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

Bor-Yuh Evan Chang[1] Manu Sridharan[2]

Computer Science Division
University of California, Berkeley
Berkeley, CA, U.S.A.

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 languages, which have a precise and consistent semantics. While precise, many of these languages may be too low-level to model feasibly 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.

Keywords: Modeling language, biological systems, pathways

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. In this paper, we investigate the use of formal 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

⋆ This research was supported in part by the National Science Foundation Grants No. CCR-9875171, No. CCR-0081588, and No. CCR-0085949, a California Microelectronics Fellowship, and a National Defense Science and Engineering Graduate Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[1] E-mail: bec@cs.berkeley.edu

[2] E-mail: manu_s@cs.berkeley.edu

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.

McAdams and Shapiro have integrated the traditional biochemical kinetic modeling with circuit diagrams and simulations akin to circuits in electrical engineering [8] to help elucidate timing relationships between chemical equations. Pathway databases [6] organize and store information about molecules and their interactions in a symbolic form and provide various ways of querying the database. Our work is complementary to this in that we seek suitable representations that clearly capture the dynamic behavior of pathways, while databases are currently more suitable for describing static configurations. Petri nets is a formalism that attempts to better capture the dynamic behavior of biological systems [4]. However, Petri nets still have a drawback similar to chemical kinetic models in that each state of a molecular species is represented by a place (rather than simply the biological entity). Recently, several researchers have proposed modeling biological pathways as concurrent computational processes utilizing mathematical formalisms, such as process algebras [13,10,12,3]. Regev *et al.* have suggested various forms of the π-calculus [9] as a framework for abstracting biological pathways in this manner. 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 describes 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.

While the π-calculus seems to be an entirely appropriate formalism as an underlying machine model, we believe that it is too low-level to directly model in. Thus, we have designed a high-level modeling language for pathways called PML (Pathway Modeling Language) that translates into the π-calculus. 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.

We present PML through some example models of biological systems in Sec. 2, followed by a full presentation of PML and its semantics in Sec. 3. Then, we give a model of cotranslational translocation on the endoplasmic reticulum (ER) membrane to demonstrate the composability of PML in Sec. 4. Finally, we discuss the benefits of PML and future work in Sec. 5.

2 PML Models

Michaelis-Menten Model. PML is largely inspired by an informal graphical style for presenting reactions used in Regev and Shapiro [12]. A Michaelis-Menten reaction is depicted in this style in Fig. 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

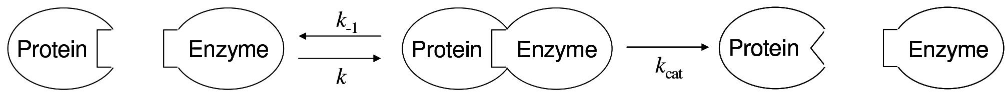

Fig. 1. A graphical view of a Michaelis-Menten reaction.

to the enzyme, but it has a new capability to bind to other molecules.

```
1  group MichaelisMenten = grp ()
2    create (bind_s, bind_p)
3    domain Enzyme = dom ()
4      create (release_s, release_p)
5      bind_s # put (release_s, release_p) ->
6        [release_s, release_p]
7      release_s # put () -> init
8      release_p # put () -> init
9      init = [bind_s]
10   end
11   domain Protein = dom ()
12     bind_s # get (release_s, release_p) ->
13       [release_s, release_p]
14     with
15       release_s # get () -> init
16       release_p # get () -> [bind_p]
17     end
18     bind_p # ...
19     init = [bind_s]
20   end
21   compose  <Enzyme(), Protein()>
22 end
```

Fig. 2. Michaelis-Menten reaction.

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 shown in Fig. 2 (lines 3–10). At any point in time, each domain has a set of active binding sites. Initially, the `Enzyme` domain has the `bind_s` site active (specified by the `init` declaration); this is the site through which the enzyme can bind to the protein in Fig. 1. The behavior of an enzyme after it binds with a protein is defined by the rule for the `bind_s` site (lines 5 and 6). Here, the enzyme "puts" two binding sites, `release_s` and `release_p`, that it has created locally and then activates those two binding sites (specified after the `->`). Binding sites are created locally to perform reactions with individual molecules, as opposed to an arbitrary molecule in a particular class.

As Fig. 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_s` and `release_p` sites (lines 7 and 8). After a reaction at either site, the enzyme returns to its `init` state (with only the `bind_s` site active). A non-deterministic choice determines whether a reaction occurs on the `release_s` site or on the `release_p` 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 (Fig. 2, lines 11–20). Initially, the protein has its `bind_s` site active, allowing it to bind to the enzyme. When it binds to an enzyme on the site, it gets the `release_s` and `release_p` 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 15), re-enabling the `bind_s` site, or it progresses (line 16), enabling a new `bind_p` site that will allow reactions with new molecules (elided). We declare the rules for the `release_s` and `release_p` sites in a with construct, as they are bound in the get construct.

We then use a group construct to simply group together these two domains. The compose declaration indicates that the initial state of the system is one enzyme molecule and one protein molecule.

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. In Fig. 3, 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

```
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
```

Fig. 3. Compartments example.

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.

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. 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 with identifiers being lexically-scoped.

PML to CorePML. CorePML has the following properties: all rules have explicit rule names, there is at most one `create` declaration in each domain or group and appears first, and there are no `compartment` or `bridge` constructs. Any well-formed PML model can be transformed to satisfy the first two properties in a straightforward manner. For the last property, 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.

```
1 domain MolA_Cytosol = dom ()
2   bind_a_Cytosol # put () -> init
3   bind_a_ER # put () -> init_ER
4   bind_a_CytERBridge # put () -> init_ER
5   ruleset init_ER = [bind_a_ER]
6   init = [bind_a_Cytosol]
7 end
```

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. 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. For example, these transformations applied to `MolA` in the `Cytosol` from Fig. 3 is shown above (lines 1,2,6). 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.

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 (how this is determined is discussed further in the extended version [2]). 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. We then 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 (line 3). 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.

Finally, we must add rules to domains to properly handle the actual compartment change. 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`. 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 `bind_a_CytERBridge` rule (line 4). 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`.

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.

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 Sec. 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 Fig. 2 would be translated to $[\![\texttt{Enzyme}]\!] \mid [\![\texttt{Protein}]\!]$, 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_s` rule of the `Enzyme` domain in Fig. 2 is

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

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. 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_s` or `release_p`. 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:

$$\begin{array}{l} !(bsToken().\overline{bind\_s}\langle release\_s, release\_p\rangle.\overline{rsToken}\langle\rangle + \overline{rpToken}\langle\rangle) \\ \mid\ !(rsToken().\overline{release\_s}\langle\rangle.\overline{bsToken}\langle\rangle) \mid !(rpToken().\overline{release\_p}\langle\rangle.\overline{bsToken}\langle\rangle) \mid \overline{bsToken}\langle\rangle \end{array}$$

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 [7, page 698]. We then modify this model to describe the synthesis and insertion into the ER membrane of the GLUT1 glucose transporter [7, page 706] to emphasize the few changes that need to be made.

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.

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

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. 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. 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.

```
domain ribosome = dom ()
  create (mrnaDone, ppDone, ppSusp)
  translate # put (mrnaDone) ->
    [mrnaDone,pptideSusp]
    <growingPolypeptide(ppDone,ppSusp)>
  mrnaDone # put () -> [ppDone]
  ppDone # put () -> [translate,ppDone]
  ppSusp # get (restart) -> [restart]
  with restart #get () -> [mrnaDone,ppSusp]
  end
  init = [translate]
end
```

The growing polypeptide is an abstraction for the polypeptide while it is being translated that interacts with several entities. An 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 [5]. 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.

```
domain growingPolypeptide = dom (done,suspend)
  {badDone}  done # get () -> []<badProtein()>
  {goodDone} done # get () -> []<goodProtein()>
  srpSigseq # get (sigseqBound) -> [suspend]
  with sigseqBound # get () -> [restart] end
  create (restart)
  suspend # put (restart) -> [sigseqBound]
  restart # put () ->
    [badDone,transloconSigseq,cleaveSigseq]
  transloconSigseq # get (transloconBound) ->
    [transloconBound]
  with transloconBound
    # put (done) -> [badDone,cleaveSigseq]
  end
  cleaveSigseq # get () -> [hsc70Polypep,badDone]
  hsc70Polypep # get () -> [goodDone]
  init = [badDone,srpSigseq,
         transloconSigseq,cleaveSigseq]
end
```

The SRP and SRP receptor have some straightforward interactions. 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 possible 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).

```
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 translocon is a membrane protein potentially with sites on either the cytosol or ER lumen side. 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.

```
domain translocon = bridge dom ()
  create (sigseqBound)
  (* Transfer the other molecule to ER. *)
  Cytosol transloconSigseq to ER
    # put (sigseqBound) -> [sigseqBound]
  ER sigseqBound # get (done) -> [done]
  with Cytosol done # get () -> init end
  Cytosol sigseqNear
    # get () -> [transloconSigseq]
  init = [transloconSigseq,sigseqNear]
end
```

After interacting with the polypeptide, the signal peptidase and Hsc70 simply returns to the initial state ready to modify/chaperone the next polypeptide. 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.

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

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

Finally, we place an instantiation of `mrna`, `ribosome`, `growingPolypeptide`, `srp`, and `srpreceptor` in the `Cytosol` compartment and a copy of `signalpeptidase` and `hsc70` in the `ER` compartment. Then, we can group these compartments with an instantiation of the bridge domain `translocon` (see the extended version [2] for a complete listing).

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 [7, 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.

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. This also 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). These minor modifications to these two domains are the only ones that need to be made, a promising sign for the composability of PML. The complete code is given in the extended version [2].

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.

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. PML also increases 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.

As acknowledged in Regev and Shapiro [12], their use of private channels to represent shared membership in a compartment has a number of drawbacks. In recent work [11], Regev *et al.* propose a biologically-motivated variant of the ambient calculus [1] to handle better compartments. An interesting difference is that while we use the compartment construct for membrane compartments but retain the use of private sites for complexing, they choose to express both using ambients. The introduction of the compartment construct is based largely on the desire to explicitly separate these two notions. Also, their formalism seems to express better compartment merging and splitting in addition to transport between compartments. It may be possible to adapt and extend PML to use the bioambient calculus as the underlying machine model.

Much work remains to be done to increase the usability of PML, such as 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. Another difficult issue is how to properly model proximity of molecules within a compartment. We currently handle these situations, 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. 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 readily admits all the quantitative information given in Priami *et al.*'s extensions to the π-calculus [10], but this must be further investigated. Also, graphical tools for both input and display could aid in the usability and understandability of PML. Lastly, a strong type system could improve the language in many ways, making it safer and more easily composable, possibly building on existing type systems for the π-calculus.

Acknowledgments. We would like to thank Roger Brent and his group for valuable discussions regarding the benefit of formal descriptions of biological systems, Aviv Regev for helping us better understand biological modeling in the π-calculus, and Gwong-Jen Chang, Robert Schneck, and the anonymous reviewers for their comments on earlier drafts of this paper.

References

[1] Cardelli, L. and A. D. Gordon, *Mobile ambients*, Theoretical Computer Science **240** (2000), pp. 177–213.

[2] Chang, B.-Y. E. and M. Sridharan, *PML: Toward a high-level formal language for biological systems*, Technical Report UCB/CSD-03-1251, University of California, Berkeley (2003).

[3] Danos, V. and C. Laneve, *Core formal molecular biology*, in: P. Degano, editor, *12th European Symposium on Programming (ESOP)*, LNCS **2618**, Warsaw, Poland, 2003, pp. 302–318.

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

[5] Herskovits, A. A. and E. 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** (2000), pp. 4621–4626.

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

[7] Lodish, H., A. Berk, S. L. Zipursky, P. Matsudaira, D. Baltimore and J. Darnell, "Molecular Cell Biology," W.H. Freeman, New York, New York, U.S.A., 1999, fourth edition .

[8] McAdams, H. H. and L. Shapiro, *Circuit simulation of genetic networks*, Science **269** (1995), pp. 650–656.

[9] Milner, R., "Communicating and Mobile Systems: the π-calculus," Cambridge University Press, 1999 .

[10] Priami, C., A. Regev, E. Shapiro and W. Silverman, *Application of a stochastic name passing calculus to representation and simulation of molecular processes*, Information Processing Letters **80** (2001), pp. 25–31.

[11] Regev, A., E. M. Panina, W. Silverman, L. Cardelli and E. Shapiro, *Bioambients: An abstraction for biological compartments* (2003), to appear.

[12] Regev, A. and E. Shapiro, *The pi-calculus as an abstraction for biomolecular systems* (2003), submitted for publication.

[13] Regev, A., W. Silverman and E. Shapiro, *Representation and simulation of biochemical processes using the pi-calculus process algebra*, in: *Pacific Symposium on Biocomputing 2001 (PSB2001)*, **6**, Hawaii, U.S.A., 2001, pp. 459–470.

[14] Sangiorgi, D. and D. 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 Sec. 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 ⟨·⟩ 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      ::= ⟨bridge⟩
                dom(siteidseq) domdesc init = [ruleidseq] end
              | id                                                   domain identifiers
domdecl     ::= domain id = domexp
domdesc     ::= ·                                                    empty
              | domdesc1 domdesc2                                    sequence
              | ruleset rulesetid = [ruleidseq]                      rule set declarations
              | createdecl                                           create sites
              | ⟨{ruleid}⟩ ⟨id⟩ siteid ⟨to id⟩ # put (siteidseq)     put rules
                –> ruleset ⟨<instanceseq>⟩
              | ⟨{ruleid}⟩ ⟨id⟩ siteid ⟨to id⟩ # get (siteidseq)     get rules
                –> ruleset ⟨<instanceseq>⟩ ⟨with domexp end⟩
createdecl  ::= create (siteidseq)
ruleset     ::= init                                                 the initial set
              | rulesetid                                            declared sets
              | [ruleidseq]                                          basic sets
```

Groups

grpexp	$::=$	$\langle\mathsf{bridge}\rangle$	
		$\mathsf{grp}(\mathit{siteidseq})\ \mathit{grpdesc}\ \mathsf{compose}\ <\mathit{instanceseq}>\ \mathsf{end}$	
	$\mid$	id	group identifiers
$\mathit{grpdecl}$	$::=$	$\mathsf{group}\ \mathit{id} = \mathit{grpexp}$	
$\mathit{grpdesc}$	$::=$	$\cdot$	empty
	$\mid$	$\mathit{grpdesc}_1\ \mathit{grpdesc}_2$	sequence
	$\mid$	$\mathit{createdecl}$	create sites
	$\mid$	$\mathit{domdecl}$	domain declarations
	$\mid$	$\mathit{grpdecl}$	group declarations
	$\mid$	$\mathit{comdecl}$	compartment declarations
$\mathit{instance}$	$::=$	$\mathit{id}(\mathit{siteidseq})$	

Compartments

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

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

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

where

$\mathit{modeldesc}$	$::=$	$\cdot$	empty
	$\mid$	$\mathit{modeldesc}_1\ \mathit{modeldesc}_2$	sequence
	$\mid$	$\mathit{domdecl}$	domain declarations
	$\mid$	$\mathit{grpdecl}$	group declarations

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 [9,14].

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*) $[\![\cdot]\!]_{\mathrm{grpdesc}}\ \cdot\ :\ \mathrm{GrpDesc} \to \Delta \to \Delta$ 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.

$$
\begin{aligned}
&[\![\cdot]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} \delta \\
&[\![\mathit{grpdesc}_1\ \mathit{grpdesc}_2]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} [\![\mathit{grpdesc}_2]\!]_{\mathrm{grpdesc}}\ ([\![\mathit{grpdesc}_1]\!]_{\mathrm{grpdesc}}\ \delta) \\
&[\![\textsf{domain}\ \mathit{id} = \mathit{domexp}]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} \delta[\mathit{id} \mapsto \mathit{domexp}] \\
&[\![\textsf{group}\ \mathit{id} = \mathit{grpexp}]\!]_{\mathrm{grpdesc}}\ \delta \stackrel{\mathrm{def}}{=} \delta[\mathit{id} \mapsto \mathit{grpexp}]
\end{aligned}
$$

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.

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

Like group descriptions, sequencing just composes the translation.

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

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.

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

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 $[\![\cdot]\!]_{\text{exp}} \cdot : \text{Exp} \rightarrow \Delta \rightarrow \text{P}$ 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.

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

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.

$$
\begin{array}{l}
[\![\mathsf{dom}(\mathit{siteid}_1, \mathit{siteid}_2, \ldots, \mathit{siteid}_n) \\
\quad \mathsf{create}\ (\mathit{siteid}''_1, \mathit{siteid}''_2, \ldots, \mathit{siteid}''_m) \\
\quad \mathit{domdesc} \\
\quad \mathsf{init} = [\mathit{ruleid}_1, \mathit{ruleid}_2, \ldots, \mathit{ruleid}_k] \\
\mathsf{end}\ (\mathit{siteid}'_1, \mathit{siteid}'_2, \ldots, \mathit{siteid}'_n)]\!]_{\text{exp}}\ \delta \\
\quad \stackrel{\text{def}}{=} [\mathit{siteid}'_1, \mathit{siteid}'_2, \ldots, \mathit{siteid}'_n / \mathit{siteid}_1, \mathit{siteid}_2, \ldots, \mathit{siteid}_n] \\
\qquad \big(\mathsf{new}\ \mathit{siteid}''_1,\ \mathit{siteid}''_2,\ \ldots,\ \mathit{siteid}''_m,\ \rho(\mathit{ruleid}_1),\ \ldots,\ \rho(\mathit{ruleid}_p) \\
\qquad\quad [\![\mathit{domdesc}]\!]^{\cdot}_{\text{domdesc}}\ \mathbf{0} \mid \overline{\rho(\mathit{ruleid}_1)}\langle\rangle + \overline{\rho(\mathit{ruleid}_2)}\langle\rangle + \cdots + \overline{\rho(\mathit{ruleid}_k)}\langle\rangle\big)
\end{array}
$$

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.

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

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.

$$
\begin{array}{l}
[\![\mathit{modeldesc}\ \mathsf{compose}\ <\mathit{instance}_1, \mathit{instance}_2, \ldots, \mathit{instance}_n>]\!] \\
\quad \stackrel{\text{def}}{=} [\![\mathit{instance}_1]\!]_{\text{exp}}\ \delta' \mid [\![\mathit{instance}_2]\!]_{\text{exp}}\ \delta' \mid \cdots \mid [\![\mathit{instance}_n]\!]_{\text{exp}}\ \delta' \\
\text{where } \delta' = [\![\mathit{modeldesc}]\!]_{\text{grpdesc}}\ \cdot
\end{array}
$$

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