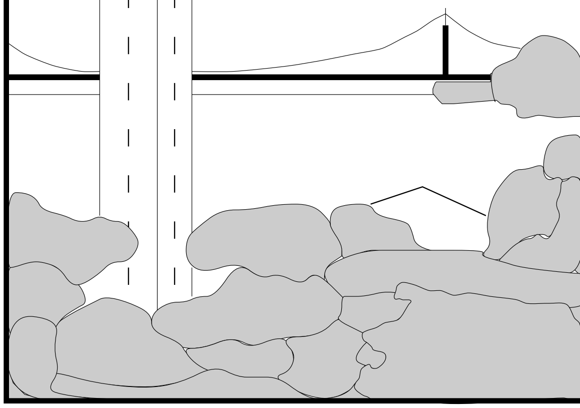
PML: Toward a High-Level Formal Language for Biological Systems

Bor-Yuh Evan Chang *Manu Sridharan*



Report No. UCB/CSD-03-1251

June 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

Documentation of knowledge about biological pathways is often informal and vague, making it difficult to efficiently synthesize the work of others into a holistic understanding of a system. Several researchers have proposed solving this problem by modeling pathways using formal computer languages, which have a precise and consistent semantics. While precise, many of these languages may be too low-level to feasibly model complex pathways. We have developed the Pathway Modeling Language (PML), a high-level language for modeling pathways. PML is based on a biological metaphor of molecules with binding sites and has special constructs for handling compartment changes in pathways. Our preliminary work has shown that PML's language constructs serve as a promising basis for modeling complex pathways in a readable and composable manner.

1 Introduction

Biological processes are highly complex systems of which our understanding is vague at best. Decades of experimentation to understand biological pathways in cells and recent advances in genomics have led to a wealth of information but only in a very fragmented form. Each scientist or group may have vastly different experimental procedures and data representations, making it difficult for a single scientist to holistically understand a system studied by multiple groups. Eventually, someone goes through the arduous task of synthesizing this knowledge, perhaps into a textbook form. Simplifying this process could speed scientific understanding and discovery.

In this paper, we investigate the use of formal computer languages for describing biological pathways. Currently, biological pathways are conveyed through prose or graph-like diagrams with loose semantics. The ambiguity and informality of such representations can make their interpretation error-prone. The use of formal languages in describing pathways would oblige the modeler to make important assumptions explicit, allow him to directly run simulations based on the description (catching obvious errors early), and possibly generate human-readable graphical representations. Moreover, since formal languages have consistent semantics, models written in these languages by different research groups should be more composable than informal models.

We have designed a high-level modeling language for pathways called PML (Pathway Modeling Language). PML is more structured than previously proposed formal languages, leading to more readable and composable models. PML constructs also have a fairly consistent biological metaphor. Finally, we have also developed a novel method for modeling biological compartments.

After discussing the merits and drawbacks of previous work in Section 1.1 and some preliminaries in Section 1.2, we discuss the goals of our work in Section 1.3. We present PML through some example models of biological systems in Section 2, followed by a full presentation of PML and its semantics in Section 3. Then, we give a model of cotranslational translocation on the endoplasmic reticulum (ER) membrane to demonstrate the composability of PML in Section 4. Finally, we discuss the benefits of PML and future work in Section 5.

1.1 Related Work

Traditionally, biological pathways have been abstracted mathematically using chemical kinetic models. In this model, a biological system is modeled by a series of chemical/biochemical equations with variables representing reactants and products. For example, a central notion in biochemistry is the Michaelis-Menten equation for describing enzyme kinetics, which is based on the following abstract model:

$$\mathsf{E} + \mathsf{S} \underset{k_{-1}}{\overset{k}{\rightleftharpoons}} \mathsf{ES} \xrightarrow{k_{\mathrm{cat}}} \mathsf{E} + \mathsf{P} \tag{1}$$

where E stands for the enzyme, S for the substrate, and P for the product; k, k_{-1}, and k_{cat} represent experimentally-determined rate constants for the corresponding reactions. From this, we can derive differential equations to describe enzymes that follow this model.

This approach does capture a useful abstraction and has several advantages including being very well studied, having a considerable theoretical basis, and offering numerous simulation tools and methodologies. However, the variables in chemical kinetic models do not directly correspond to biological entities, as they would for elements in a chemical equation. Typically, we view a system as composed of biological entities that may undergo modification or state-change. This distinction is blurred in the biochemical equation in that a variable more closely corresponds to a biological entity in a *specific state*. In Equation (1), it is implicit that S is modified to become P rather than S leaving and P entering the system. Also, without careful specification, it is unclear how various equations relate to each other, making decomposability very difficult.

McAdams and Shapiro have integrated the traditional biochemical kinetic modeling with circuit diagrams and simulations akin to circuits in electrical engineering for describing the bacteriophage λ lysis-lysogeny decision circuit [MS95]. Though the timescale for switching in electrical versus genetic circuits are vastly different, there are similar timing and delay effects. Analysis of these timing and delay effects are particularly difficult and error-prone in both electrical and genetic circuits. Such a representation makes connections between various equations clearer forcing a more thorough understanding of the assumptions on timing and sequencing of events. Also, some level of abstraction can be achieved by selectively choosing which elements to associate a kinetic model and which to model as simple logic gates.

Pathway databases [Kar01] organize and store information about molecules and their interactions in a symbolic form and provide various ways of querying the database. The goal is to enable scientists to more easily relate pieces of what is known and enable analyses that would otherwise be impossible. Most databases store information in an object-oriented and hierarchical manner, which is akin to how we normally characterize biological entities. However, in many cases, the representation does not precisely describe the dynamic behavior of a pathway, especially when molecules change form and move between compartments. Our work is complimentary to pathway databases in that we seek suitable representations that clearly captures the dynamic behavior of pathways, while databases are currently more suitable for describing static configurations.

Petri nets are probably the most prevalent formalism used to represent the dynamic behavior of biological systems [GP98]. An advantage of Petri nets is that they have an inherent graphical nature similar to common representations in biochemistry. At a high-level, a Petri net consists of *places*, *transitions*, and *tokens*. The state of the system is a *marking* that specifies the number (and possibly type of tokens) on each place. *Input* and *output* functions govern which transitions can fire, which lead to new markings. The biological interpretation typically associates a place with a molecular specie, a token with a molecule, and a transition with a reaction. Another advantage of Petri nets are that stochastic extensions that associate delays and/or rates, which are necessary to faithfully capture some aspects of biological systems, are well-studied. However, Petri nets are still very similar to chemical kinetic models in that each state of a molecular specie is represented by a place (rather than simply the biological entity). Moreover, it seems difficult to have a modular and easily composable representation for complex entities. In fact, Petri nets have been most successful in studying pathways in a rather low-level chemical perspective.

Recently, several researchers have proposed modeling biological pathways as concurrent computational processes utilizing mathematical formalisms, such as process algebras [RSS01, PRSS01, RS03, DL03]. Regev *et al.* have suggested various forms of the π-calculus [Mil99] as a framework for abstracting biological pathways in this manner. The π-calculus was intended to capture the essence of concurrent computation in a minimalistic manner to enable formal study, similar in spirit to the λ-calculus or the Turing machine for sequential computation. It defines a small, yet powerful, language with well-defined semantics (briefly discussed in Section 1.2). A notable distinction of the π-calculus with respect to other process algebras is the notion of *mobility* that enables dynamic rearrangement of the network topology. This is particularly relevant in modeling dynamic changes in interactability of biological entities. This approach combines many of the advantages of the other modeling methodologies. Like Petri nets, the π-calculus has well-defined operational semantics that crisply describe the dynamic behavior of the system and facilitates simulation in a straightforward manner, but like pathway databases, the focus is on describing locally the properties of a biological entity.

1.2 The π-calculus

Since the π-calculus [Mil99] is the basis for the work on which we build [RSS01, PRSS01, RS03] and it underlies our work as well, we present an explanation of the basics here.

The π-calculus is a simple, well-studied formalism for modeling concurrent, communicating processes. *Processes* and *channels* are the fundamental constructs of the π-calculus. A process performs a sequence of send and receive actions on channels. Messages sent on channels are themselves the names of channels, a key feature known as *mobility*. By sending the names of channels around to other processes, one can dynamically change the connections between processes. Mobility makes the π-calculus quite powerful and suitable for modeling a variety of concurrent systems.

The syntax of the monadic π-calculus is as follows:

$$
\begin{array}{lrcll}
\text{Processes} & P, Q & ::= & \mathbf{0} & \text{inert process} \\
 & & | & (P \mid Q) & \text{parallel composition} \\
 & & | & !P & \text{replication} \\
 & & | & \mathsf{new}\ u\ P & \text{channel creation} \\
 & & | & M & \text{sums} \\
\text{Sums} & M, N & ::= & u(v).P & \text{receive } v \text{ on } u \\
 & & | & \overline{u}\langle v\rangle.P & \text{send } v \text{ on } u \\
 & & | & M + N & \text{non-deterministic choice}
\end{array}
$$

The process that does nothing is represented by $\mathbf{0}$. The parallel composition of two processes creates a process in which the *guard* (the leading send or receive action) of each process may be executed. Replication essentially corresponds to an infinite number of copies of the process. The new construct scopes channel names, allowing for some processes to communicate on private channels invisible to other processes. For sums, sends and receives of channels have a straightforward syntax. Non-deterministic choice between sums M and N means that either the guard of M or N will be chosen for execution, and the sum which was not chosen will be consumed. Note, this variant known as the monadic π-calculus specifies that exactly one channel name must be sent in each action; the polyadic π-calculus, which allows for any number of channel names to be sent in an action, has a simple translation to the monadic π-calculus.

The operational semantics of the π-calculus is defined in terms of the the reduction judgment $P \longrightarrow P'$, which is defined inductively as follows:

$$\frac{}{(u(v).P + M) \mid (\overline{u}\langle w\rangle.Q + N) \longrightarrow [w/v]P \mid Q}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad \frac{P \longrightarrow P'}{\mathsf{new}\ u\ P \longrightarrow \mathsf{new}\ u\ P'} \quad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

The first rule says that if one process is sending w on channel u, and another process is receiving v on channel u, then this can be reduced by removing the send/receive action pair and replacing v with w in the receiving process. The second rule allows for reduction on one side of a parallel composition. The third rule allows for reduction inside a scoping construct. The final rule says that P can be reduced to Q if P is congruent to P', P' reduces to Q', and Q' is congruent to Q. The congruence relation $\equiv$ is defined as follows:

$$\begin{array}{rl}
\textbf{Parallel composition:} & P \mid \mathbf{0} \equiv P,\ P \mid Q \equiv Q \mid P,\ P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
\textbf{Scoping:} & P \mid \mathsf{new}\ x\ Q \equiv \mathsf{new}\ x\ (P \mid Q) \text{ if } x \notin \mathrm{fn}(P), \\
& \mathsf{new}\ x\ \mathbf{0} \equiv \mathbf{0},\ \mathsf{new}\ xy\ P \equiv \mathsf{new}\ yx\ P \\
\textbf{Replication:} & !P \equiv P \mid !P
\end{array}$$

where $\mathrm{fn}(P)$ are the free channel names in P. The definitions for parallel composition say that the inert process can be reduced away and that parallel composition is commutative and associative. The first definition for scoping says that if P does not refer to some scoped variable x, we can move a new declaration for x from a process in parallel with P outside of P; this transformation is called *scope extrusion*. The last congruence definition allows a process to be reduced from within a replication operator by making a copy outside the operator.

1.3 Contributions

After investigating prior work in formal abstractions of biological systems as described above, we note an important distinction between the various approaches. Petri net-based and π-calculus-based models introduce a *fixed* and *consistent* formal computational model for describing biological pathways to facilitate sharing, direct simulation, and formal reasoning, while traditional chemical kinetic models seem to impart different computational models for each pathway. Pursuing other formal approaches is tempting. For example, applying the general methodology of rewrite systems seems natural, as we often view a pathway as a series of reactions (such as binding and modification); however, it is unclear how this approach would yield a better computational model for concurrent systems and seems prone to degenerating into developing a machine model on a per pathway basis.

As we are generally concerned about the dynamic behavior of inherently concurrent biological pathways, abstracting them as concurrent systems seems a natural fit. With this view, the π-calculus is a reasonable basis for formally abstracting biological pathways. It is unclear whether any radically different machine model would be much better suited.

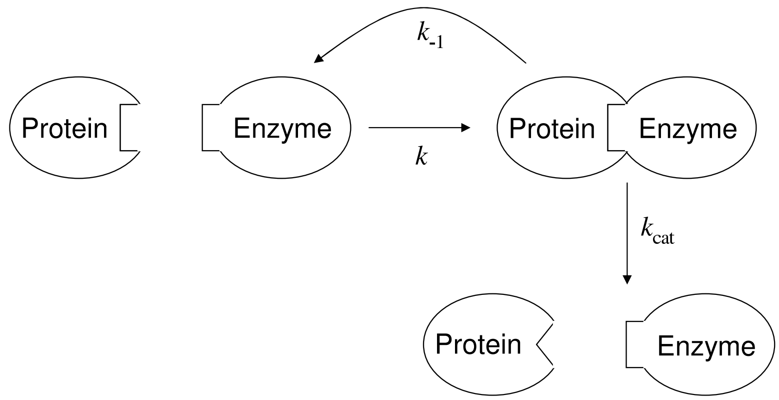


Figure 1: A graphical view of a Michaelis-Menten reaction.

However, there are still a number of issues with deriving models in the π-calculus. While the π-calculus may be an effective language for simulation and formal reasoning, it seems too low-level for direct description of biological pathways. The same parallel can be drawn in programming at high-level source languages versus the language computers interpret. In Regev *et. al* [RSS01, PRSS01, RS03], the structure and modularization of their models is not immediately evident, making it difficult to safely compose them. Compositionality is an important feature for any modeling language, as one should be able to easily take detailed models of different parts of a pathway, written by different researchers, and plug them together. Also, the correspondence between biological objects and π-calculus constructs does not seem to be always consistent. For example, a private channel in the π-calculus is used for compartmentalization and complexing and a global channel to indicate chemical complementarity, but the two get mixed resulting in non-standard match and mismatch checks. In fact, the handling of compartments is awkward, and it is not clear to the present authors if moving molecules between compartments can be reasonably modeled following their methodology.

We have designed PML to address the deficiencies in modeling biological pathways directly in the π-calculus. It is our hope that PML's metaphor of binding sites, principled structuring, and special handling of compartments will make it suitable for writing understandable and composable models of complex biological systems.

2 PML Models

In this section, we present our modeling language, PML, through two example models that illustrate its features.

2.1 Michaelis-Menten Model

PML is largely inspired by an informal graphical style for presenting reactions used in Regev and Shapiro [RS03]. A Michaelis-Menten reaction described in Equation (1) is depicted in this style in Figure 1. Initially, the protein and enzyme have compatible *binding sites*, indicated by the complementary notches in the molecules, allowing them to react. When reacting, the enzyme and protein molecule are “attached” and can therefore perform further private interactions. If the reaction goes forward, the protein is transformed to have a new binding site; it can no longer bind to the enzyme, but it has a new capability to bind to other molecules.

This view of pathways, as reactions that change the binding capabilities of molecules underlies PML. In PML, each dependent set of binding sites is modeled as a *domain* (a molecule can in general consist of multiple independent domains). The enzyme in a Michaelis-Menten reaction is modeled as a domain as follows:

```
1 domain Enzyme = dom ()
2   create (release_substrate, release_product)
3   bind_substrate # put (release_substrate, release_product) ->
4     [release_substrate, release_product]
5   release_substrate # put () -> init
6   release_product # put () -> init
7   init = [bind_substrate]
8 end
```

At any point in time, each domain has a set of active binding sites. Initially, the `Enzyme` domain has the `bind_substrate` site active (specified by the `init` declaration); this is the site through which the enzyme can bind to the protein in Figure 1. The behavior of an enzyme after it binds with a protein is defined by the rule for the `bind_substrate` site (lines 3 and 4). Here, the enzyme “puts” two binding sites, `release_substrate` and `release_product`, that it has created locally and then activates those two binding sites (specified after the ->).Note that the `bind_substrate` site is not active after this transition—an enzyme cannot bind to two proteins simultaneously. Binding sites are created locally to perform reactions with individual molecules, as opposed to an arbitrary molecule in a particular class. Any protein molecule can bind to the `bind_substrate` site of an enzyme, but only proteins that have already bound to the enzyme can react on its `release_substrate` or `release_product` site; the protein molecule learns of these sites since they are “put” by the enzyme in the reaction on `bind_substrate`.

As Figure 1 shows, there are two possible results after the protein and enzyme bind; in either case, the enzyme remains unchanged. This behavior is reflected in the rules for the `release_substrate` and `release_product` sites (lines 5 and 6). After a reaction at either site, the enzyme returns to its `init` state (with only the `bind_substrate` site active). A non-deterministic choice determines whether a reaction occurs on the `release_substrate` site or on the `release_product` site, corresponding to the reverse and forward reactions, respectively. An extension of PML could allow for annotating the rules for these sites with reaction rates to more accurately reflect the probability of each reaction direction, but we do not yet deal with this issue.

The behavior of the protein molecule is encapsulated in its own domain.

```
domain Protein = dom ()
  bind_substrate # get (release_substrate, release_product) ->
    [release_substrate, release_product]
```

```
4     with
5       release_substrate # get () -> init
6       release_product # get () -> [bind_product]
7     end
8     bind_product # ...
9     init = [bind_substrate]
10  end
```

Initially, the protein has its `bind_substrate` site active, allowing it to bind to the enzyme. When it binds to an enzyme on the site, it gets the `release_substrate` and `release_product` sites from the enzyme and then activates those sites. Depending on which of these sites is used for the next reaction, the protein either returns to its initial state (line 5), re-enabling the `bind_substrate` site, or it progresses (line 6), enabling a new `bind_product` site that will allow reactions with new molecules (elided). We declare the rules for the `release_substrate` and `release_product` sites in a `with` construct, as they are bound in the `get` construct.

```
group MichaelisMenten = grp ()
  create (bind_substrate, bind_product)

  domain Enzyme = dom ()
    create (release_substrate, release_product)
    bind_substrate # put (release_substrate, release_product) ->
      [release_substrate, release_product]
    release_substrate # put () -> init
    release_product  # put () -> init
    init = [bind_substrate]
  end

  domain Protein = dom ()
    bind_substrate # get (release_substrate, release_product) ->
      [release_substrate, release_product]
    with
      release_substrate  # get () -> init
      release_product  # get () -> [bind_product]
    end
    bind_product # ...
    init = [bind_substrate]
  end

  compose  <Enzyme(), Protein()>
end
```

Figure 2: Full PML model of a Michaelis-Menten reaction.

We group these two domains together to describe the whole reaction in Figure 2. A `group` construct simply groups together other groups and domains. The `bind_substrate` and `bind_product` sites are created in this group, as they are only relevant to this reaction.

The compose declaration indicates that the initial state of the system is one enzyme molecule and one protein molecule.

Note in Regev and Shapiro [RS03], this reaction is modeled with the substrate and product being separate named entities. As discussed in Section 1.1, this decomposition can be somewhat misleading, as it is not clear whether the substrate is transformed into the product or whether the substrate is consumed and the product is generated. PML's syntactical structuring makes this distinction clear, as the latter case would have been represented by a *spawning* of a new molecule in the `release_product` rule of `Protein`, as follows:

```
release_product # get () -> []<Product()>
```

In this case, the set of active sites following the execution of the rule is empty, indicating that the substrate is consumed and the product is explicitly created.

2.2 Compartments

PML has special syntactic constructs for compartments. Ideally, molecules are specified independently of their compartment membership, so that different compartment memberships can then be employed in different pathways without changing the specification of the molecule. This seems closer to the biology, as the description of a molecule should not change depending on where it is located. The specification style displayed in Regev *et al.* [RSS01] does not appear to have this property; every entity in a pathway must be explicitly parameterized by its compartment membership, then before each reaction, the compartment must be checked before proceeding. Furthermore, while their models allow for restricted communication based on compartments, it is not clear to the present authors how they would model a dynamic change in compartment membership of a molecule, a key property of many pathways.

The compartment constructs of PML are illustrated in Figure 3. Here, we have two simple molecules, `MolA` and `MolB`. When a `MolB` molecule interacts with a `MolA` molecule on the `bind_a` site, its `bind_b` site becomes active allowing it to participate in further reactions (elided). We model a system in which `MolA` and `MolB` begin in different compartments and therefore cannot interact. If `MolA` is transported from its compartment to the compartment with `MolB`, then it can interact with `MolB`. The `Cytosol` compartment contains one `MolA` molecule, and the `ER` compartment contains one `MolB` molecule (declared with compose, just as in groups). The `CytERBridge` molecule bridges the `Cytosol` and `ER` compartment allowing molecules to be transported across the membrane that separates the compartments. Since bridge domains are not contained in a single compartment, we must explicitly declare in which compartment its binding sites are exposed. In this case, its `bind_a` site is exposed to the `Cytosol` compartment. We also give the `bind_a` rule in `CytERBridge` an explicit name `CytERTrans`. Explicit names can be added to any rule for clarity, and they are necessary in the more general case where there are multiple rules for a single binding site. Any molecule binding on the `bind_a` site of a `CytERBridge` molecule (in this case, `MolA`) will be transported from the `Cytosol` compartment to the `ER` compartment, as indicated with the declaration to ER in the `CytERTrans` rule. Our compartment syntax admits a clean separation between molecule behavior and compartment membership and allows for simple modeling of compartment changes through bridges.

```
group CompExample = grp ()
  create (bind_a, bind_b)
  domain MolA = dom ()
    bind_a # put () -> init
    init = [bind_a]
  end
  domain MolB = dom ()
    bind_a # get () -> [bind_b]
    bind_b # ...
    init = [bind_a]
  end
  compartment Cytosol = com ()
    compose <MolA()>
  end
  compartment ER = com ()
    compose <MolB()>
  end
  domain CytERBridge = bridge dom ()
    {CytERTrans} Cytosol bind_a to ER # get () -> init
    init = [CytERTrans]
  end
  compose <ER(),Cytosol(),CytERBridge()>
end
```

Figure 3: Compartments example.

3 Semantics of PML

We define the semantics of PML in terms of the semantics of the π-calculus via two translations: from a PML model to CorePML, a subset of PML that does not have compartment and bridge constructs along with some other simplifications, and from CorePML to the π-calculus. A complete description of the syntax of PML and a formal presentation of the CorePML to π-calculus translation is given in the appendix.

Note that in the absence of a more complete type system, we assume some basic well-formedness conditions on PML models as input to our translation. All references to named entities (rules, domains, groups, *etc.*) must be resolvable. Note, identifiers are lexically-scoped, *i.e.*, they are only accessible within the block they are declared and any nested blocks. The outermost group of a PML model must not take any parameters, as it cannot be instantiated. We disallow recursive instantiations of groups or domains, as this does not seem to make sense biologically—we have not seen recursive biological structures in pathways. We also have some constraints on which site names can be put in a transition rule, explained further in the following section.

3.1 PML to CorePML

CorePML is a subset of PML designed to be explicit and have only essential syntactic constructs, thereby simplifying further translation. CorePML has the following properties:

- All rules in CorePML have explicit rule names.
- All identifiers in a CorePML model are unique.
- There is at most one create declaration in each CorePML domain and group, and this declaration appears first in the domain or group.
- CorePML has no compartment or bridge constructs.

Any well-formed PML model can be transformed to satisfy the first three properties in a straightforward manner. The first property is satisfied by taking any rule without an explicit rule name and making the name of the binding site the explicit rule name, *i.e.*, the binding site name is the implicit name of the rule. Identifiers can be made unique by doing α-renaming where necessary. Finally, once all references to binding site names are checked to see that they respect the create declarations in the original model (*i.e.* they do not reference a binding site name that has not been declared previously), all create declarations in each domain and group can be combined and moved to the top of the domain or group.

The only property of CorePML that is not satisfied through a trivial transformation of any well-formed PML model is the lack of compartment and bridge constructs. We present an algorithm for translating away compartment constructs from PML informally, illustrated for the example in Figure 3. A model output from this translation must satisfy the following two conditions:

1. Two molecules initially in different compartments must not be able to interact with each other.
2. An interaction between a molecule m and a bridge must respect the compartment change declaration in the bridge; after the interaction, m can interact with molecules in the target compartment and cannot interact with molecules in the source compartment.

Together, these properties imply that at any time, molecules in different compartments cannot interact.

We satisfy property 1 with a simple renaming of domains and binding sites. For each domain D composed or spawned in some compartment C, we create a new domain D_C specific to C (we know a domain is spawned in a compartment if another domain initially in that compartment spawns it, or if it is spawned by some domain in a rule that may also cause the spawning domain to move to that compartment). All non-local binding sites b mentioned in D_C (those that are not created in the domain or received in some rule) are renamed b_C to ensure that reactions on that site can only occur with other molecules in C. We also change all spawn constructs to spawn domains particular to the initial compartment. As an example, the copy of the `MolA` molecule from Figure 3 for the `Cytosol` compartment would look like the following:

```
domain MolA_Cytosol = dom ()
  bind_a_Cytosol # put () -> init
  init =[bind_a_Cytosol]
end
```

Since MolB only appears in the ER, there will be no MolB molecules with a bind_a_Cytosol site, and therefore the MolA and MolB molecules in different compartments will not be able to react initially. Note that if the MolA domain was also composed in the ER compartment, we would have created a MolA_ER domain, and these molecules would be able to interact with MolB_ER molecules initially.

Satisfying property 2 is slightly more complicated. First, for each non-bridge domain that can change compartments, we add rules to allow it to interact appropriately in any compartment where it may eventually reside. The set of possible compartments for a domain D is determined as follows. Let a *compartment change site* be a binding site in a bridge domain that causes a compartment change, *e.g.*, the bind_a site in the CytERBridge domain. We first find all the compartment change sites S that D can interact with. Then, the possible compartments for D are the target compartments seen in the bridge rules for each S, plus the initial compartment of D. For MolA, the possible compartments are Cytosol, its initial compartment, and ER, its compartment after interacting with the CytERBridge domain on the bind_a site. Now that we know the set of possible compartments for each domain, we add the necessary rules for that domain to interact in all of those compartments. For the MolA_Cytosol domain, we need to add a rule so that it can interact in the ER domain:

```
domain MolA_Cytosol = dom ()
  bind_a_Cytosol # put () -> init
  bind_a_ER # put () -> init_ER
  ruleset init_ER = [bind_a_ER]
  init = [bind_a_Cytosol]
end
```

We must also create new rule sets for the new compartment (*e.g.* init_ER). In general, any set of rules can be named with this declaration. The init rule set has special meaning, as described earlier in Section 2.

Finally, we must add rules to domains to properly handle the actual compartment change. To do this, we must first ensure that we know exactly when a molecule is interacting with a bridge, so we do not erroneously transfer molecules between compartments. For each compartment change site S in bridge B, we rename S to a fresh name S_B. In our example, we rename the bind_a_Cytosol site in CytERBridge (already renamed once to satisfy property 1) to bind_a_CytERBridge. Now, we add rules for these newly named compartment change sites to the appropriate domains as follows. We make a copy of the rule for the compartment change site as previously named, change the name to match the new compartment change site, and change the right-hand site of the rule to refer to binding sites and domains (if any are spawned) for the new compartment. For example, in MolA_Cytosol, we add the following rule:

```
bind_a_CytERBridge # put () -> init_ER
```

Now, when a MolA_Cytosol domain interacts with the CytERBridge domain, it activates its bind_a_ER site, indicating its compartment change from Cytosol to ER. The final transformed version of our model is seen in Figure 4; no references to compartments or bridges remain. Applying this transformation to remove compartment declarations, along with

```
group CompExample = grp ()
  create (bind_a_Cytosol, bind_a_CytERBridge, bind_a_ER, bind_b_ER)
  domain MolA_Cytosol = dom ()
    bind_a_Cytosol # put () -> init
    bind_a_ER # put () -> init_ER
    bind_a_CytERBridge # put () -> init_ER
    ruleset init_ER = [bind_a_ER]
    init = [bind_a_Cytosol]
  end
  domain MolB_ER = dom ()
    bind_a_ER # get () -> [bind_b_ER]
    bind_b_ER # ...
    init = [bind_a_ER]
  end
  group Cytosol = grp ()
    compose <MolA_Cytosol()>
  end
  group ER = grp ()
    compose <MolB_ER()>
  end
  domain CytERBridge = dom ()
    {CytERTrans} bind_a_CytERBridge # get () -> init
    init = [CytERTrans]
  end
  compose <ER(),Cytosol(),CytERBridge()>
end
```

Figure 4: Final result of translation to eliminate compartments.

the simpler transformations discussed previously, converts any PML model to a CorePML model.

To perform the above transformation, we must restrict the way in which compartment change sites are used. The translation relies on a syntactic analysis being able to identify precisely all potential interactions on compartment change sites. Therefore, compartment change sites cannot be "put" onto other sites in any rule, since the receiver of the compartment change site may also receive other sites through that reaction, and we cannot distinguish these cases syntactically. In our example, no rule can put the `bind_a` site. For similar reasons, bridge domains cannot receive compartment change sites through a reaction. Sites are generally transferred between molecules to facilitate further private reactions and thus are created locally. Therefore, it seems that these restrictions on transferring non-local sites do not significantly hinder expressiveness.

3.2 CorePML to the π-calculus

In this section, we present informally our translation from CorePML to the π-calculus; a formal presentation is given in Section B. At the top-level, a CorePML model consists of several group and domain declarations with a compose statement, corresponding to all of these entities existing simultaneously in the pathway. In the π-calculus, this behavior corresponds to a parallel composition of the translations of the groups and domains. For example, the Michaelis-Menten model in Figure 2 would be translated to the π-calculus as $[\![\texttt{Enzyme}]\!] \mid [\![\texttt{Protein}]\!]$, where $[\![\texttt{Enzyme}]\!]$ and $[\![\texttt{Protein}]\!]$ are the π-calculus translations of the `Enzyme` and `Protein` domains, respectively.

For translating domains, we adopt the strategy of uniformly making each rule a "function". For example, the π-calculus term for the `bind_substrate` rule of the `Enzyme` domain in Figure 2 is

$$!(bsToken().\overline{bind\_substrate}\langle release\_substrate, release\_product\rangle.\overline{rsToken}\langle\rangle + \overline{rpToken}\langle\rangle)$$

We create a token channel for each function, *e.g.* $bsToken$, to be used for calling a function; a call is performed by sending on the token channel, and the function does not perform its action until receiving on the token channel. We translate binding sites as π-calculus channels, `put` actions as π-calculus sends, and `get` actions as π-calculus receives; we see above the `put` on the `bind_substrate` site translated in this example. After performing its `put` or `get` action, the rule function enables the new set of binding sites with the choice operator that non-deterministically calls one of the newly enabled site's rule function. In this example, we send on either the *rsToken* channel or the *rpToken* channel, corresponding to calling either the rule function for `release_substrate` or `release_product`. Finally, we encapsulate the entire rule function in the π-calculus replication operator; this allows the function to be called any number of times (*i.e.* an unrestricted function). One nice aspect of this translation is that it handles both recursive and non-recursive references to rules uniformly.

The translation of a domain is a parallel composition of all its rule functions with a non-deterministic choice of sends on the token channels corresponding to rules in the `init` set. Here is the full translation of the `Enzyme` domain:

$$\begin{array}{ll} & !(bsToken().\overline{bind\_substrate}\langle release\_substrate, release\_product\rangle.\overline{rsToken}\langle\rangle + \overline{rpToken}\langle\rangle) \\ \mid & !(rsToken().\overline{release\_substrate}\langle\rangle.\overline{bsToken}\langle\rangle) \\ \mid & !(rpToken().\overline{release\_product}\langle\rangle.\overline{bsToken}\langle\rangle) \\ \mid & \overline{bsToken}\langle\rangle \end{array}$$

The first three lines are the rule functions for the `bind_substrate`, `release_substrate`, and `release_product` rules, and the final line corresponds to the `init` declaration, calling the `bind_substrate` rule function. Note that we ignore handling the scoping of channel names properly here; this issue and other details are handled fully in the formal presentation.

4 Example: Cotranslational Translocation

In this section, we present an abstract model of the cotranslational translocation of a general secretory protein across the ER membrane [LBZ+99, page 698] to test the effectiveness of PML in capturing some important aspects of compartmentalization, such as transport across a membrane. We then modify this model to describe the synthesis and insertion into the ER membrane of the GLUT1 glucose transporter [LBZ+99, page 706] to emphasize the few changes that need to be made.

4.1 Targeting the ER Lumen

In this model, an arbitrary protein is translated by a ribosome and transferred from the cytosol of a cell into the lumen of the endoplasmic reticulum (ER) cotranslationally. In our abstraction, a *ribosome* begins translating some *mRNA* exposing a *signal sequence*. The signal sequence attracts an *SRP* (signal recognition particle) that binds to the signal sequence, suspending translation. The SRP and *SRP receptor* (located on the ER membrane) interaction drags the ribosome complex close to the membrane. The signal sequence then interacts with the *translocon* gate, opening it as SRP disassociates from the complex. Translation resumes into the translocon pore, transporting the *growing polypeptide* into ER lumen. In the ER lumen, a *signal peptidase* cleaves the signal sequence, and then *Hsc70* chaperones bind to the growing polypeptide, facilitating the proper transport and folding of the nascent chain.

The mRNA is abstracted as a domain with a single site that initiates translation. Degradation of mRNA is ignored but could be introduced as another site.

```
domain mrna = dom ()
  translate # get (done) -> done
  with
    done # get () -> translate
  end
  init = [translate]
end
```

Upon reacting on the `translate` site, the mRNA instance gets a `done` site that is used by the bound ribosome to signal when translation has completed.

An abstract ribosome in this model can only interact with an mRNA to begin translation (indicated by having one global site `translate`), which instantiates/creates a `growingPolypeptide` with two private sites for signaling completion and suspension.

```
domain ribosome = dom ()
  create (mrnaDone, polypeptideDone, polypeptideSuspend)

  translate # put (mrnaDone) -> [mrnaDone,polypeptideSuspend]
    <growingPolypeptide(polypeptideDone,polypeptideSuspend)>

  mrnaDone # put () -> [polypeptideDone]
  polypeptideDone # put () -> [translate,polypeptideDone]

  polypeptideSuspend # get (restart) -> [restart]
  with
    restart # get () -> [mrnaDone,polypeptideSuspend]
  end

  init = [translate]
end
```

Upon interacting with an mRNA, a private site `mrnaDone` is exchanged between these particular instances of the ribosome and the mRNA for indicating completion of translation.

The growing polypeptide is an abstraction for the polypeptide while it is being translated that interacts with several entities.

```
domain growingPolypeptide = dom (done,suspend)
  (* Translation may complete at any time. *)
  {badDone}  done # get () -> []<badProtein()>
  {goodDone} done # get () -> []<goodProtein()>

  (* SRP interaction with the signal sequence causes suspension. *)
  srpSigseq # get (sigseqBound) -> [suspend]
  with
    sigseqBound # get () -> [restart]
  end

  create (restart)
  suspend # put (restart) -> [sigseqBound]
  restart # put () -> [badDone, transloconSigseq, cleaveSigseq]

  (* Translocon interaction with the signal sequence. *)
  transloconSigseq # get (transloconBound) -> [transloconBound]
  with
    transloconBound # put (done) -> [badDone,cleaveSigseq]
  end

  (* Signal sequence cleavage. *)
  cleaveSigseq # get () -> [hsc70Polypeptide, badDone]

  (* Hsc70 chaperone interaction. *)
  hsc70Polypeptide # get () -> [goodDone]

  init = [badDone, srpSigseq, transloconSigseq, cleaveSigseq]
end
```

In developing this model, we found several aspects of its behavior unclear or incompletely specified from the prose description; these ambiguities were not apparent until we pursued this formalization. For example, what happens if the translation completes before SRP binds? It may be an implicit assumption that SRP is present at such high concentration that there is negligible probability of this happening. We chose to model that translation could finish at anytime, but we distinguish the formation of a "good" protein versus a "bad" protein depending on whether some required interactions take place (such as Hsc70 binding) before translation completes. Another ambiguity from the prose description is whether or not the translocon can bind to the signal sequence without SRP. Indeed, SRP is *not* essential for this pathway to function correctly [HB00]. This illustrates that writing formal models can lead to asking important questions about the functioning of a system and finding potential deficiencies in existing knowledge to explore. Finally, note that though in our description that the signal sequence cleavage and Hsc70 chaperone interaction do not occur until the polypeptide reaches the ER lumen, there is no explicit mention of these conditions; they instead will be enforced when we instantiate a `growingPolypeptide` in a particular compartment. This is fairly close to biology in that we would expect that a functional signal peptidase could cleave such a signal sequence *in vitro*, meaning it is the compartmentalization that prevents the interaction, not the chemical complementarity.

The SRP and SRP receptor have some straightforward interactions.

```
domain srpreceptor = dom ()
  srpSrpreceptor # get () -> init
  init = [srpSrpreceptor]
end

domain srp = dom ()
  create (sigseqBound)
  srpSigseq # put (sigseqBound) -> [srpSrpreceptor]
  srpSrpreceptor # put () -> [sigseqNear]
  sigseqNear # put () -> [sigseqBound]
  sigseqBound # put () -> init
  init = [srpSigseq]
end
```

The signal sequence and translocon interaction does not in fact require SRP but in reality is required to make the probability of interaction feasible. While we do not currently support any stochastic modeling, it should be easy to incorporate annotations associated with any set of active binding sites. In our model, the transition involving `sigseqNear` would signal the translocon that a signal sequence is near, thereby increasing the probability of interaction (if we had the ability to specify this).

The translocon is a membrane protein potentially with sites on either the cytosol or ER lumen side. We thus indicate it as a bridge and must qualify in which compartment its sites are.

```
domain translocon = bridge dom ()
  create (sigseqBound)

  (* Transfer the other molecule to the ER. *)
  Cytosol transloconSigseq to ER # put (sigseqBound) ->
    [sigseqBound]

  ER sigseqBound # get (done) -> [done]
  with
    Cytosol done # get () -> init
  end

  (* Presumably increase the probability of binding the
     signal sequence. *)
  Cytosol sigseqNear # get () -> [transloconSigseq]

  init = [transloconSigseq,sigseqNear]
end
```

Whatever interacts with the translocon on the `sigseqBind` site in the cytosol is transferred into the ER. This ensures that the knowledge of compartmentalization is confined to the compartment declarations and bridge declarations. As alluded to in the SRP representation, after the translocon gets the "signal sequence near" indication (*i.e.* interaction on the `sigseqNear` site), the only possible next reaction is to the bind the signal sequence with presumably higher probability.

The signal peptidase and Hsc70 abstractions are fairly simple. After interacting with the polypeptide, it simply returns to the initial state ready to modify/chaperone the next polypeptide.

```
domain signalpeptidase = dom ()
  cleaveSigseq # put () -> init
  init = [cleaveSigseq]
end

domain hsc70 = dom ()
  hsc70Polypeptide # put () -> init
  init = [polypeptideBind]
end
```

Note, we have modeled that one `hsc70` binding to the nascent chain is sufficient to produce a "good" protein. We can view this as the collective of Hsc70 chaperones required to yield the proper folding.

Finally, we place the domains in the proper compartments and for convenience, group everything together.

```
group ERCotranslationalTranslocation = grp ()
  create (transloconSigseq, sigseqNear, cleaveSigseq, hsc70Polypeptide)

  compartment Cytosol = com ()
    create (translate, srpSrpreceptor, srpSigseq)
    domain mrna = dom ... end
    domain ribosome = dom ... end
    domain growingPolypeptide = dom ... end
    domain srp = dom ... end
    domain srpreceptor = dom ... end
    compose <mrna(), ribosome(), srp(), srpreceptor()>
  end

  compartment ER = com ()
    domain signalpeptidase = dom ... end
    domain hsc70 = dom ... end
    compose <signalpeptidase(), hsc70()>
  end

  domain translocon = bridge dom ... end
  compose <Cytosol(), ER(), translocon()>
end
```

4.2 Targeting the ER Membrane

We modify the model in the previous section to target a protein with α-helical transmembrane segments, such as the GLUT1 glucose transporter, into the ER membrane [LBZ+99, page 706]. The difference in the translocation of these proteins is that the polypeptide has a *signal anchor* (not at the N-terminus) and then continues with alternations between special segments called *stop transfers* and signal anchors; these segments are generally α-helices. The SRP binds to the first signal anchor, but upon translocation, the N-terminal is left in the cytosol. When the stop transfer becomes exposed, an interaction pushes the pair of α-helices into the inner membrane space with the segment between them residing in the ER lumen. Then, this process repeats for each pair of signal anchor and stop transfer segments.

First, we have an abstraction for the GLUT1 protein when it is functional and has been correctly inserted. We do not model any of its further interactions, so we leave the body of its declaration empty, but any implementation is possible.

```
domain glut1 = bridge dom ()
  init = []
end
```

The `growingPolypeptide` is modified to have a *stop transfer* site `transloconStoptransfer`. We also simplify and assume that if the signal sequence gets bound, then a proper GLUT1 protein (`glut1`) will be produced; this eliminates the `goodDone` rule. Also, we remove the `cleaveSigseq` and `hsc70Polypeptide` sites because they are unused in this model, though this is not necessary.

```
domain growingPolypeptide = dom (done,suspend)
  {badDone} done # get () -> []<badProtein()>

  (* SRP interaction with the signal sequence remains the same. *)
  srpSigseq # get (sigseqBound) -> ...

  (* Translocon interaction with the signal sequence puts the stop
     transfer site. *)
  transloconSigseq # get (transloconBound) -> [transloconBound]
  with
    create (transloconStoptransfer)
    transloconBound # put (done, transloconStoptransfer) ->
      [transloconStoptransfer]

    transloconStoptransfer # get () -> []<glut1()>
  end

  init = [badDone, srpSigseq, transloconSigseq]
end
```

Note, in the model above, as soon as the `transloconStoptransfer` occurs the `glut1` protein is formed. This abstracts the multi-step signal-anchor/stop-transfer process into one step. We have also modeled more explicitly the multi-step reaction, but since we model no other interactions for the intermediate forms, there is not much gain for that level of detail.

The translocon is almost the same except that on interaction on `transloconSigseq`, it no longer does a compartment transfer; interactions on a new site for the `stoptransfer` cause a compartment transfer into the `InnerERMembraneSpace` (which we also create).

```
domain translocon = bridge dom ()
  create (sigseqBound)

  Cytosol transloconSigseq # put (sigseqBound) -> [sigseqBound]

  ER sigseqBound # get (done) -> [stoptransfer]
  with
    Cytosol stoptransfer to InnerERMembraneSpace # put () -> [done]
    Cytosol done # get () -> init
  end

  Cytosol sigseqNear # get () -> [transloconSigseq]

  init = [transloconSigseq,sigseqNear] (* same as before *)
end
```

These minor modifications to these two domains are the only ones that need to be made, a promising sign for the composability of PML.

5 Conclusion

We have presented PML, a high-level language for modeling biological pathways. By abstracting away low-level details, PML makes models easier to write and understand. The understandability of PML models is also aided by its consistent biological metaphor of binding sites, its structuring, and its special syntax for compartments. PML seems to be a good start for developing a language suitable for writing modular and readable models of complex pathways.

Benefits of PML. Modeling using PML has several advantages over direct modeling in the π-calculus. Each domain is an independent part of a molecule, composed of a set of dependent binding sites that changes dynamically, corresponding to transformations in the molecule. In contrast, the π-calculus models we have seen use channels to represent binding sites on molecules, shared membership in a compartment, and communication between different parts of the same molecule. This overloading of the semantics of channels makes their models difficult to understand; a loose analogy in programming languages may be reading Java code versus reading assembly. When reading a model written in PML, one can, at least, immediately make a rough sketch to see what is going on. In fact, as mentioned earlier, the illustrative diagrams of molecules with binding sites in Regev and Shapiro [RS03] partially inspired the structure of PML.

Another benefit of using PML is our handling of cell compartments. As acknowledged in Regev and Shapiro [RS03], their use of private channels to represent shared membership in a compartment has a number of drawbacks. First, this method is not as biologically faithful; each molecule must "know" about its compartment membership when in actuality, molecules simply reside in a compartment and do not inherently behave differently based on their compartment membership. Second, for strict correctness, the private compartment channel must be checked at every transition since two molecules in different compartments should not be able to react. It is easy to not perform these checks properly making extension with new compartments that contain similar molecules difficult. Also, even performing this check properly is awkward, as one must check for both identical compartments and biological complementarity (a proper `put/get` pair) simultaneously, requiring an extra equality checking construct not typically included in the standard π-calculus. PML's explicit constructs for compartments alleviate all these issues, as compartment handling is done through a translation invisible to the modeler. Thus, domains can be modeled without knowledge of compartments and can be placed in several different compartments with the intended behavior. Also, we have not found a model directly written in the π-calculus that transfers a molecule between compartments, which we handle with PML's `bridge` construct. Note that Regev mentions an extension of their π-calculus variant with direct support for compartments, but we did not have access to a description of this variant to perform a direct comparison.

Using a high-level language allows the modeler to ignore certain details of any implementation of the language. We give the semantics of PML as a translation to the π-calculus, but a simulator need not implement this translation directly. If another intermediate representation could be simulated more efficiently, it could easily be employed without any change to the models. In contrast, the lack of a consistent structure in Regev's π-calculus models may make it more difficult to translate them to a different format. In fairness, we have not actually implemented PML and simulated large pathway descriptions with it, so we may not completely understand all the issues involved in simulation.

A final benefit of PML is its increased composability and modularity. Our consistent metaphor for language constructs makes it easier for different groups to decompose their descriptions with the same structure, making them easier to plug together. Since names can be introduced arbitrarily in π-calculus models, they do not have natural structures upon which to modularize. Note that we may still have composability problems because of the lack of types in PML; a type system with proper interfaces for domains and groups would greatly aid principled composability.

Future Work. Much work remains to be done to increase the usability of PML. One issue that must still be resolved is how to properly name domains and binding sites. Names should reflect the function of a domain or binding site in its context, but it is difficult to create appropriate names when one is only modeling the functionality of a single pathway. For example, if molecules `M1` and `M2` interact in a pathway, it is tempting to call the binding site `M1M2Bind` when in fact the binding mechanism is probably more general. It is also not clear whether names of binding sites should be verbs reflecting the action performed by the molecule during the reaction or nouns reflecting the properties that allow another molecule to bind at that site. These naming issues can affect composability and readability, as a person used to a certain naming style may have trouble reading a model written by someone using a completely different style.

Another difficult issue is how to properly model proximity of molecules within a compartment. Often, certain reactions serve only to move other molecules closer together, facilitating their reaction. Modeling these molecules independently does not capture their distance from each other, making the model somewhat biologically unfaithful (since it allows the molecules to interact independent of their distance from each other). We currently handle these situations in an ad-hoc manner, either using a shared private site for proximity as in Regev's work or using a signal site to only enable a reaction after previous steps have occurred to ensure proper spatial relations. A more general solution to this issue would be of great benefit in the modeling of many pathways.

For simulations of our models to be useful, they must contain quantitative information about molecular concentrations, reaction rates, *etc.* We believe that PML easily admits all the quantitative information given in Priami *et al.*'s extensions to the π-calculus [PRSS01]. Molecular concentrations are easily admitted into `compose` declarations, and reaction rates can be placed on each rule for a binding site. Furthermore, each specification of the set of active sites for a domain can include probability annotations, reflecting the relative likelihood of rules executing in that situation. In future work, we would like to implement a simulator (or translate to Regev's language to use their simulator) and model some more realistic pathways to understand exactly what quantitative information is necessary.

Longer term directions for this work include typing and graphical tools. A stronger type system could improve the language in many ways, making it safer and more easily composable. One could imagine typing binding sites based on their polarity, the length of tuples transferred through the site, and perhaps other biologically-motivated characterizations. Existing type systems for the π-calculus may serve as a starting point for a type system for PML. A graphical tool could possibly make it much easier to write the initial structure of a model. Given our metaphor for domains and binding sites, it seems that one should at least be able to draw domains and complementary binding sites, and then perhaps specify the details of transitions textually. Automatically generating graphical descriptions from textual models in PML could also aid in their understandability.

Acknowledgments. We would like to thank Roger Brent and his group for valuable discussions regarding the benefit of formal descriptions of biological systems and Aviv Regev for helping us better understand biological modeling in the π-calculus.

References

[DL03] Vincent Danos and Cosimo Laneve. Core formal molecular biology. In Pierpaolo Degano, editor, *12th European Symposium on Programming (ESOP)*, volume 2618 of *LNCS*, pages 302–318, Warsaw, Poland, April 2003.

[GP98] Peter J. E. Goss and Jean Peccoud. Quantitative modeling of stochastic systems in molecular biology by using stochastic Petri nets. *Proceedings of the National Academy of Science USA*, 95(12):6750–6755, 1998.

[HB00] Anat A. Herskovits and Eitan Bibi. Association of *Escherichia coli* ribosomes with the inner membrane requires the signal recognition particle receptor but is independent of the signal recognition particle. *Proceedings of the National Academy of Sciences USA*, 97(9):4621–4626, 2000.

[Kar01] Peter D. Karp. Pathway databases: A case study in computational symbolic theories. *Science*, 293(5537):2040–2044, 2001.

[LBZ+99] Harvey Lodish, Arnold Berk, S.Lawrence Zipursky, Paul Matsudaira, David Baltimore, and James Darnell. *Molecular Cell Biology*, chapter 17. W.H. Freeman, New York, New York, United States, fourth edition, 1999.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[MS95] Harley H. McAdams and Lucy Shapiro. Circuit simulation of genetic networks. *Science*, 269(5224):650–656, August 1995.

[PRSS01] Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80(1):25–31, 2001.

[RS03] Aviv Regev and Ehud Shapiro. The pi-calculus as an abstraction for biomolecular systems. Submitted for publication, 2003.

[RSS01] Aviv Regev, William Silverman, and Ehud Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing 2001 (PSB2001)*, volume 6, pages 459–470, Hawaii, January 2001.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge, United Kingdom, 2001.

A PML Syntax

In this section, we present the complete syntax of PML. Comments are any sequence of characters between the comment delimiters `(*` and `*)` with proper nesting. We have three classes of identifiers for rules, rule sets, sites, and blocks (*i.e.* domains, groups, and compartments) with *ruleid*, *rulesetid*, *siteid*, and *id* ranging over the respective classes. Identifiers can contain letters, numbers, underscore, and single quotes, and they must start with a letter. A name for any set of rules can be created with a ruleset declaration; as discussed in Section 2, the special init set specifies the initial set of active binding sites (rules).

We use the following conventions for presenting the grammatical rules. The *seq* suffix is used to range over comma-separated sequences. For example, *ruleidseq* ranges over comma-separated sequences of *ruleid*s. The $\langle\cdot\rangle$ brackets are used to indicate optional phrases. By convention, we use lowercase italics for a variable ranging over some class written with initial caps; for example, *domexp* ranges over DomExp.

Domains

domexp	::=	$\langle$bridge$\rangle$	
		dom($siteidseq$) $domdesc$ init $=$ [$ruleidseq$] end	
	$\mid$	id	domain identifiers
domdecl	::=	domain $id = domexp$	
domdesc	::=	$\cdot$	empty
	$\mid$	$domdesc_1\ domdesc_2$	sequence
	$\mid$	ruleset $rulesetid$ $=$ [$ruleidseq$]	rule set declarations
	$\mid$	$createdecl$	create sites
	$\mid$	$\langle\{ruleid\}\rangle$ $\langle id\rangle$ $siteid$ $\langle$to $id\rangle$ # put ($siteidseq$)	put rules
		$\texttt{->}$ $ruleset$ $\langle <instanceseq> \rangle$	
	$\mid$	$\langle\{ruleid\}\rangle$ $\langle id\rangle$ $siteid$ $\langle$to $id\rangle$ # get ($siteidseq$)	get rules
		$\texttt{->}$ $ruleset$ $\langle <instanceseq> \rangle$ $\langle$with $domexp$ end$\rangle$	
createdecl	::=	create ($siteidseq$)	
ruleset	::=	init	the initial set
	$\mid$	$rulesetid$	declared sets
	$\mid$	[$ruleidseq$]	basic sets

Groups

grpexp	::=	$\langle$bridge$\rangle$	
		grp($siteidseq$) $grpdesc$ compose $<instanceseq>$ end	
	$\mid$	id	group identifiers
grpdecl	::=	group $id = grpexp$	
grpdesc	::=	$\cdot$	empty
	$\mid$	$grpdesc_1\ grpdesc_2$	sequence
	$\mid$	$createdecl$	create sites
	$\mid$	$domdecl$	domain declarations
	$\mid$	$grpdecl$	group declarations
	$\mid$	$comdecl$	compartment declarations
instance	::=	id($siteidseq$)	

Compartments

comexp	::=	com($siteidseq$) $grpdesc$ compose $<instanceseq>$ end	
	$\mid$	id	compartment identifier
comdecl	::=	compartment $id = comexp$	

B Formal Translation from CorePML to the π-calculus

Recall that at the top-level, a pathway in CorePML is a compose of a set of instantiations of domains and groups. More explicitly, we say that a model at the top-level is an expression of the form

$$\mathit{modeldesc}\ \textsf{compose}\ {<}\mathit{instance}_1, \mathit{instance}_2, \ldots, \mathit{instance}_n{>}$$

where

$$\begin{array}{rcll} \mathit{modeldesc} & ::= & \cdot & \text{empty} \\ & | & \mathit{modeldesc}_1\ \mathit{modeldesc}_2 & \text{sequence} \\ & | & \mathit{domdecl} & \text{domain declarations} \\ & | & \mathit{grpdecl} & \text{group declarations} \end{array}$$

We then define the translation to the π-calculus inductively on the structure of a CorePML model (*modeldesc*).

Intuitively, every domain and group represents some biological entity, and we translate them into π-calculus processes. A compose declaration in the π-calculus is then a parallel composition of each of the instantiations. A domain is the smallest unit of mutually dependent binding sites. The rules indicate what dynamic behavior occurs upon a binding interaction on that site, specifically what set of binding sites are present in the next state. We then can represent the next reaction as a competition between all the binding sites in the present site. This can be expressed by choice between the representation of each of the rules. Because these rules can be recursive, this translates to a use of replication in the π-calculus in a similar manner to handling recursive definitions [Mil99, SW01].

First, let Exp be the set of domain and group expressions (DomExp and GrpExp) and $\delta : \mathrm{Id} \to \mathrm{Exp}$ be a mapping from identifiers to domain and group expressions with δ ranging over Δ. Also, we will need to generate fresh names, we call a *token* for translating reaction rules. We write $[y/x]P$ as capture-avoiding substitution of y for x in P.

We define the translation for group descriptions (*grpdesc*) $[\![\cdot]\!]_{\mathrm{grpdesc}} : \mathrm{GrpDesc} \to \Delta \to \Delta$ as possibly extending an environment that maps identifiers to domain or group expressions and use this same translation function for model descriptions (*modeldesc*) as they are simply a subset of group descriptions.

$$[\![\cdot]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} \delta$$

$$[\![\mathit{grpdesc}_1\ \mathit{grpdesc}_2]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} [\![\mathit{grpdesc}_2]\!]_{\mathrm{grpdesc}}\ ([\![\mathit{grpdesc}_1]\!]_{\mathrm{grpdesc}}\ \delta)$$

$$[\![\textsf{domain}\ \mathit{id} = \mathit{domexp}]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} \delta[\mathit{id} \mapsto \mathit{domexp}]$$

$$[\![\textsf{group}\ \mathit{id} = \mathit{grpexp}]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} \delta[\mathit{id} \mapsto \mathit{grpexp}]$$

The domain and group declarations extend δ and sequencing composes the translations.

The translation of domain descriptions $[\![\cdot]\!]^{\delta}_{\mathrm{domdesc}} \cdot : \mathrm{DomDesc} \to \Delta \to \mathrm{P} \to \mathrm{P}$ translates the reaction rules into a π-calculus process that for each rule. We assume that we have a mapping $\rho : \mathrm{RuleId} \to \mathrm{Token}$ from rule identifiers to fresh tokens and have made sure any lexical scoping constraints have been respected.

$$[\![\cdot]\!]^{\delta}_{\mathrm{domdesc}}\ P \stackrel{\mathrm{def}}{=} P$$

$$[\![\mathit{domdesc}_1\ \mathit{domdesc}_2]\!]^{\delta}_{\mathrm{domdesc}}\ P \stackrel{\mathrm{def}}{=} [\![\mathit{domdesc}_2]\!]^{\delta}_{\mathrm{domdesc}}\ ([\![\mathit{domdesc}_1]\!]^{\delta}_{\mathrm{domdesc}}\ P)$$

Like group descriptions, sequencing just composes the translation.

$$
\begin{aligned}
&[\![\{\mathit{ruleid}\}\ \mathit{siteid}\ \#\ \mathsf{put}\ (\mathit{siteid}_1, \mathit{siteid}_2, \ldots, \ldots, \mathit{siteid}_m)\\
&\quad \texttt{->}\ \ [\mathit{ruleid}_1, \mathit{ruleid}_2, \ldots, \mathit{ruleid}_k]\texttt{<}\mathit{instance}_1, \mathit{instance}_2, \ldots, \mathit{instance}_n\texttt{>}]\!]^{\delta}_{\text{domdesc}}\ P\\
&\quad\ \stackrel{\text{def}}{=}\ !\Big(t().\overline{\mathit{siteid}}\langle \mathit{siteid}_1, \mathit{siteid}_2, \ldots, \mathit{siteid}_m\rangle.\\
&\qquad\quad \big(\overline{\rho(\mathit{ruleid}_1)}\langle\rangle + \overline{\rho(\mathit{ruleid}_2)}\langle\rangle + \cdots + \overline{\rho(\mathit{ruleid}_k)}\langle\rangle\\
&\qquad\quad\ \mid [\![\mathit{instance}_1]\!]_{\text{exp}}\ \delta \mid [\![\mathit{instance}_2]\!]_{\text{exp}}\ \delta \mid \cdots \mid [\![\mathit{instance}_n]\!]_{\text{exp}}\ \delta\big)\Big)\\
&\qquad \mid P\\
&\quad \text{where } t = \rho(\mathit{ruleid})
\end{aligned}
$$

For a put rule, we send on the channel corresponding to the site, and enable the next set of sites. Also, any instantiations are translated. As noted above, the rules that describe binding reactions on sites can be recursive and can be translated by using replication. Rather than distinguishing between recursive and non-recursive rules, we translate each rule uniformly, treating rules, in essence, as unrestricted (non-linear) function definitions and function calls.

$$
\begin{aligned}
&[\![\{\mathit{ruleid}\}\ \mathit{siteid}\ \#\ \mathsf{get}\ (\mathit{siteid}_1, \mathit{siteid}_2, \ldots, \ldots, \mathit{siteid}_m)\\
&\quad \texttt{->}\ \ [\mathit{ruleid}_1, \mathit{ruleid}_2, \ldots, \mathit{ruleid}_k]\texttt{<}\mathit{instance}_1, \mathit{instance}_2, \ldots, \mathit{instance}_n\texttt{>}\\
&\quad \mathsf{with}\ \mathit{domdesc}\ \mathsf{end}]\!]^{\delta}_{\text{domdesc}}\ P\\
&\quad\ \stackrel{\text{def}}{=}\ !\Big(t().\mathit{siteid}(\mathit{siteid}_1, \mathit{siteid}_2, \ldots, \mathit{siteid}_m).\\
&\qquad\quad \big(\overline{\rho(\mathit{ruleid}_1)}\langle\rangle + \overline{\rho(\mathit{ruleid}_2)}\langle\rangle + \cdots + \overline{\rho(\mathit{ruleid}_k)}\langle\rangle\\
&\qquad\quad\ \mid [\![\mathit{instance}_1]\!]_{\text{exp}}\ \delta \mid [\![\mathit{instance}_2]\!]_{\text{exp}}\ \delta \mid \cdots \mid [\![\mathit{instance}_n]\!]_{\text{exp}}\ \delta\\
&\qquad\quad\ \mid [\![\mathit{domdesc}]\!]^{\delta}_{\text{domdesc}}\ \mathbf{0}\big)\Big)\\
&\qquad \mid P\\
&\quad \text{where } t = \rho(\mathit{ruleid})
\end{aligned}
$$

The translation for get is similar to put except that the knowledge of the other sites yields possibly new sites in the with clause.

Instantiations are made with the compose construct that intuitively places a molecule described by the grp or dom expression in the pathway. We equate sites with channels in the π-calculus using the same names in both domains. To translate an instantiation, the translation function $[\![\cdot]\!]_{\text{exp}}$: Exp $\rightarrow \Delta \rightarrow$ P creates names for the local names and substitutes the names given by the instantiation for the parameters in the body of the translation group or domain expression.

$$
\begin{array}{l}
[\![\mathsf{grp}(siteid_1, siteid_2, \ldots, siteid_n) \\
\quad \mathsf{create}\ (siteid''_1, siteid''_2, \ldots, siteid''_m) \\
\quad grpdesc \\
\quad \mathsf{compose}\ {<}instance_1, instance_2, \ldots, instance_k{>} \\
\mathsf{end}\ (siteid'_1, siteid'_2, \ldots, siteid'_n)]\!]_{\mathrm{exp}}\ \delta \\
\quad \stackrel{\mathrm{def}}{=} [siteid'_1, siteid'_2, \ldots, siteid'_n / siteid_1, siteid_2, \ldots, siteid_n] \\
\qquad \big(\mathsf{new}\ siteid''_1,\ siteid''_2,\ \ldots,\ siteid''_m \\
\qquad\quad [\![instance_1]\!]_{\mathrm{exp}}\ \delta' \mid [\![instance_2]\!]_{\mathrm{exp}}\ \delta' \mid \cdots \mid [\![instance_k]\!]_{\mathrm{exp}}\ \delta'\big) \\
\text{where } \delta' = [\![grpdesc]\!]_{\mathrm{grpdesc}}\ \delta
\end{array}
$$

The body of the group expression translates to parallel composition on the instances given by its compose declaration. The instances can be of any of the declarations in δ or the domains or groups declared in this group.

$$
\begin{array}{l}
[\![\mathsf{dom}(siteid_1, siteid_2, \ldots, siteid_n) \\
\quad \mathsf{create}\ (siteid''_1, siteid''_2, \ldots, siteid''_m) \\
\quad domdesc \\
\quad \mathsf{init} = [ruleid_1, ruleid_2, \ldots, ruleid_k] \\
\mathsf{end}\ (siteid'_1, siteid'_2, \ldots, siteid'_n)]\!]_{\mathrm{exp}}\ \delta \\
\quad \stackrel{\mathrm{def}}{=} [siteid'_1, siteid'_2, \ldots, siteid'_n / siteid_1, siteid_2, \ldots, siteid_n] \\
\qquad \big(\mathsf{new}\ siteid''_1,\ siteid''_2,\ \ldots,\ siteid''_m,\ \rho(ruleid_1),\ \ldots,\ \rho(ruleid_p) \\
\qquad\quad [\![domdesc]\!]^{\cdot}_{\mathrm{domdesc}}\ \mathbf{0} \mid \overline{\rho(ruleid_1)}\langle\rangle + \overline{\rho(ruleid_2)}\langle\rangle + \cdots + \overline{\rho(ruleid_k)}\langle\rangle\big)
\end{array}
$$

For domains, we must also create new declarations for all rule tokens, ensuring their proper scoping (the above translation assumes there are p rules in the domain). The init construct translates to a choice of sending on the tokens for the rules in the set.

$$
[\![id(siteid'_1, siteid'_2, siteid'_n)]\!]_{\mathrm{exp}}\ \delta \ \stackrel{\mathrm{def}}{=}\ [\![\delta(id)(siteid'_1, siteid'_2, siteid'_n)]\!]_{\mathrm{exp}}\ \delta
$$

This translation simply looks up the expression corresponding to the name in an expression, and then performs the translation of the instantiation of the expression.

Finally, we can define the translation function $[\![\cdot]\!]$ from models in CorePML to the polyadic π-calculus.

$$
\begin{array}{l}
[\![modeldesc\ \mathsf{compose}\ {<}instance_1, instance_2, \ldots, instance_n{>}]\!] \\
\quad \stackrel{\mathrm{def}}{=}\ [\![instance_1]\!]_{\mathrm{exp}}\ \delta' \mid [\![instance_2]\!]_{\mathrm{exp}}\ \delta' \mid \cdots \mid [\![instance_n]\!]_{\mathrm{exp}}\ \delta' \\
\text{where } \delta' = [\![modeldesc]\!]_{\mathrm{grpdesc}}\ \cdot
\end{array}
$$

We translate the model description into the domain/group expression environment and then compose the translations of the instantiations in parallel in that environment.