence 24, 169-181. http://www.primordion.com/pub/publishedPapers/Jai2005_WebbWhite.pdf

Wikipedia, 2006. Wikipedia, The Free Encyclopedia, is a good initial source for further information on the following and other topics: Glycogen phosphorylase, Phosphorylation, Unified Modeling Language, Systems Modeling Language, Lego. http://en.wikipedia.org/wiki/Main_Page.