Synthesis of Biological Models from Mutation Experiments

Ali Sinan Köksal[1] Yewen Pu[1] Saurabh Srivastava[1]
Rastislav Bodík[1] Jasmin Fisher[2] Nir Piterman[3]

[1]University of California, Berkeley [2]Microsoft Research, Cambridge [3]University of Leicester
koksal@cs.berkeley.edu, yewenpu@mit.edu, saurabhs@cs.berkeley.edu,
bodik@cs.berkeley.edu, jasmin.fisher@microsoft.com, nir.piterman@le.ac.uk

Abstract

Executable biology presents new challenges to formal methods. This paper addresses two problems that cell biologists face when developing formally analyzable models.

First, we show how to automatically synthesize a concurrent in-silico model for cell development given in-vivo experiments of how particular mutations influence the experiment outcome. The problem of synthesis under mutations is unique because mutations may produce non-deterministic outcomes (presumably by introducing races between competing signaling pathways in the cells) and the synthesized model must be able to replay all these outcomes in order to faithfully describe the modeled cellular processes. In contrast, a "regular" concurrent program is correct if it picks any outcome allowed by the non-deterministic specification. We developed synthesis algorithms and synthesized a model of cell fate determination of the earthworm *C. elegans*. A version of this model previously took systems biologists months to develop.

Second, we address the problem of under-constrained specifications that arise due to incomplete sets of mutation experiments. Under-constrained specifications give rise to distinct models, each explaining the same phenomenon differently. Addressing the ambiguity of specifications corresponds to analyzing the space of plausible models. We develop algorithms for detecting ambiguity in specifications, i.e., whether there exist alternative models that would produce different fates on some unperformed experiment, and for removing redundancy from specifications, i.e., computing minimal non-ambiguous specifications.

Additionally, we develop a modeling language and embed it into Scala. We describe how this language design and embedding allows us to build an efficient synthesizer. For our *C. elegans* case study, we infer two observationally equivalent models expressing different biological hypotheses through different protein interactions. One of these hypotheses was previously unknown to biologists.

This work was supported in part by NSF grant 1019343 to the Computing Research Association for the CIFellows Project, and by NSF grant CCF-1139011.

POPL'13, January 23–25, 2013, Rome, Italy.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Program Synthesis, Specification Ambiguity Analysis, Executable Biology

1. Introduction

Diseases can be caused by perturbed gene and protein regulatory networks. For example, disease X may be related to the levels of proteins P and R, and P may negatively regulate R. Once the level of P is decreased, high levels of R may cause disease X. To avoid the disease, we may want to increase the level of P. One way to infer protein regulatory networks is to carry out *mutation experiments*, in which cells are genetically modified to suppress or enhance the activity of a certain protein, leading the cell to exhibit abnormal behavior such as uncontrolled cell divisions. If, by suppressing the activity of protein P, the resulting phenotype can be attributed to, say, an increased activity of a known protein R, we can infer from this mutation experiment that P negatively regulates R. From many such inferences, experimental biologists deduce regulatory networks that describe the causal events leading to specific cell fates and other behaviors.

Experimental biologists are concerned about the correctness of their models that give a dynamic explanation of how the observed outcomes are produced. Executable biology [14] addresses this concern by building executable models that can be verified against performed experiments. Treating cells as concurrent agents models the fact that cells do not evolve at synchronous rates [19, 20]. Verification ensures that a concurrent model is correct for all variations of cell growth rates, by exploring all possible executions of the model [15].

Unfortunately, turning informal maps of regulatory networks common in biological literature into executable models is laborious because it involves explicitly defining timing delay and strength of how multiple proteins regulate each other. In our previous work, some of us developed a model of vulval cell fate specification (*i.e.*, how vulval cells make the decision to develop into a particular cell type) in the *C. elegans* worm [12]. This model correctly predicted an unknown protein-protein interaction, however it took several months to tweak the details of the model before it was verified against the experimental data. Whenever new experiments are added, or when the model is extended with new components, similar tweaks are required.

This paper develops techniques for synthesizing executable models from experimental observations and prior biological knowledge. Two challenges make this synthesis problem interesting. First, the outcomes of some cellular systems, such as fates of stem cells, are non-deterministic. For example, in the *C. elegans* system that we study, some mutations cause the six observed vulval pre-

cursor cells (VPCs) to acquire one of two alternative fates, presumably due to races in the communication among cells. The desired executable model must be able to reproduce all the observed behavior in order to be correct. We synthesize concurrent cell models such that, for each observable outcome, there exists a schedule that leads the model to produce the outcome. This requirement makes our synthesis task a new problem, which is more complex than what has been previously addressed.

Second, the incomplete set of mutation experiments forms only a partial specification. Because only certain genes are mutated from the total combinatorial set of possible mutations, we cannot be certain that an executable model that verifies against these mutations, whether it is synthesized or manually constructed, is the sole explanation of the cellular regulatory process. This is because there could exist an alternative model that is observationally identical on the current specification but observationally distinct on an additional mutation. Finding such an additional mutation would uncover *ambiguity* in the current specification.

To confirm that we have synthesized a unique model, we go beyond synthesis and develop methods for the analysis of the space of plausible models, i.e., models that agree with the specification. If observationally distinct models exist, we suggest a new mutation that differentiates them. If no alternative models exist, we determine the smallest set of experiments that is sufficient to arrive at the unique model. Finding such a minimal set is interesting because, should biologists decide to redo the experiments for validation, they only need to perform the experiments that suffice to synthesize a unique model. Finally, it is interesting to ask whether there are observationally identical but internally different models. Such models present regulatory networks where the network function is "implemented" via different protein interactions. These models cannot be distinguished by observing phenotypes; we must, say, instrument proteins with fluorescent markers (similar to tracing the program) and observe the cell during its development. This is a harder experiment, but the cost of instrumentation is reduced with the help of formal methods, as we can identify which genes to mark given the internal differences between the observationally identical models.

We have built an efficient verifier, synthesizer and specification ambiguity analyzer that implements algorithms for the analyses described above. Our synthesizer takes as input the mutations, the results observed under mutations and a template structure of the cells, and from them it generates a verified model. The template of the cell defines the cell components, and a superset of their interconnections (inhibition, activation), allowing biologists to formally state existing knowledge on the system being modeled. Additionally, the granularity of the discretized concentration levels for each component is set a priori. What we synthesize is the internal logic and timing of the components, i.e., how their concentration changes in terms of their incoming signals, and we therefore off-load the most difficult task of systems biology modeling to a computation search engine.

This paper makes the following contributions:

1. We designed SBL, a domain-specific language for expressing our models using an execution model with restricted asynchrony called bounded asynchrony [15]. We embed SBL into the Scala programming language [25] and build a lightweight synthesizer, which is publicly available [21]. We describe how to translate SBL programs into formulas in order to solve synthesis and specification analysis problems (Sections 3 and 4).
2. We formulate the verification problem (Section 5.1) and the program synthesis problem (Section 5.2). We observe that unlike previous synthesis tasks, e.g., concurrent synthesis [28] or synthesis from examples [17] or invariants [30], which are expressed as formulas with two levels of quantification (2QBF), this problem is expressed as a formula with three levels of quantification (3QBF), which makes it a new kind of problem. We develop efficient algorithms for solving this problem that reduce to three communicating SAT solvers.
3. We develop methods for analyzing the specifications and the space of plausible models (Section 5.3): We describe algorithms for determining whether internally or externally distinguishable models exist, and for finding minimal non-ambiguous specifications. These algorithms build on our 3QBF synthesis procedure, and can potentially guide new wet-lab experiments by computing mutation experiments that disambiguate alternative models.
4. We evaluate our framework by describing that it efficiently (1) generates valid models for the *C. elegans* VPCs. The model fixes a bug in previous modeling, an incorrect modeling of a component's behavior when it is mutated; (2) shows that no behaviorally distinct models exist (even after extending the experiment space to consider mutations for each component in the VPCs), but two internally different models were synthesized, one of which expresses a previously unknown biological hypothesis; and (3) prunes the specification from forty-eight mutation experiments to a minimal set of four experiments (Section 6).

2. Technical Overview

This section presents an overview of the program synthesis and specification analysis methods we have developed for modeling biological systems. We describe how scientists typically conduct mutation experiments to infer informal genetic regulatory networks, discuss how these models can be formalized, present our programming language for expressing, verifying and synthesizing formal biological models, and outline our synthesis and specification analysis algorithms for programs in this language.

2.1 Background on Mutation Experiments.

Here we give a brief background on mutation experiments, in the context of developmental systems biology. The role of these experiments is to understand cellular genetic regulatory networks, in particular those that control stem cell differentiation. These regulatory networks are of interest in part because their failure may trigger disease:

> Cancer is fundamentally a disease of failure of regulation of tissue growth. In order for a normal cell to transform into a cancer cell, the genes which regulate cell growth and differentiation must be altered. (Wikipedia)

Hence, to understand cancer, one needs to understand cell differentiation. There are two common mechanisms for cell differentiation: (1) a single cell divides into cells of different types based on the asymmetric accumulation of substances inside the cell; and (2) multiple identical cells differentiate by mutually communicating with the goal of arriving at coordinated fates [13]. We focus on the second mechanism, and aim to mechanistically explain cell differentiation by modeling intercellular communication.

The specific goal of developmental biologists is to infer the program that stem cells "execute" to agree on their fates. This program executes within one cell division cycle during which a pluripotent cell decides its fate, potentially by communicating with other cells.

One method for inferring this program is to mutate a set of genes in the cell and observe the resulting changes in the cell development. These experiments are particularly attractive because phenotype changes resulting from the cell taking a different fate are visually observable, avoiding the need for the more expensive

tracing of temporal protein levels by the means of tagging cell proteins with fluorescent genes.

From gene mutation experiments, biologists infer protein interactions, namely which proteins are activated or inhibited by the mutated protein. For example, Yoo *et al.* [32] infers:

> In this assay, depletion of [genes] lst-2, lst-3, lst-4, or dpy-23, as well as ark-1, caused [a phenotype change, namely] ectopic vulval induction, suggesting that they function as negative regulators of the EGFR-MAPK [protein] pathway [due to the phenotype change being linked to inhibition of the pathway].

Biologists unify such piecemeal information to create informal models of cellular programs, such as the one in Figure 1 from [12]. This model shows how five cells—an anchor cell (AC), three vulval precursor cells (VPC), as well as the hyp7 cell—communicate to determine the fate of the VPCs. Each VPC contains the same set of components, which is composed of receptors (let-23 and lin12) and proteins (lst, sem-5, let-60 and mpk-1). The edges between these components show the activation (→) vs. inhibition (⊣) relationships between them.

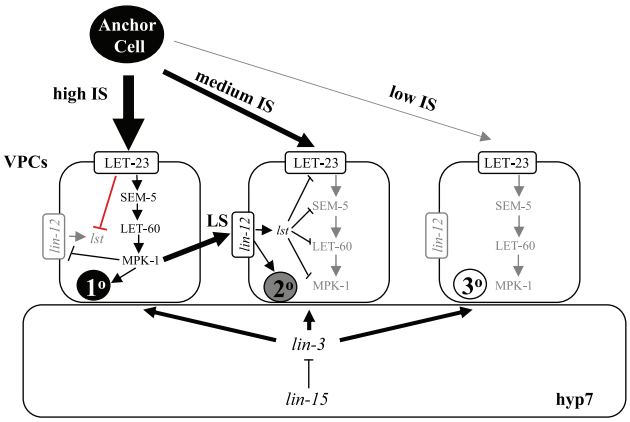


Figure 1. An informal diagram of cell fate specification in a system of three VPC cells [12]. These cells react to the inductive signal (IS) from the anchor cell and communicate among themselves using the lateral signal (LS) to decide one of three fates.

While these informal models may capture all known interactions among cell components, they do not describe the dynamics of the cell, such as what race conditions permit the cells to take non-deterministic fates that have been observed under some mutations. Due to this lack of dynamic information, one cannot be certain that these diagrams accurately describe the cell fate specification mechanism.

2.2 Executable Biology

The goal of *executable biology* [14] is to create executable models that allow the observation of the dynamic behavior of biological systems. Furthermore, these models are verified against experimental observations. For concurrent discrete models, verification, say, with model checking, ensures that all executions of the model agree with the observed outcomes. By verifying the program under the non-deterministic interleaving of cell steps, we ensure that a program is a faithful model of a cell system where cells may progress at varying rates[1] [15, 20].

It is challenging to create verifiable, concurrent models of communication between cells. To transform the informal model in Figure 1 into an executable model, the designer must model (1) protein levels; (2) timing delay or rates at which proteins react with other components; and (3) how a protein behaves when both an activator and an inhibitor of the protein are active. We have previously developed a verified model of *C. elegans* VPC cells; that model took several months to develop [12]. This paper develops methods for automatically synthesizing executable models of concurrent cellular systems.

Non-deterministic experiment outcomes. A mutation experiment may produce different outcomes when run repeatedly. A correct model must reproduce *all* non-deterministic outcomes of a given mutation experiment. We synthesize concurrent cell models that satisfy this requirement by ensuring that each outcome that must be observed is reproduced by the model under some interleaving of cell steps.

For biological reasons, we use a restricted model of concurrency, *bounded asynchrony* [15]. Because neighboring cells always advance at relatively similar rates, rather than at arbitrary speeds, fully asynchronous models are too unconstrainted to reproduce the observation in certain mutation experiments. One-bounded asynchrony is one way to achieve restricted asynchrony, ensuring that between two execution steps of a cell, no other cell can take more than two steps.

Because of the requirement to reproduce all possible outcomes, model synthesis in this setting is a more complex synthesis task than what has been previously addressed. In this paper, we advance the state-of-the-art in solving this new synthesis problem.

2.3 Modeling Language

We have developed a high-level programming model, SBL, inspired by biological diagrams such as the one in Figure 1. SBL introduces programming abstractions for cells, cell components, and interaction between components.

Programs in SBL (Figure 2) are composed of cells, which execute according to a schedule s that adheres to the *1-bounded-asynchrony* constraint. The schedule is of bounded length; the number of steps in the schedule corresponds to the desired discretization of the cell division cycle. Multiple cells can take simultaneous steps. Cells are composed of components, which model proteins or cell receptors. Components communicate with other components in the same cell or in other cells; communicating components are connected with directed edges, which correspond either to activation or inhibition relationships. Components of a cell execute synchronously; all take one step when the cell is scheduled. Components have state—a discretized concentration—usually modeled at 2-5 levels. When the component executes, it updates its next state based on its current state and the states of its activators or inhibitors. Each component is modeled with an update function $(L, L^k) \to L$, where L are levels and k is the number of components activators and inhibitors, combined.

Thanks to these abstractions, SBL programs are syntactically smaller compared to models expressed in the Reactive Modules language [1], which was the modeling language used in earlier work [12]. As a result, we are able to develop efficient synthesis algorithms for programs in SBL.

EXAMPLE 1. *To illustrate, we consider the problem of designing a simple distributed protocol. Mutations in this setting correspond to*

[1] Another way to model varying cell rates is to use stochasticity. In stochastic models [2, 24], this non-determinism takes the form of protein models making probabilistic transitions, accounting for variability of protein level change rates in nature. However, moving non-determinism from protein modeling into the scheduler allows protein models to be deterministic, which in turn enables discrete verification techniques.

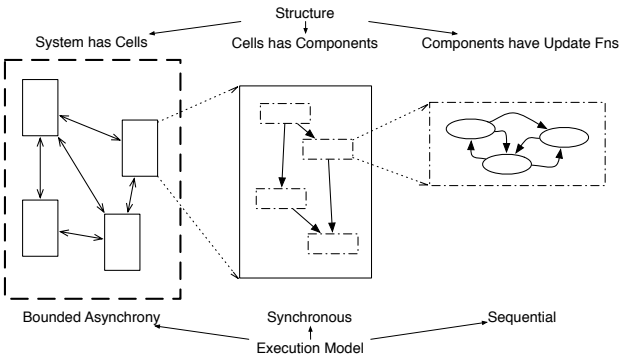


Figure 2. Hierarchical organization of programs in SBL. The system is composed of cells, which in turn are composed of components. At each time step, components update their discrete state using an update function, in terms of their previous state and incoming signals from other components. Edges between components denote which components can communicate between them. Cells group together components that always move synchronously, and they adhere to a restricted form of concurrency.

environment effects on the system being designed, and the specification consists of input-output pairs defining the desired behavior for given environments.

The goal is to design a weak consensus protocol for a three-node system. (In a biological system, these nodes would be cells, and node components would be proteins in the cells.) Two nodes (called sensors N1 and N2) are listening to a signal from a master node (a base station BS). When the base station sends a signal, at least one of the sensors must make a decision to take a measurement. When a sensor takes a measurement, it sends a release message to the other sensor permitting the other sensor not to take a measurement in order to save its power. The decision to make a measurement is made on the basis of (1) the strength from the base station; in normal conditions, the sensor that received the stronger signal should take measurement as it is closer to the base station; and (2) receiving the release message from the other signal. The environment may cause the communication between the two sensors to be down, sensors must take a measurement if no signal was received from their peer. Similar to a system of cells progressing at similar rates, we assume that sensors have bounded skew, i.e.*, they run under bounded asynchronous schedules.*

An implementation of this protocol is presented in Figure 3. Figure 3(a) presents a hierarchical view of how cell communication is organized, and which components each cell contains. On the left is the top-most level with three nodes; the base station node (BS) contains one component, the base node, which emits a constant high (H) or low (L) signal to nodes N1 and N2. These nodes decide to commit or to delegate by communicating with each other. Figure 3(b), (c) and (d) show a graphical representation of update functions for three components in nodes N1 and N2 (the remaining simpler update functions have been omitted from the figure).

2.4 Language Extensions for Verification

To make programs in our language amenable to verification, we now introduce component mutations, formalize specifications, and define a correctness condition for programs.

We model cell mutation with an adversary who perturbs the cell program such that a set of adversary-selected cell components receive adversary-supplied semantics. Typically, a cell component is mutated either to be suppressed or to stay at a high concentration level throughout the execution of the program, although we also support other mutation types.

The set of mutation experiments performed in the lab serve as our correctness specification. Let F be the set of possible outcomes of a mutation experiment. For example, if a cell can take one of three fates, the outcomes of an experiment with six cells is a six-tuple from $F = \{1, 2, 3\}^6$. Let M be the set of possible mutations that one can apply on a cell; typically, all cells involved in an experiment are mutated identically. The set of experiments *Exp* is a subset of $M \times F$, where $(m, f) \in$ *Exp* if the fate f has been observed on the mutation m. With n cell components and three possible mutations per component (e.g., no mutation; suppressed; high level), M is exponential in the number of components of the cell. As a result, biologists do not carry out all mutations.

Having an incomplete set of experiments implies that we have to accommodate partial specifications. While the set of experiments *Exp* is a subset of $M \times F$, we assume that once a mutation has been carried out, the lab has observed all possible outcomes for this mutation by repeating the experiment a sufficient number of times. This is a reasonable assumption for systems that have been reliably studied by many independent labs, such as our case study, vulval fate specification in *C. elegans* (Section 6). Without this assumption, we would have no upper bound on the specification, as any (m, f) pair could potentially be observed in experiments that have not been performed so far. The assumption allows us to synthesize with both positive examples (outcomes that must be produced by the model for an experiment) and negative ones (outcomes that must never be observed for an experiment). To model such full knowledge for a single mutation, our specification is a (partial) map $E : M \to 2^F$. The domain of E is the set of performed mutations. If $m \in dom(E) \wedge f \notin E(m)$, we assume that mutation m cannot result in fate f; the pair (m, f) is a negative example. We say that a program $P : M \to F$ is a *correct model* of E if, for each $m \in dom(E)$, the execution $P(m)$ may produce each element of $E(m)$ by controlling some aspect of the execution of P, namely the schedule that controls the concurrent execution of cells in the program.

Correctness Condition. To define a correctness condition, we view an SBL program as a function $P : (M, S) \to F$, where M and F are domains of mutations (input configurations) and fates, while S is the set of schedules adhering to bounded asynchrony. The explicit schedule allows us to formulate a correctness condition $correct(P, E)$ of a program P on a specification $E : M \to 2^F$, which has two parts:

1. *demonic scheduling:* A demonic scheduler cannot make the model produce a fate that is outside the specification, *i.e.*, $demonic(P) = \forall m \in dom(E).\forall s \in S : P(m, s) \in E(m)$.
2. *angelic scheduling:* An angelic scheduler must be able to produce each fate in the specification, *i.e.*, $angelic(P) = \forall m \in dom(E).\forall f \in E(m).\exists s \in S : P(m, s) = f$.

The demonic requirement asks that the model is an underapproximation of the specification, while the angelic requirement asks that it is an overapproximation. Angelic scheduling adds a layer of difficulty that is handled through the construction of a novel verifier (Section 5.1).

EXAMPLE 2. *The specification for Example 1, expressed as a set of experiments, is shown in Figure 4. The left column shows the mutations (environment effects) M, while the right column shows the desired outcomes F. It is interesting to note that we are using the mutations as the environment adversary; the mutations describe situations under which the nodes N1 and N2 must operate according to the expected outcomes. For example, the last row describes the situation in which the signal arriving at N1 is high, while the*

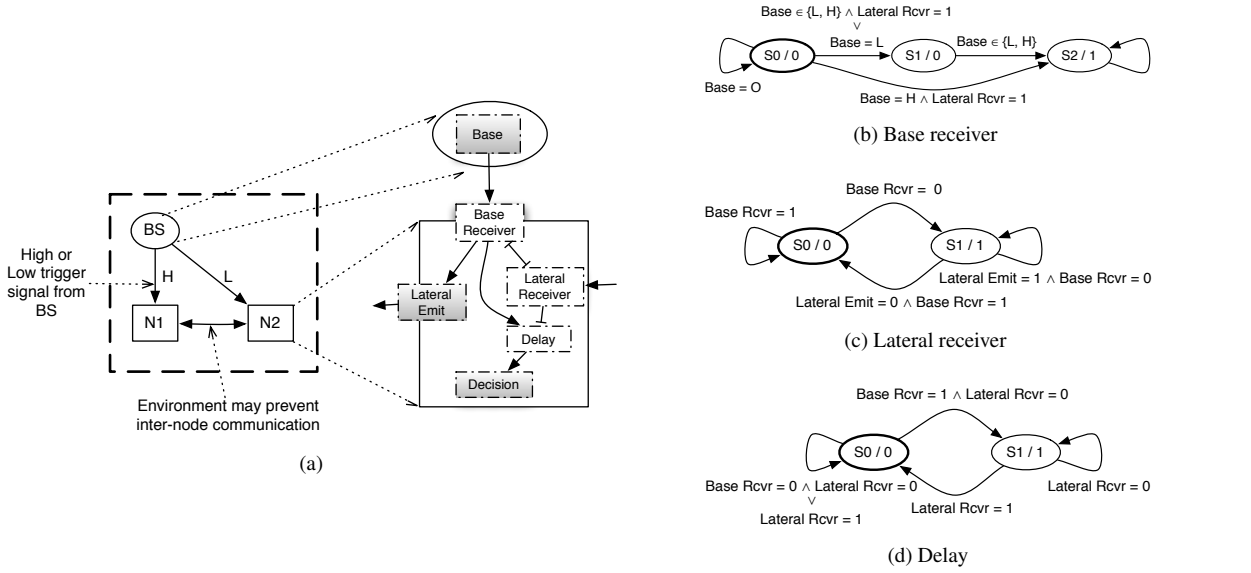


Figure 3. (a) Hierarchical view of node connections, and of their components. The top node is the base station, and the bottom nodes are distributed sensors which may not communicate with each other due to environment effects. (b), (c), (d) Graphical representation of update functions for base receiver, lateral receiver and delay components in the distributed sensors. Each state is labeled with its name and the output value that the state maps to.

Base station trigger	Inter-node comm.	N1	N2
N1=H, N2=H	Y	C	D
		D	C
		C	C
N1=L, N2=L	Y	C	D
		D	C
		C	C
N1=H, N2=L	Y	C	D
N1=L, N2=H	Y	D	C
N1=H, N2=H	N	C	C
N1=L, N2=L	N	C	C
N1=H, N2=L	N	C	C
N1=L, N2=H	N	C	C

Figure 4. The specification for the distributed protocol example, giving required outcomes for nodes N1 and N2 under a range of scenarios of base station trigger signals and cases of whether the two nodes can communicate between themselves (Y) or not (N). C = Commit, D = Delegate.

signal arriving at N2 is low, and the communication between nodes is down. We can think of this mutation as the adversary lowering the signal to N2 and preventing the communication between the two sensor nodes. The outcome C means that a node has committed to taking a measurement while D means that the measurement was delegated to the peer node.

2.5 Language Extensions for Synthesis

In order to allow synthesis of update functions in our programs, we extend our language such that these can be left unspecified. We describe partial programs in SBL and we define the synthesis problem.

The input to the synthesizer is the specification E and a *partial program* $P^?$ to be completed by the synthesizer, if feasible, into a program P^h such that the predicate $correct(P^h, E)$ holds. A partial program is a program template in which certain fragments are parameterized and need to be supplied by the synthesizer. Our language allows parameterization of (1) cell component behavior; and (2) how components communicate. Because update functions model timing delay and change rates of proteins, we found them to be the hardest part of the model to produce manually. By parameterizing update functions, we can indirectly leave unspecified also the connections between components: for example, if a biologist is unsure whether a protein P is inhibited by a protein Q or a protein R, both Q and R can be connected to P; if Q turns out not to influence P, the synthesizer is able to produce an update function for P that disregards the state of Q. The parameterized update functions are constrained to agree with the activation and inhibition semantics specified in the partial program by restricting their structure. This is achieved by stating monotonicity invariants on how a protein's input concentrations can influence its concentration; these invariants are described in Section 4.2.

From the user standpoint, the partial program $P^?$ encodes biological assumptions; it defines the components in the cells as well as a superset of connections between them. It thus (1) conveys the desire to model particular proteins and (2) states the knowledge of which (superset of) pairs of proteins communicate. Partial programs encoding biological assumptions form the basis for the ambiguity analysis described in Section 2.6.

Our synthesis problem is to find update functions h that yield a correct model:

DEFINITION 2.1 (Synthesis problem). *For a partial program $P^?$ to be completed with hole values h into P^h, the synthesis problem is to find the update functions h that yield a correct model:*

$$S(h) := \exists h : demonic(P^h) \wedge angelic(P^h)$$

A correct model must reproduce all observed experiments, and this is captured in the $angelic(P)$ correctness condition, which is a formula with two levels of quantification (2QBF). This makes the synthesis problem a 3QBF problem, while typical synthesis problems are 2QBF (of the form $\exists$ *hole* $\forall$ *input* : ϕ).

Formulas with more than one level of quantification cannot be handed off directly to an SMT solver, because the perfor-

mance of SMT solvers is only reliable for existential (one quantifier) formulas. One way to tackle 2QBF problems is to develop a counterexample-guided inductive synthesis (CEGIS) algorithm. In the classical CEGIS algorithm, an inductive synthesizer produces a program that is correct on a small sample of inputs; a verifier then checks this candidate program on remaining inputs [29]. To handle the 3QBF synthesis problem $S(h)$, we develop a novel two-part CEGIS algorithm, where an inductive synthesizer communicates with two verifiers, one for each of the two correctness conditions, and collects two kinds of counterexamples, one from each verifier (Section 5.2).

EXAMPLE 3. *The update functions for Example 1, presented in Figures 3(b), (c) and (d) are produced by our synthesizer. These update functions control how these components react to signals from the base station and the peer sensor. The synthesizer takes four seconds to generate these update functions. Intuitively, a sensor's protocol is simple: if you receive a weak signal, wait a little while and wait for the release signal from the other sensor. If it does not arrive, take a measurement. Still, even for this simple protocol, designing the update functions manually is not trivial.*

2.6 Ambiguity Analysis

Assume that a biologist produces an executable model that verifies against all performed experiments. Now imagine that after he publishes his conclusions from this model, another biologist performs a new mutation experiment whose outcome invalidates the model as well as the conclusions drawn from it. (Given a new mutation experiment m_{n+1}, a model P becomes invalid if it cannot reproduce an outcome observed for m_{n+1}, or if it produces an outcome that has never been observed after performing m_{n+1} sufficiently many times.)

Naturally, we are interested in the question of whether one can ascertain the validity of a model in the absence of complete experiments. In particular, under what assumptions can a model be considered the sole explanation of biological phenomena?

We view this question as analysis of ambiguity in the specification E, and define an *alternative model query* that answers the question. We first introduce aggregate outcomes and specification ambiguity.

DEFINITION 2.2 (Aggregate outcome). *Let P be a model and m a mutation. The* aggregate outcome *of P on m, denoted $P[m]$, is the set of outcomes produced by P mutated with m over the set S of all schedules: $P[m] := \{P(m, s) \mid \forall s \in S\}$*

A specification E is ambiguous for a partial program $P^?$ (that expresses a set of biological assumptions) if we can find two completions P^{h_1} and P^{h_2} that disagree on some new experiment. Of course, one of these models would become invalid given the new experiment.

DEFINITION 2.3 (Specification ambiguity). *Given a partial program $P^?$, a specification E is ambiguous, denoted $Amb(E, P^?)$, if $\exists m \in M \;\; \exists h_1, h_2.\; correct(P^{h_1}, E) \wedge correct(P^{h_2}, E) \wedge P^{h_1}[m] \neq P^{h_2}[m]$.*

Note that m must be a new experiment, *i.e.*, $m \in M \setminus dom(E)$, because the two models must agree with the specification E on all mutations in $dom(E)$.

In Section 6, we show that the specification for our case study is unambiguous given provided biological assumptions (*i.e.*, , there is no need for more experiments at the desired level of modeling). We also show that removing some historically important experiments indeed makes the specification ambiguous, permitting alternative explanations for coordination between cells.

DEFINITION 2.4 (Alternative model query). *Given a partial program $P^?$ stating biological assumptions and an existing (perhaps previously synthesized) model P (that need not be an instantiation of $P^?$), the* alternative model query *finds a mutation m and a new model P^h such that $P[m] \neq P^h[m]$, or shows that no such h and m exist.*

We develop an algorithm to solve this query in Section 5.3.1.

EXAMPLE 4. *We now ask whether we can find alternative models for Example 1 using the alternative model query. Suppose we relaxed the specification and do not care about the outcome on the case $N1 = L$, $N2 = L$. We ask our synthesizer to generate models under this relaxed specification such that they differ from the model in Example 1. Our synthesizer generates an alternative model that has much simpler behavior (as it need not be non-deterministic under the row that we ignored). The update functions are shown in Figure 5. When we ask for a mutation that distinguishes among the models, the synthesizer produces the omitted row. (Note that this last query is a special case of the alternative model query, such that both input programs are fully specified.)*

Now consider an experimental scenario where one wants to validate a set of experiments performed in the literature by performing them again. Is it possible to identify the smallest set of experiments whose replication is sufficient to yield a non-ambiguous specification?

To answer this question, we define a *minimization query* that computes such a minimal set.

DEFINITION 2.5 (Minimization query). *Given a non-ambiguous specification E, the* minimization query *computes a minimal non-ambiguous specification E_m from E,* i.e., *$\neg Amb(E_m, P^?) \wedge \neg\exists E', E' \subset E_m \wedge \neg Amb(E', P^?)$.*

An algorithm that solves the minimization query is presented in Section 5.3.2.

In our case study, we show that, under our assumptions $P^?$, one needs to replicate about 10% of experiments. This result suggests that computing which experiments to perform might reduce unnecessary laboratory work.

EXAMPLE 5. *We explored the minimization query for Example 1. Our synthesizer prunes E down to the first three rows of Figure 4 as a minimally unambiguous specification. This is somewhat surprising but it gives substantial insight into the problem, as the user can now understand that the specification of the four cases with lost communication was redundant, while the user may have presumed that it was necessary.*

3. Language

In this section, we present the formal semantics of our language, by first defining the language constructs, and then giving operational semantics rules for execution.

The basic construct in SBL is a *component*. We denote the set of all components in a program by $Comp$. Components are connected via a set of directed edges, defined by the relation $Edges \subseteq Comp \times Comp$. Edges model channels of communication between cell components. For each component c, we say a component c' is an *input component* of c if there is an edge $(c', c) \in Edges$. For each c, we define the set of input components $Input_c$ as $\{c' : (c', c) \in Edges\}$. A component c has a state σ_c that takes values from a finite domain L_c. Each component c is also associated with an update function, denoted f_c, that updates its state σ_c, given the current value of its input components. The function f_c has domain $L_c \times \Pi_{c' \in Input_c} L_{c'}$ and range L_c. The update

$$\textsc{Run-Program}\ \frac{S = s :: ss \qquad \bar{\bar{\sigma}}_{init} \xrightarrow{s,Cells,Edges} \bar{\bar{\sigma}}' \qquad \langle \bar{\bar{\sigma}}', ss\rangle \xrightarrow{Cells,Edges} \langle \bar{\bar{\sigma}}_{final}, [\,]\rangle}{\langle \bar{\bar{\sigma}}_{init}, S\rangle \xrightarrow{Cells,Edges} \langle \bar{\bar{\sigma}}_{final}, [\,]\rangle}$$

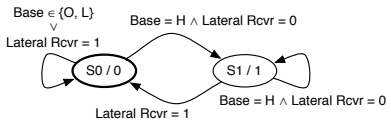
$$\textsc{Run-Program-Base}\ \frac{}{\langle \bar{\bar{\sigma}}, [\,]\rangle \xrightarrow{Cells,Edges} \langle \bar{\bar{\sigma}}, [\,]\rangle}$$

$$\textsc{Advance-Cells}\ \frac{\forall cell \in Cells\ \ \bar{\sigma}_{cell} \xrightarrow{\bar{\bar{\sigma}},cell,Edges,s(cell)} \bar{\sigma}'_{cell} \qquad \bar{\bar{\sigma}}' = \cup_{cell \in Cells}\{\bar{\sigma}'_{cell}\}}{\bar{\bar{\sigma}} \xrightarrow{s,Cells,Edges} \bar{\bar{\sigma}}''}$$

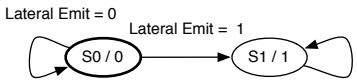
$$\textsc{Cell-Enabled}\ \frac{\forall \sigma_c \in \bar{\sigma}\ \ \sigma_c \xrightarrow{c,\bar{\bar{\sigma}},Edges} \sigma'_c \qquad \bar{\sigma}' = \cup_{c \in cell}\{\sigma'_c\}}{\bar{\sigma} \xrightarrow{\bar{\bar{\sigma}},cell,Edges,1} \bar{\sigma}'} \qquad \textsc{Cell-Disabled}\ \frac{}{\bar{\sigma} \xrightarrow{\bar{\bar{\sigma}},cell,Edges,0} \bar{\sigma}}$$

$$\textsc{Advance-Component}\ \frac{\Sigma = \{\sigma_{c'} : (c', c) \in Edges\} \qquad f_c(\Sigma, \sigma) = \sigma'}{\sigma \xrightarrow{c,\bar{\bar{\sigma}},Edges} \sigma'}$$

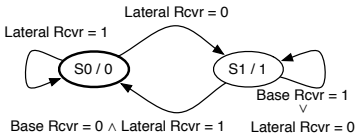
Figure 6. Small-step semantics for program execution. RUN-PROGRAM runs a schedule by advancing the cells according to each micro-step in the schedule, with RUN-PROGRAM-BASE as the base case. ADVANCE-CELLS rule updates the states of cells, depending on the current micro-step s. If a cell is enabled, it is advanced by applying the CELL-ENABLED rule. Conversely, if a cell is disabled, the CELL-DISABLED rule keeps its state unchanged. ADVANCE-NODE rule updates the state of a component by invoking the update function on the states of all input component states and its own state



(a) Base receiver



(b) Lateral receiver



(c) Delay

Figure 5. Update functions generated using the alternative query model to differentiate from the model in Example 1 under ambiguous specification, obtained by removing row 2 of Figure 4.

function for a component is chosen from a sequence of functions $F_c := [f_{c,1}, \ldots, f_{c,k}]$ that describe possible alternative behaviors of that component under different mutations, i.e., the natural and altered behaviors of the component.

A *cell* is a set of components. Within a cell, we have a synchronous execution model, i.e. all components of a cell update their state simultaneously. The state of a cell $\bar{\sigma}$ is defined as the set of states of the components that the cell contains. We denote the set of all cells in a program by $Cells$. $Cells$ forms a partition on all the components in the program. A pair of cells $(cell_1, cell_2)$ are said to be *communicating* if there exists a pair of components $(comp_1, comp_2)$ connected by an edge in the respective cells.

The pair $(Cells, Edges)$ constitutes a *program*. The program state $\bar{\bar{\sigma}}$ is the set of all cell states in the program. The input to a program is a *configuration* (*i.e.*, a mutation). A configuration is a function from components to integers, that expresses for each component c the index of the function in F_c that should be used as the update function f_c. The output of a program is defined as the state of user-designated components in the final state reached in an execution.

Partial Programs. The sequence F_c of functions associated with component c need not be specified concretely. When at least one component function is not concretely specified, we say the program is *partial*. Typically, users will only concretely specify the behaviors under well-understood mutations that would not make sense to redefine. For example, a typical example in the biological case is the knock-out mutation which subdues the function of the component and fixes it to the OFF state.

Operational semantics Figure 6 shows the small-step semantic rules for program execution. Here, we assume that the program starts in the initial state $\bar{\bar{\sigma}}_{init}$, and that it has already been preprocessed by fixing a particular update function for each component according to the input configuration. The semantics are defined recursing down the program structure. The RUN-PROGRAM rule executes the program by moving all cells in accordance with a schedule S. The ADVANCE-CELLS rule captures the intuition that each schedule step s partitions the cells into the sets $enabled$ (for which $s(cell) = 1$) and $disabled$ (for which $s(cell) = 0$). Rules CELL-ENABLED and CELL-DISABLED describe how cell states are updated for enabled and disabled cells, respectively. For the disabled cells, the state remains unchanged. Enabled cells are advanced by applying the ADVANCE-COMPONENT rule for each component, which corresponds to updating the component state by reading the state of connected neighbors and using the component's update function.

Bounded Asynchrony. The concurrency notion that our execution model admits is bounded asynchrony. This model faithfully represents biological systems where complete synchrony is too

strict, and complete asynchrony does not accurately model cells that progress at similar but not identical rates.

Fisher et al. [15] define bounded asynchrony with schedules consisting of micro- and macro-steps. Each micro-step consists of a subset of the components stepping synchronously. This is what we have been calling a schedule up to this point. Next we block micro-steps together into a macro-step. Each k-bounded macro-step consists of all components taking k steps split across multiple micro-steps. For example, let us consider three nodes and the schedule 110 (micro-step) indicates the first two take a step while the third waits. Suppose the second schedule is the micro-step 001. Then the two micro-steps together make a macro-step in which all nodes take one step and which is therefore 1-bounded.

Schedules over micro-steps are much more expensive to enumerate than schedules over macro-steps, especially 1-bounded macro-steps. Schedules over 1-bounded macro-steps (where each node necessarily moves once), can be succinctly encoded without loss of information as pairwise happens-before between connected nodes. That is, a 1-bounded macro-schedule is an assignment of $<$, $>$, or $=$ to each edge in the node topology[2]. The following lemma holds:

LEMMA 1 (Fisher et al. [15]). *A micro-schedule exists if and only if a realizable macro-schedule exists over the node topology.*

Here a realizable macro-schedule is one that does not cause an inconsistent ordering of nodes in a cycle. We use this result critically to efficiently encode partial programs as formulas (Section 4), and restrict schedules to be 1-bounded.

Using macro-steps allows us to define a compact symbolic encoding of our programs into formulas, which would have not been possible with micro-steps.

4. Translating Programs into Formulas

We now describe how to translate execution of SBL programs to SMT formulas, enabling verification and synthesis. We first give rewrite rules that construct a formula corresponding to the symbolic execution of a program. We then describe additional constraints that encode biological domain knowledge to be used in synthesis of programs.

4.1 Translation of Program Execution

The translation of program execution is parameterized by the following symbolic variables:

- For each time step t and each pair of connected cells (c_1, c_2), we define a channel configuration variable $channel_{t,c_1,c_2}$ that must hold exactly one of the three values "$<$", "$>$" and "$=$". These variables encode the symbolic schedule for program execution. Variables $channel_{t,c_1,c_2}$ and $channel_{t,c2,c1}$ are asserted to be consistent in the following way:

$$\begin{aligned} channel_{t,c1,c2} = \text{“}>\text{”} &\Leftrightarrow channel_{t,c2,c1} = \text{“}<\text{”} \\ &\wedge \\ channel_{t,c1,c2} = \text{“}<\text{”} &\Leftrightarrow channel_{t,c2,c1} = \text{“}>\text{”} \\ &\wedge \\ channel_{t,c1,c2} = \text{“}=\text{”} &\Leftrightarrow channel_{t,c2,c1} = \text{“}=\text{”} \end{aligned}$$

- For each component c, we represent each function $f_i \in F_c$ as a lookup table with symbolic values for each value in its domain $L_c \times \Pi_{c' \in Input_c} L_{c'}$. Entries of the lookup table are represented by the variables $table_{v_c, v_{c_1}, \ldots, v_{c_n}}$ that take values in L_c.
- For each component c, we represent its mutation symbolically as a variable m_c, that encodes the index of the function to use among F_c. If m_c has value i, then the function $f_{c,i}$ will be used as the update function of component c.
- For each component c at each execution step, we create a variable $\sigma_{t,c}$ that takes values in the domain of L_c. These variables represent the component state symbolically over the execution of the program.

[2] Technically, for micro-steps it is the sequence of ordered bell numbers or Fubini numbers [16], while for 1-bounded macro-steps it is $3^{\text{num edges}}$.

Translation rules for compiling program execution to an SMT formula are shown in Figure 7.

T_{RUN} is the top-level rule for translating the execution of a program, unrolling the execution for k steps. T_{READ} uses the symbolic macro-schedule values to assert the input states that should be read by each component at a given macro-step. If a cell $cell$ runs before or at the same time as another cell $cell'$ (i.e. the macro-step variable between the two cells at a time step has value "$<$", or "$=$"), the components in $cell$ reads their input states from $cell'$ at the previous time step. On the other hand, if $cell$ runs after $cell'$, it reads its input states from the current time step. Finally, T_{UPDATE} asserts that the state of a component is updated in terms of its symbolic mutation, input state, own state, and update function. The outermost conjunction enumerates over possible update functions. For each mutation, the inner conjunction enumerates possible input value tuples of the update function. The symbolic state is updated given symbolic lookup variables for the chosen mutation.

This translation does not impose any constraints on the parameterized update functions, and therefore encodes a very large space of possible update functions. To help with the program synthesis task, we need to restrict this space. This is achieved in Section 4.2 by asserting biologically motivated constraints on the structure of the parameterized update functions.

4.2 Domain-Specific Constraints on Update Functions

The translation in Section 4.1 does not impose restrictions on the structure of the update functions that are left unspecified by the user. When modeling biological systems, formulating a hypothesis typically involves stating high-level invariants about whether a component *activates or* inhibits another one. In this section, we describe how the space of update functions is restricted using this high-level knowledge.

We first formalize how the high-level biological invariants are stated by defining a partial labeling of edges with activation and inhibition semantics.

DEFINITION 4.1 (Edge labeling). *Given a partial program $P^?$, the partial function label $: Edges \to \{activating, inhibiting\}$ annotates edges in $P^?$ as either activating or inhibiting.*

As a component's state expresses its activation level, we assume the existence of a total order on its possible states. This will allow us to state the properties that restrict the space of update functions.

DEFINITION 4.2 (State ordering). *Let c be a component, and L_c the set of possible state values for c. The state ordering $\leq_c$ is a total order on L_c.*

Using the edge labeling and the state ordering for each component, we now define a partial order on the combined input values for a component.

DEFINITION 4.3 (Input ordering). *Given a component c with update function $f_c : L_c \times L_{c_1} \times \ldots \times L_{c_n} \to L_c$, the partial order $\preceq_c$ on elements of $L_{c_1} \times \ldots \times L_{c_n}$ is defined as:*

$$\begin{aligned} &(v_1, \ldots, v_n) \preceq_c (u_1, \ldots, u_n) \\ &\quad := \forall i \in \{1, \ldots, n\}. \\ &\qquad (label((c_i, c)) = activating \wedge v_i \leq_{c_i} u_i) \vee \\ &\qquad (label((c_i, c)) = inhibiting \wedge v_i \geq_{c_i} u_i) \end{aligned}$$

$$\mathrm{T}_{\mathrm{RUN}}[\![Cells, Edges]\!] := \bigwedge_{t\in\{1,\ldots,k\}} \bigwedge_{cell\in Cells} \bigwedge_{c\in cell} \mathrm{T}_{\mathrm{READ}}[\![t, c, Edges]\!] \wedge \mathrm{T}_{\mathrm{UPDATE}}[\![t, c, Edges]\!]$$

$$\mathrm{T}_{\mathrm{READ}}[\![t, c, Edges]\!] := \bigwedge_{\substack{(c',c)\in Edges \\ c'\in cell' \\ c\in cell}} \begin{pmatrix} ((channel_{t,cell',cell}=\text{“<”})\Rightarrow(\sigma_{read,t,c,c'}=\sigma_{t-1,c'})) \\ \wedge \\ ((channel_{t,cell',cell}=\text{“=”})\Rightarrow(\sigma_{read,t,c,c'}=\sigma_{t-1,c'})) \\ \wedge \\ ((channel_{t,cell',cell}=\text{“>”})\Rightarrow(\sigma_{read,t,c,c'}=\sigma_{t,c'})) \end{pmatrix}$$

$$\mathrm{T}_{\mathrm{UPDATE}}[\![t, c, Edges]\!] := \bigwedge_{f_i\in F_c} m_c = i \Rightarrow \left(\bigwedge_{(v_c,v_{c_1},\ldots,v_{c_n})\in dom(f_i)} \begin{matrix} (\sigma_{t-1,c},\sigma_{read,t,c_1,c},\ldots,\sigma_{read,t,c_n,c})=(v_c,v_{c_1},\ldots,v_{c_n}) \\ \Rightarrow \\ \sigma_{t,c}=table_{v_c,v_{c_1},\ldots,v_{c_n}} \end{matrix} \right)$$

Figure 7. Translation rules for symbolic execution of programs.

Intuitively, $\preceq$ is a partial order on the strength of the input values to a component, based on the activation and inhibition annotations. We now describe two kinds of invariants that restrict the space of possible update functions.

Input monotonicity Our first property is motivated by the following observation: If there is an activating edge from component c_1 to component c_2, then an increase in σ_{c_1} should not have by itself the effect of decreasing σ_{c_2}. Conversely, if c_1 and c_2 are connected through an inhibiting edge, then a decrease in the value of σ_{c_1} should not result by itself in the decrease of σ_{c_2}:

$$\forall i_1, i_2 \in L_{c_1} \times \ldots \times L_{c_n}.\, \forall v \in L_c.\\ i_1 \preceq_c i_2 \Rightarrow f_c(v, i_1) \leq_c f_c(v, i_2)$$

State monotonicity The second property that we assert imposes a monotonicity constraint on f_c in terms of the value of σ_c. This property expresses that, for the same input value, a greater the activation level of the component cannot be updated to a smaller value:

$$\forall i \in L_{c_1} \times \ldots \times L_{c_n}.\, \forall v_1, v_2 \in L_c.\\ v_1 \leq_c v_2 \Rightarrow f_c(v_1, i) \leq_c f_c(v_2, i)$$

We found that asserting constraints that encode these two invariants based on user annotations on component connections is crucial for ensuring that the structure of update functions agree with existing biological knowledge.

5. Synthesis and Querying Spaces of Models

In section 4, we described how we translate program execution to formulas. In this section, we present algorithms that leverage this translation for verification and synthesis, as well as specification ambiguity analysis.

The formula that encodes program execution parameterizes (1) update functions (which are the holes in partial programs); (2) schedules; and (3) input configurations (*i.e.*, mutations). The space for update functions and the space for schedules are typically very large. However, specifications are typically wet-lab experiments which are sparse and inherently small (of order 10^2 experiments). Based on this observation, we develop algorithms that unroll quantifications for input configurations only.

In the following, we refer to the symbolic output parameter of translating the execution of P with input m and schedule s as $P(m, s)$, and we denote by E the specification (given as a partial function from M to 2^F).

5.1 Verifying Programs

The correctness condition presented in Section 2.4 is defined as:

$$correct(P) := demonic(P) \wedge angelic(P)$$

The properties $demonic(P)$ and $angelic(P)$ are in 1QBF and 2QBF respectively. As a result, the correctness condition $correct(P)$ is in 2QBF.

We verify correctness conditions $demonic(P)$ and $angelic(P)$ separately, using a verifier V_d that searches for demonic schedules that lead to the violation of the specification, and a verifier V_a that checks whether all non-deterministic outcomes for a given mutation can be reached for some angelic schedule.

Verifying for demonic schedules. The formula $demonic(P)$ states that the set $E(m)$ is an upper bound for all observed outcomes of P with input m:

$$demonic(P) := \forall m \in dom(E).\, \forall s \in S.\, P(m, s) \in E(m)$$

To check this property, we attempt to disprove it by searching for a demonic schedule that produces an unobserved outcome for an input in $dom(E)$, the domain of E. Given the observation that there is a small set of input values in $dom(E)$, we solve this formula by unrolling the existential quantification over this set, and by querying symbolically for a demonic schedule. The condition $P(m, s) \notin E(m)$ is expressed by unrolling over values in $E(m)$, which is also a small set. We thus solve the 1QBF formula:

$$\bigvee_{m\in dom(E)} \exists s. \bigwedge_{f\in E(m)} P(m, s) \neq f$$

If this formula is satisfiable, P does not satisfy $demonic$, and we obtain a concrete counterexample (m, s) such that running P on input m and schedule s leads to an unobserved fate. If it is unsatisfiable, then P is correct with respect to $demonic$.

Verifying for angelic schedules. The $angelic$ condition states that all outcomes in the set that m maps to must be observable, i.e. appear in some execution of P on m:

$$angelic(P) := \forall m \in dom(E),\ f \in E(m).\, \exists s.\, P(m, s) = f$$

This amounts to searching for an angelic schedule for each $f \in E(m)$. We reduce the 2QBF correctness property to an efficiently solvable 1QBF problem by unrolling values of the domain $dom(E)$, again based on the assumption that this is a small domain. To unroll $angelic(P)$, we construct the following query for each $m \in dom(E)$ and for each $f \in E(m)$:

$$\exists s.\, P(m, s) = f$$

If the above formula is unsatisfiable for some m and f, then no angelic schedule can be found for reaching that outcome when running P, and (m, f) is a counterexample input/output pair witnessing that $angelic(P)$ does not hold. If the formula is satisfiable for each $m \in dom(E)$ and for each $f \in E(m)$, then verification for angelic schedules succeeds.

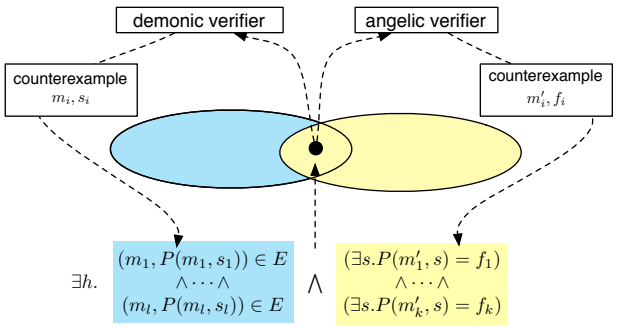


Figure 8. The synthesizer consists of three communicating solvers. The two verifiers generate two kinds of counterexamples, and the synthesizer generates models that satisfy the constraints for all counterexamples.

A program verifies against the specification E if it verifies against both V_d and V_a.

5.2 Synthesizing Programs

In our language, it is possible to define a partial program $P^?$ that admits freedom in the update functions of its components. We now present a synthesis algorithm for finding update functions in $P^?$ such that the completed program P^h is correct with respect to the correctness condition $correct(P)$. Our procedure leverages the two verifiers V_d and V_a to check correctness properties $demonic$ and $angelic$ respectively, in order to solve the following synthesis problem:

$$S(h) := \exists h.\ demonic(P^h) \wedge angelic(P^h)$$

This formula is in 3QBF, due to the quantifier alternation $\exists\forall\exists$ resulting from $angelic(P^h)$ being nested within the quantification over h.

We solve $S(h)$ by developing a counterexample-guided inductive synthesis (CEGIS) algorithm, which decomposes the 3QBF problem into two 1QBF solvers (an inductive synthesizer and the demonic verifier V_d) and one 2QBF solver (the angelic verifier V_a). The inductive synthesizer produces a candidate model that is correct on all counterexamples and sends this model to both verifiers. If both approve the model, the synthesis successfully terminates. If either fails, counterexamples are produced, refining the correctness constraints placed on the inductive synthesizer, making it eventually produce a correct model (or conclude that no model exists in the model space described by $P^?$). The solver architecture is shown in Figure 8.

Precisely, the synthesizer maintains two sets of counterexamples, $CE_1 \subseteq dom(E) \times S$ and $CE_2 \subseteq dom(E) \times F$. The first set contains pairs of inputs and schedules, and is computed with counterexamples given by the verifier for demonic schedules. The second one is a subset of the input/output specifications, and is in turn computed with counterexamples found by the verifier for angelic schedules. Starting with initial sets CE_1 and CE_2, the synthesizer solves at each step the following formula to find a candidate model:

$$\exists h. \quad \left(\bigwedge_{(m,s)\in CE_1} P^h(m,s) \in E(m) \right) \wedge \left(\bigwedge_{(m,f)\in CE_2} \exists s.\ P^h(m,s) = f \right)$$

If the above formula is unsatisfiable, the partial program cannot be completed, i.e. synthesis fails. Otherwise, the valuation of h defines a candidate model that we attempt to verify using verifiers V_d and V_a. If at least one of the verifiers returns with a counterexample, the synthesizer attempts to find a new candidate after updating the sets CE_1 and CE_2 with the counterexamples returned by either verifier. If a candidate model is validated by both verifiers, we obtain the completed program P^h that is correct with respect to the specification E.

5.3 Querying for Ambiguity Analysis

Given the above procedure for synthesizing programs, we are now interested in querying spaces of possible models. In particular, we analyze ambiguity of specifications. If a specification is underspecified, we aim to reduce ambiguity by expanding it. If, on the other hand, it is overspecified, our goal is to reduce the specification size without introducing ambiguity.

Computing Aggregate Outcome We first give an iterative algorithm to find aggregate outcome set for a given program p and a given input m. The aggregate outcome set $P[m]$ is the set of outcomes of P on m over all schedules. We approach the task by first computing the outcome of P on m under an initial schedule s. We then enlarge the set of observed outcomes Obs by searching for a schedule leading the program to produce a previously unseen outcome. To find such an outcome, we build a formula that states that the new outcome must differ from each value in the Obs set inferred so far. Each step of the algorithm thus attempts to extend Obs by solving the following formula:

$$\exists s. \bigwedge_{f\in Obs} P(m,s) \neq f$$

If this formula is satisfiable, we obtain an outcome that we add to the set Obs, and then attempt to solve the formula with the updated set. If it is unsatisfiable, we have obtained all outcomes that can be produced by P on input m.

5.3.1 Alternative Models

To ascertain that a given hypothesis is the sole explanation to a biological phenomenon, a biologist would like to learn whether there exists another hypothesis that differs from the first on its observable outcome on an unperformed experiment, but is correct on the known experiments. Given a program P_1 that expresses the first hypothesis, and a partial program $P_2^?$ that expresses a space of alternatives for the second, we can state this query formally as:

$$\exists m.\exists h.correct(P_2^h) \wedge P_2^h[m] \neq P_1[m]$$

If this query is satisfiable, then there is an alternative program P_2^h and a new experiment m such that performing the experiment m will invalidate at least one of P_1 and P_2^h. We now describe an algorithm to solve this query.

Given the hypothesis that the space of mutation experiments M is small, we approach this task by unrolling the existential quantification over m. The problem then reduces to synthesizing P_2^h for a given mutation m, such that $P_2^h[m] \neq P_1[m]$.

$P_2^h[m]$ can differ from $P_1[m]$ in two distinct ways: (1) It can either contain an output value not in $P_1[m]$; or (2) it can be a strict subset of $P_1[m]$. We give one algorithm for each case.

Case 1. A program P_2^h that produces an outcome not seen in $P_1[m]$ can be found by augmenting the synthesis query described in Section 5.2 with a constraint asserting that there exists a schedule that leads P_2^h to produce an outcome not in $P_1[m]$, *i.e.*, $P_2^h[m] \setminus P_1[m] \neq \emptyset$. We solve the following formula to answer this query:

$$\exists h.\ correct(P_2^h) \wedge \exists s.P_2^h(m,s) \notin P[m]$$

This formula is satisfiable if and only if there exists a completion of program P_2^h that produces an outcome not in $P[m]$. It is 3QBF, and is handled using the mechanism of a synthesizer communicating with two verifiers to perform inductive synthesis described in Section 5.2.

Case 2. Alternatively, P_2^h may be found by attempting to synthesize a model that always produces outcomes in a strict subset of $P_1[m]$. This is achieved by discarding elements of $P_1[m]$ one at a time, to see if such a model can be found. We do not need consider all subsets of $P_1[m]$, as we only state that $P_1[m] \setminus \{f\}$ is only an upper bound of the possible outcomes for input m.

$$\exists h. \qquad correct(P_2^h) \wedge \left(\bigvee_{f \in P_1[m]} \forall s.\ P_2^h(i,s) \in P_1[m] \setminus \{f\}\right)$$

This formula is satisfiable if and only if there exists a completion of program P_2^h such that its observable outcome set is a strict subset of observable outcomes of P_1 on input m. Similarly to Case 1, we use the scheme of cooperating solvers described in Section 5.2 to solve this formula.

5.3.2 Minimization

In a context where performing experiments is an expensive process, a researcher may want to obtain a minimal non-ambiguous specification that sufficiently constrains the space of models to validate a hypothesis. Given a partial program $P^?$ that expresses a hypothesis, and a specification E that is non-ambiguous with respect to $P^?$, the task of finding a minimal non-ambiguous specification E_m is stated as:

$$\neg Amb(E_m, P^?) \wedge \neg\exists E', E' \subset E_m \wedge \neg Amb(E', P^?)$$

We compute a minimal specification E_m by iteratively restricting the domain of E for the partial program $P^?$. This can be done by invoking the *alternative model* query once for each mutation in $dom(E)$.

At each step, we check whether program $P^?$ can be completed to a program P^h that decides a set of outputs $P^h[m]$ distinct from $E(m)$, considering as specification the set of currently non-redundant input values. This check is performed using the *alternative model query* described in Section 5.3.1. If synthesis fails, m is marked as redundant. Otherwise, removing m from the specification leads to ambiguity, and as a result m should be kept in the final set of pruned inputs. Upon considering all inputs in the domain of E, a minimal specification is obtained by removing from the domain of E those inputs that are marked as redundant.

6. Case Study: *C. elegans* vulval development

We attempt to synthesize a model for the vulval precursor cells (VPCs) that start off identical but through coordination among themselves and with the Anchor Cell (AC) agree on specific fates. From informal descriptions of protein interactions found in biological literature, we develop our template $VPC^?$. The template is shown in Figure 9(a) (derived from Figure 1.)

From the template, we observed that there are nodes with extremely simplistic on-off behavior. These are LS, the downstream nodes of the cascade (sem5, let60, and mpk1) and the fate nodes. While we can introduce holes in them (with expected performance degradation, yet not being intractable), biologists have a very clear understanding of these nodes, and so expect to see a simple and known behavior in them. Additionally, introducing holes in these nodes leaves too much freedom to the synthesizer, such that generated models do not have a biological interpretation.

Therefore we run our tool with unknown update functions for lin12, let23, and lst. The generated update functions satisfy the specification and template structure of the program. On the other hand, lin12, which has a very well understood behavior, colludes with the other components to give models that are hard to explain to the biologists. Therefore, we additionally allow the user to specify the behavior of lin12 concretely and synthesize let23 and lst. Let23 and lst are indeed the most complex functions in their timing delays (and have the most complex interconnection dependences). Indeed, in our attempts prior to synthesis (when designing the verifier) to write the model by hand, we actually failed. Additionally, the models previously written did not maintain the requisite lin12 behavior. Therefore, our synthesizer was solving a problem that had been impossible to solve manually, even after considerable effort.

The specification consists of forty-eight experimental observations of the fate outcomes of six VPC cells in sequence. Some of these observation have non-deterministic outcome fates. A fragment of the specification is shown in Figure 9(b).

From the template and these experiments, our synthesizer generated update function solutions to let23 and lst that were confirmed by the biologists to be plausible behaviors. The output from the synthesizer is shown in Figure 10(a).

It is important to note that this is a very significant achievement. Previously, when we had written down a model for VPCs in RM [12] it had the following flaws: (1) The previous model did not satisfy a biological invariant required on the lin12 component, and all efforts to fix the model failed, (2) RM is too expressive and therefore there were cases where the model "read the future" which was hard to interpret biologically, (3) the model lacked readability prohibiting debugging, extension, and biological interpretation. Our synthesized alternative model solves all these. Our first biologically relevant result is therefore that through synthesis we have revalidated the (experimentally-confirmed) prediction from previous work, *without the vagaries of human modeling*.

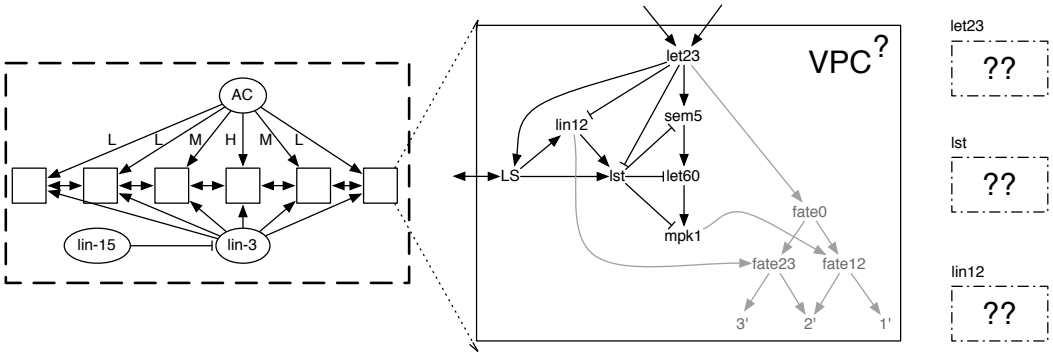
6.0.3 Specification ambiguity for *C. elegans* VPC models

Next, we analyzed the ambiguity in the specification. The important biological unknown is the specific node within the cascade let23-sem5-let60-mpk1 that sends out the inhibitory signal to lin12 and lst. We attempted experimenting with all four options under our definition of understanding the specification ambiguity:

Alternative models for particular input configuration 44 of the 48 experimental observations are deterministic. We wanted to know how many models exist if only the deterministic outcomes are asserted. We found that under this relaxed specification, *all* four options of inhibition coming from any node of the cascade work.

Then using the alternative model query from Section 5.3, we asked for a model including any one of the four remaining outcomes. The synthesizer eliminates two that have inhibition emanating from let60 and mpk1. This was significant since it formally confirmed the biologist's intuition that the inhibition comes from higher up in the cascade. Additionally, it showed that sem5 (in addition to let23, which was conjectured earlier) was a valid possibility for the inhibitor.

Input configuration for disambiguating models Next, we attempted to observationally distinguish these two remaining valid models. Our 48 observations mutate the entire cascade (all nodes let23 to mpk1) together. We wanted to infer if a finer-grained mutation exists that distinguishes these two remaining mechanistic hypotheses. We expanded the experimental set by enumerating all possibilities of the cascade nodes (2^4 possibilities of expansion for each of the 48 rows) leading to 384 experiments. Our synthesizer shows that *no other mutations exist* that would observationally distinguish these two hypotheses. This saves the biologist significant effort (336 experiments, each of which are expensive and time-consuming) as they now know that mutation experiments will not



Exp#	Mutations					Fate pattern					
	AC	lin12	lin15	Vul	lst	P3.p	P4.p	P5.p	P6.p	P7.p	P8.p
1	Formed	wt	wt	wt	wt	3	3	2	1	2	3
5	Formed	wt	ko	wt	wt	1/2	1/2	2	1	2	1/2
7	Formed	wt	ko	ko	wt	3	3	3	3	3	3
13	Formed	ko	ko	wt	wt	1	1	1	1	1	1

Figure 9. (a) The template $VPC^?$ we use for our experiments, which is derived as simply the union of connections known to biologists [12] as informally shown by Figure 1. The "fate" nodes are instrumentation nodes to help read out the outcome. (b) A small fraction of the specification E (4 rows out of 48), obtained from literature in biology [12]. A fate pattern of $1/2$ indicates that both 1 and 2 are outcomes observed in experiments.

suffice to distinguish these explanations and out-of-band experiments need to be performed.

Inferring the minimal specification We run our minimization query from Section 5.3 for each of the VPC queries, with significant results. We infer that, for the space we are searching over, only four experimental observations suffice to yield a unique model. This set contains all non-deterministic outcomes, and additionally others that together constrain the system enough to yield the unique model that is explained by the 48 experiments.

Wet-lab predictions Our exploration demonstrated that (1) let23 is not the only possibility for inhibition, but sem5 is as well; (2) let60 and mpk1 cannot play that role; and (3) the models using let23 and sem5 cannot be distinguished observationally. These suggest a possible inhibition from sem5, that cannot be distinguished through mutation experiments on the components included in our model, therefore other types of experiments would need to be done.

7. Performance evaluation

We implemented our language as an embedded DSL in Scala. Our synthesis and analysis framework, also implemented in Scala, uses the Z3 theorem prover [10] as its underlying constraint solver. We interface with Z3 through the Scala^{Z3} library [22]. Our framework consists of 5K lines of code.

We show performance results for the evaluation of our synthesis procedure in Figure 11(a). For each example, we present total execution time, maximum memory usage, number of calls to the underlying SMT solver Z3, average call time, the structure of holes in the partial programs, as well as the search space for synthesizing update functions. VPC1, VPC2, VPC3 and VPC4 are models of the fate decision in *C. elegans* vulval precursor cell development that express each different biological hypotheses about the cells through their topology. VPC1 and VPC2 are synthesized using a specification E with domain size 48, while VPC3 and VPC4 are synthesized using a specification E' whose domain is restricted to 44 elements. Sensors is the example introduced in Section 2. For each example, we report the total running time for synthesis, the maximum memory usage, number of calls to the underlying SMT solver Z3, the average time Z3 takes to solve these queries, a description of holes in the partial program as a sequence of number of states for each unspecified update function, and the size of the search space for synthesizing these functions.

In all cases, we find that even for a complicated synthesis problem such as the VPCs, our synthesizer is efficient.

In Figure 11(b), we present performance results for the pruning procedure described in Section 5.3.2. We report the domain size for the result of the procedure and the initial domain size in the *pruned/total* column.

As expected, the time for pruning is significantly higher than for only synthesis. This is because multiple synthesis and verification queries are solved in the process of minimization. However, compared to the amount of time this could potentially save the biologists, *i.e.*, months or even years of work in doing redundant experiments, our inference times are insignificant.

8. Related Work

Inference of biological models While model checking of (manually written) logical biological models has been an active area of research, we are not aware of work that synthesizes these models. In contrast, a growing body of literature exists on inference of non-logical models. The first class of such models uses ordinary differential equations (ODEs). An example of ODE model inference from temporal and spatial data is the work by Aswani *et al.*, who reduce the amount of prior knowledge needed to infer an accurate model [3]. Rizk *et al.* find parameters for ODE models by optimizing a notion of continuous degree of satisfaction of temporal logic formulas [27]. Because ODE models are continuous, these techniques do not appear directly applicable for inference of logical models based on concurrent systems.

Machine learning has also been used to infer biological models. Barker *et al.* use time series data of protein levels to infer whether a protein is an activator or a suppressor of another protein [4]. Time series data of concentrations is not available in our setting, so these approaches do not apply to the inference of our models.

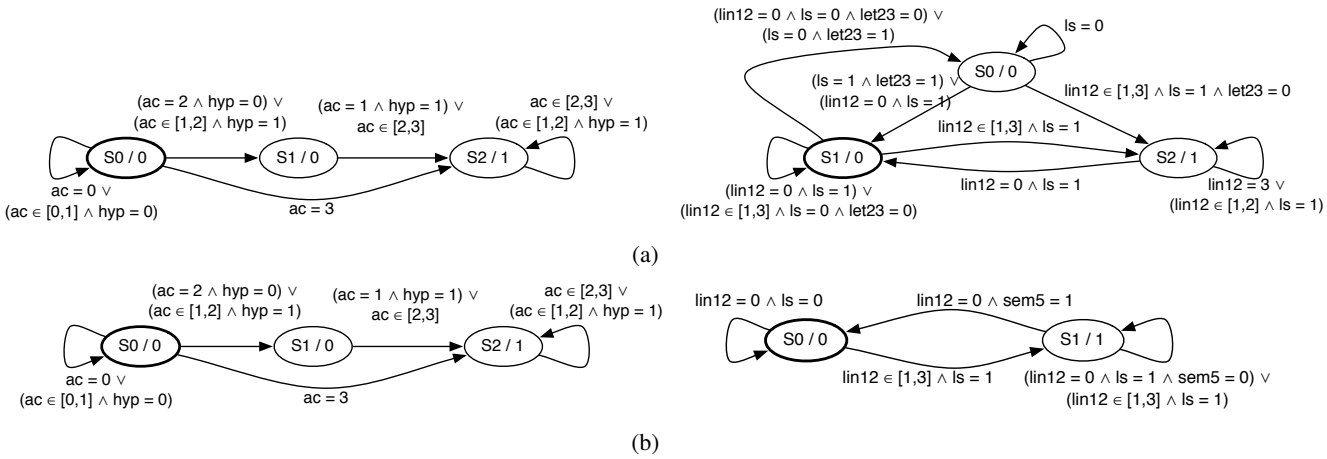


Figure 10. Synthesized update functions given two different connection topologies, for let23, and lst. (a) The topology with lst and lin12 inhibited by let23. The template allows for three let23 states and three lst states. (b) The topology with lst and lin12 inhibited by sem5. The template allows for three let23 states and two lst states.

example	time	mem.	# calls	$\frac{\text{time}}{\#\text{ calls}}$	holes	search space
VPC1	96.64	2.34	282	0.09	$(3, 3)$	$2.25 \cdot 10^{34}$
VPC2	87.77	2.33	285	0.08	$(3, 2)$	$1.21 \cdot 10^{21}$
VPC3	48.29	0.77	139	0.10	$(4, 3)$	$1.47 \cdot 10^{42}$
VPC4	49.18	1.26	133	0.09	$(5, 3)$	$7.25 \cdot 10^{50}$
Sensors	4.30	2.40	51	0.01	$(3, 2, 2)$	$2.53 \cdot 10^{13}$

(a)

example	time	mem.	# calls	$\frac{\text{time}}{\#\text{ calls}}$	$\frac{\text{pruned}}{total}$
VPC1	2964.82	2.20	3805	0.54	$4/48$
VPC2	1845.94	1.69	3544	0.31	$3/48$
VPC3	273.77	1.31	491	0.29	$4/44$
VPC4	316.32	1.35	482	0.37	$4/44$
Sensors	14.46	0.71	167	0.04	$3/8$

(b)

Figure 11. All times are in seconds, and memory usage is in gigabytes. (a) Evaluation results for synthesis. The number of levels (*i.e.*, states) for each synthesized update function is shown in the *holes* column. (b) Evaluation results for specification pruning. We report for each example the size of the pruned specification domain and the size of the original specification domain.

Stochastic modeling An alternative to modeling biological systems using non-deterministic concurrency is to use stochasticity [2, 18, 24]. If we were interested in making predictions on the system's output behavior, *i.e.*, the *most likely* the behavior of the cell for a given mutation, we might select a model that predicts concentrations of proteins under varying initial parameters, including those not yet measured in the lab. Such predictive models are often stochastic.

In contrast, we care to only discover a mechanistic explanation for the cellular system (*i.e.*, how proteins communicate to agree on a particular cell fate). It is appropriate here to rely on a discrete model because the modeling problem is to find a program that reproduces each observed outcome on at least one execution — as opposed to some ratio of all executions. The existence of such a schedule is sufficient to determine the need to have the crucial protein-protein interaction.

Synthesis algorithms for concurrent systems. Our synthesis algorithm extends the synthesis algorithm for concurrent data structures [28]. That work showed how to extend the CEGIS algorithm [29] from the sequential setting into the semantics of concurrent programs. The resulting algorithm however did not handle the richer specification used in this paper (*i.e.*, the angelic correctness). Indeed, new algorithms had to be developed for the specifications of this paper. The Paraglider project developed synthesizers for concurrent data structures by deriving them from high-level specifications [31]. It is not clear how these derivation algorithms can be adapted to synthesis of concurrent systems under input-output examples such as ours.

Model checking [5–7, 11, 18] and abstract interpretation [9] have been applied to analyze various biological systems. All such efforts to manually construct and validate models have severely demonstrated the need for a synthesis system.

Various other paradigms have been used to model biological systems, including Petri nets [8], boolean networks [23], and process algebras [26]. While our techniques are not directly applicable, our success in synthesis for a model previously expressed in the expressive RM formalism demonstrates potential for synthesis in these other formalisms as well.

9. Conclusion

We present a language and develop algorithms for synthesizing concurrent models from experiments that perform mutations on biological cells and observe the results of the mutation on developed cells. We synthesize models that reproduce all non-deterministic outcomes of experiments. This variant of synthesis requires a 3QBF algorithm, which we design by allowing three solvers to communicate counterexamples. We also develop algorithms for analyzing specification ambiguity, ascertaining that a model is the sole biological explanation whenever possible under given biological assumptions, and computing minimal non-ambiguous specifications. We carried out a significant case study, synthesizing a model of vulval cell fate specification in the *C. elegans* earthworm that expresses a previously unknown biological hypothesis.

References

[1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[2] A. Arkin, J. Ross, and H. H. McAdams. Stochastic kinetic analysis of developmental pathway bifurcation in phage lambda-infected Escherichia coli cells. *Genetics*, 149(4):1633–1648, Aug 1998.

[3] Anil Aswani, Soile V. E. Keränen, James Brown, Charless C. Fowlkes, David W. Knowles, Mark D. Biggin, Peter Bickel, and Claire J. Tomlin. Nonparametric identification of regulatory interactions from spatial and temporal gene expression data. *BMC Bioinformatics*, 11:413, 2010.

[4] Nathan A. Barker, Chris J. Myers, and Hiroyuki Kuwahara. Learning genetic regulatory network connectivity from time series data. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 8(1):152–165, 2011.

[5] Grégory Batt, Calin Belta, and Ron Weiss. Temporal logic analysis of gene networks under parameter uncertainty. *IEEE Transactions of Automatic Control*, page 2008.

[6] Grégory Batt, Delphine Ropers, Hidde de Jong, Johannes Geiselmann, Radu Mateescu, Michel Page, and Dominique Schneider. Analysis and verification of qualitative models of genetic regulatory networks: A model-checking approach. In *IJCAI*, 2005.

[7] Nathalie Chabrier and François Fages. Symbolic model checking of biochemical networks. CMSB '03, 2003.

[8] C. Chaouiya. Petri net modelling of biological networks. *Brief. Bioinformatics*, 8(4):210–219, Jul 2007.

[9] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Abstract interpretation of cellular signalling networks. VMCAI'08, pages 83–97.

[10] Leonardo de Moura and Nikolaj Bjørner. Z3: Efficient SMT solver. In *TACAS'08: Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340, 2008.

[11] David L. Dill. Model checking cell biology. In *CAV*, page 2, 2012.

[12] J. Fisher, N. Piterman, A. Hajnal, and T. A. Henzinger. Predictive modeling of signaling crosstalk during C. elegans vulval development. *PLoS Comput. Biol.*, 3(5):e92, May 2007.

[13] Jasmin Fisher, David Harel, and Thomas A. Henzinger. Biology as reactivity. *Commun. ACM*, 54(10):72–82, 2011.

[14] Jasmin Fisher and Thomas A. Henzinger. Executable cell biology. *Nature Biotechnology*, 25(11):1239–1249, November 2007.

[15] Jasmin Fisher, Thomas A. Henzinger, Maria Mateescu, and Nir Piterman. Bounded asynchrony: Concurrency for modeling cell-cell interactions. In *FMSB*, pages 17–32, 2008.

[16] `https://oeis.org/A000670`.

[17] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 317–330. ACM.

[18] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 319(3):239–257, 2008.

[19] Na'aman Kam, Irun R. Cohen, and David Harel. The immune system as a reactive system: Modeling t cell activation with statecharts. In *HCC*, pages 15–22, 2001.

[20] Na'aman Kam, David Harel, Hillel Kugler, Rami Marelly, Amir Pnueli, E. Jane Albert Hubbard, and Michael J. Stern. Formal modeling of c. elegans development: A scenario-based approach. In *CMSB*, pages 4–20, 2003.

[21] `http://www.cs.berkeley.edu/~koksal/`.

[22] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the Power of Z3: Integrating SMT and Programming. In *CADE*, pages 400–406, 2011.

[23] S. Li, S. M. Assmann, and R. Albert. Predicting essential components of signal transduction networks: a dynamic model of guard cell abscisic acid signaling. *PLoS Biol.*, 4(10):e312, Oct 2006.

[24] H. H. McAdams and A. Arkin. Stochastic mechanisms in gene expression. *Proc. Natl. Acad. Sci. U.S.A.*, 94(3):814–819, Feb 1997.

[25] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[26] Aviv Regev and Ehud Shapiro. The pi-calculus as an abstraction for biomolecular systems. 2004.

[27] Aurélien Rizk, Grégory Batt, François Fages, and Sylvain Soliman. Continuous valuations of temporal logic specifications with applications to parameter optimization and robustness measures. *Theor. Comput. Sci.*, 412(26):2827–2839, 2011.

[28] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 136–148. ACM.

[29] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS-XII*, pages 404–415, New York, NY, USA, 2006. ACM.

[30] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, 2010.

[31] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Not.*, 43(6):125–135, June 2008.

[32] A. S. Yoo, C. Bais, and I. Greenwald. Crosstalk between the EGFR and LIN-12/Notch pathways in C. elegans vulval development. *Science*, 303(5658):663–666, Jan 2004.