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 systems. This paper addresses two problems that cell biologists face when developing formally analyzable models.

First, we show how to automatically synthesize an in-silico model for cell development given in-vivo experiments of how particular mutation experiments influence the cell fate. The problem of synthesis under mutations is unique because mutation experiments may have non-deterministic outcomes (presumably due to races between competing signaling pathways in the cells) and the synthesized model must be able to replay all these outcomes or else the model does not faithfully describe the desired cellular processes. (In contrast, a "regular" concurrent program synthesized under a permissive specification need not exhibit all allowed behaviors.) We developed synthesis algorithms and synthesized a model of cell fate determination in the development of earthworm *C. elegans*. A former version of this model previously took systems biologists months to develop.

Second, we address the problem of under-constrained specifications that arise due to missing mutation experiments which results in multiple models explaining the same phenomenon. This corresponds to analyzing the space of plausible models, i.e., models that can be synthesized under given specifications. We develop algorithms for exploring ambiguity in specifications, i.e., whether there exist alternative models that would produce different fates on some unperformed experiment. Using the algorithm, we show that for our *C. elegans* case study, two observationally equivalent models with same fates but with different protein interactions can be inferred. One of these was previously unknown to biologists.

In addition to posing the synthesis and ambiguity testing problems and developing algorithms, 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.

1. Introduction

Diseases can be caused by perturbed gene and protein regulatory networks. For example, disease X can be caused by a decrease of protein P, which has negative regulatory control over protein R. Once the level of P is decreased, high levels of R may cause disease X. Understanding regulatory function is an open question. One way to learn about regulation of cellular processes is to use *mutation experiments*, where 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 supressing protein P, the resulting phenotype can be attributed to, say, an overexpression of a known protein R, we can infer from this experiment that P negatively regulates R. From many such inferences, experimental biologists deduce detailed maps of regulatory networks that aim to describe the causal events leading to specific cell fates and behaviors.

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

Experimental biologists worry about the correctness of their models that are pieced together from a limited set of experiment observations, to explain dynamically what is only observed end-to-end. Executable biology [13, 14] addresses this concern. In this field, a formal and executable model is built so that it can be verified, say with model checking, against observed experiments. Unfortunately, turning informal maps of regulatory network into executable models is laborious because it involves explicitly defining timing and strength of how multiple proteins regulate others. In our previous work, some of us developed a model of VPC fate determination in the *C. elegans* worm [12]. This model correctly predicted an unknown protein-protein interaction but it took us three month to tweak the details of the model before it agreed with experiments. When new experiments are added, or if the model extended with new components, similarly expensive tweaks are required.

This paper develops techniques for synthesizing executable models from observed experiments and prior biological knowledge. Two challenges makes this synthesis problem an interesting formal methods problem. First, the outcomes of some cellular systems are nondeterministic. For example, in the *C. elegans* system that we study, some mutations cause the six observed Vulval Precursor Cells (VPCs) determine one of three alternative fates, presumably due to races in the communication among cells. The desired executable model must be able to reproduce all the observed behavior to be faithful to the experimentally observed cell phenomena. The choice of modeling language semantics is a non-trivial decision. Stochastic modeling would allow alternative transitions in protein concentration, which in turn allows the system to evolve into one of several states. Such systems are extremely sensitive to the probability assignments, and such precise probabilities are rarely known. Our modeling approach, on the other hand, achieves non-determinism by perturbing the schedule under which our (asynchronous) cells advance [15]. By using schedule interleaving for non-determinism, we leave proteins as deterministic functions. This in turn allowed us to makes it easier to synthesize the crucial protein functions.

Second, the mutation experiments form a partial specification and therefore are challenging to synthesize from. Because only certain genes are mutated from the total combinatorial set of possible mutations, biologists cannot be certain that a executable model verified against these mutations, synthesized or manually constructed, is the sole explanation of the cellular regulatory process. There could exist an alternative model that is *observationally identical* on the current ambiguous specification but *observationally distinct* on an additional mutation.

Therefore we go beyond synthesis and develop methods for the analysis of the space of plausible models, i.e., models that agree with the current partial specification. If observationally distinct models exist, we need to suggest a new mutation that differentiates them. On the other hand, if no alternative models exist, we want to determine what is the smallest set of experiments that rules out these models. Finding such a minimal set is interesting because, should biologists decide to redo the experiments for vali-

dation, it will suffice to validate the mutation experiments that sufficiently constrain the plausible models. Finally, it is interesting to ask whether there are observationally identicla but internally different models. Such models present alternative regulatory networks. These models cannot be distinguished by observing phenotypes; we must instrument proteins with fluorescent markers (similar to tracing the program) and observe the cell during its development. This is a harder experiment but formal methods here help in identifying which genes to mark, reducing the cost of instrumentation.

These observations lead to the following domain characteristics:

1. System specification will be a small finite set of input-output examples.
2. Non-deterministic outcomes are allowed, and when present required to be exhibited. Models are terminating and produce the stated outcomes within bounded time.
3. The model will be structured hierarchically, as shown in Figure 2. Cells follows a bounded asynchrony execution model.

We have built an efficient verifier, synthesizer and a specification ambiguity analyzer that all exploit the above characteristics. Our synthesizer takes as input the mutation results and a template structure of the cells, and from it generates a validated model. The template of the cell defines the contained components, and their interconnections (inhibition, activation). These are well-known from the biological literature. Additionally, the concentration levels on the connections are known, which are discretized. What is not known is the internal logic and timing of the components, i.e., when they trigger and under what conditions of the incoming signals. Our synthesizer generates exactly these, off-loading the most difficult task of systems biology modeling to a computation search engine.

This paper makes the following contributions:

1. We formulate the model synthesis problem and observe that unlike previous synthesis tasks, e.g., concurrent synthesis [21] or synthesis from examples [16] or invariants [23], which were 2QBF, this problem is 3QBF. We develop efficient algorithms for solving this problem that boil down to three communicating SAT solvers.
2. Analyzing the specifications and the space of plausible models: We developed algorithms for determining whether internally or externally distinguishable models exist. These build on our 3QBF synthesis solution. These could guide new wet-lab experiments. Our tool already suggests a new experiment that could potentially lead to a new biological discovery.
3. We designed a programming language for expressing our models based on bounded asynchrony [15]. We embed this domain-specific language into Scala and built a lightweight synthesizer, which is publicly available. We describe how to build such synthesizers.
4. Our synthesizer efficiently (1) generates valid models for the *C. elegans* VPCs. The model is readable and also fixes a bug in previous modeling—incorrect modeling of a mutation; (2) shows that no behaviorally distinct models exist (even after expanding the mutation space), but two internally different models were synthesized; (3) prunes to the specification to the minimal set of 4 mutation experiments.

2. Technical Overview

This section illustrates synthesis of programs under design considerations of executable biology. We formalize mutation experiments, describe the programming language that we use to construct the biological model, outline our synthesis algorithms, and describe the queries for analyzing the space of plausible models.

2.1 Background on Mutation Experiments.

Here we give a brief background on mutation experiments in developmental systems biology. The role of these experiments is to understand cellular genetic regulatory networks, including those that control stem cell differentiation. The 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: (i) the cell divides into cells of different type, *e.g.*, based on the gradient of a "broadcast" signal from an anchor cell; (ii) multiple identical cells differentiate by mutually communicating in order to arrive at coordinated fates. Therefore, to understand cell differentiation, one needs to understand inter-cell communication.

Developmental systems biologists seek to infer the program that stem cells "execute" to decide their fate. This program executes in a single division cycle during which a pluripotent cells decided its fate, potentially by communicating with other cells. One method for inferring this program is to mutate the cell and observe the resulting changes in the cell development. These experiments are particularly attractive because phenotype changes are visually observable, avoiding the need for the more expensive tracing of temporal protein levels by tagging cell proteins with fluorescent genes.

From gene mutation experiments, biologists infer protein interactions. For example, Yoo *et al.* infers:

> In this assay, depletion of lst-2, lst-3, lst-4, or dpy-23, as well as ark-1, caused ectopic vulval induction, suggesting that they function as negative regulators of the EGFR-MAPK pathway. [25]

From such piecemeal information, biologists create informal models of cellular programs, such as the one in Figure 1 from [12], which show 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. The edges between cell components (receptors and proteins) show the activation (→) vs. suppression (⊣).

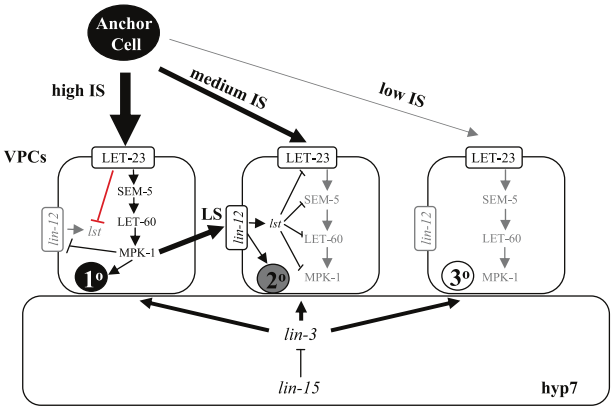


Figure 1. An informal *C. elegans* connection diagram of three VPC cells [12]. These cells react to the inductive signal (IS) from the anchor cell and communicate using the lateral signal (LS) among themselves.

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 that have been outcomes under some mutations. Due to this lack of dynamic information, one cannot be certain that these diagrams contain all protein interactions exercised by the cell.

The goal of *executable biology* [13] is to create models that can be verified against observed experiments. Mechanical verification allows tackling the combinatorial complexity of races in inter-cell communication.

A challenge in executable biology is in creating a model. To transform the informal model in Figure 1 into an executable model, the designer must model at least (i) protein levels; (ii) timing or rates at which proteins react to other components; and (iii) how a protein behaves when both an activator and an suppressor of the protein are active. We have previously developed a verified model of *C. elegans* VPC cells; that model took three months to develop [15]. This paper develops method for automatic synthesis of executable models.

2.2 Program Inference from Mutation Experiments

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 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. With n cell components and three possible mutations per component (e.g., no mutation; suppressed; high level), M is exponential in the size of the cell. As a result, biologists do not carry out all mutations.

Incomplete experiments imply that we have to accommodate partial specifications. Technically, the set of experiments $Exp \subset M \times F$, where $(m, f) \in Exp$ if the fate f has been observed on the mutation m. To facilitate synthesis with both positive and negative examples, we (reasonably) 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. Without this assumption, we would have no upper bound on the specification, as any (m, f) pair could potentially be observed in future experiments. 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. We will adapt this correctness condition for our modeling approach in the next subsection, which explain that we control P through a schedule.

2.3 Our Modeling Approach and the Modeling Language

A correct model must reproduce all nondeterministic outcomes $E(m)$ of an experiment with a mutation m. Usually, a model is not designed to produce the alternative outcomes in a single execution. Instead, the model includes some form of non-determinism so that the set of possible model executions covers the experimental observations $E(m)$. In stochastic models [3, 19], this non-determinism takes the form of proteins models making probabilistic transitions, accounting for variability of protein level change rates in nature. Our modeling approach moves non-determinism from protein modeling into the scheduler that controls the execution of our concurrent models [15]. This shift allows our protein models to be deterministic, which we believe simplifies protein model synthesis.

Our modeling language is not fully asynchronous because cells in nature evolve at similar rates. Therefore, our schedules adhere to a *1-bounded asynchrony* constraint, which divides the schedule into blocks such that in each block, each cell takes exactly one step. As a corollary, between two executions of a cell, no other cell can take more than two steps [15].

We have materialized our bounded-asynchrony programming model in a high-level programming language inspired by biological diagrams of Figure 1. The language introduces programming abstractions for cells, cell components, and interaction between components. Thanks to these abstractions, our programs are syntactically smaller, which makes their synthesis very efficient, compared to models expressed in the Reactive Modules language [2], which was the modeling language used in [12].

Correctness Condition. Programs in our language have the form $P : (M, S) \to F$, where M and F are domains of configurations (mutations) and fates, while S is the set of bounded-asynchrony schedules. The explicit schedule allows us to formulate a precise correctness condition $correct(P, E)$ of amodel P on a specification $E : M \to 2^F$, which has two parts:

1. *demonic scheduling:* A demonic scheduler cannot make the model produce 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 the model overapproximation. Our models are precise in that they meet both conditions.

Programs in our language (cf. Figure 2) are composed of cells, which execute according to a schedule s. The schedule is of bounded fixed length; the number of steps 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 components in the same cell or in other cells; communicating components are connected with directed edges, which correspond either to activation or suppression 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 suppressors. 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 suppressors, combined.

2.4 An Example

In an attempt to make the description of our problem more accessible, we have translated our running example from a problem of cell model inference to a problem of designing a toy distributed protocol. Turning an inference into design means that our specifications are not observed laboratory experiments but rather human-provided desired outcomes of the system under design, which is an insignificant change in our setting.

The goal. We wish to design a weak consensus protocol for a three-cell system, where each cell is a node in a distributed system. (This protocol is inspired by communication between biological cells.) Two cells (called sensors N1 and N2) are listening to a signal from

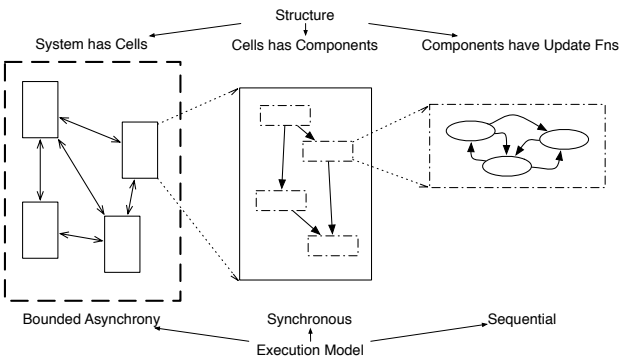


Figure 2. Hierarchical organization of concurrent system.

a master cell (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 (i) 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 (ii) receiving the release message from the other signal. To complicate the situation, the communication between the two sensors may be down due to power saving, and so sensors cannot rely on receiving a message from their peer. To help matters, we can assume that sensors have bounded skew, *i.e.*, they run under bounded asynchrony schedule.

The specification, expressed as a set of experiments, is shown in Figure 3(b). The left column shows the mutations (configurations) 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 signal arriving at N2 is low, and the communication between nodes is turned off. We can think of this mutation as the adversary lowering the signal to N2 and turning off the radio 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.

Figure 3(a) shows the program. On the left is the top-most level with three cells; in the middle is a sensor node showing its components (if this node was a cell, the components would be proteins). The ?? blocks on the right show the unspecified update function that we are asking the synthesizer to compute.

2.5 Synthesis

The input to the synthesizer is the specification E and a *partial program* $P^?$ that is completed by the synthesizer, if possible, into a program P^h such that $correct(P^h, E)$. 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 the update functions that model cell components. Because update functions model timing and change rates of proteins, they are 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 suppressed 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.

From the user standpoint, the partial program $P^?$ encodes biological assumptions because it defines the components in the cells as well as a superset of connections between them. It thus (i) conveys the desire to model particular proteins and (ii) states the knowledge of which (superset of) pairs of proteins communicate. These 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:

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

This synthesis problem is harder than that for other concurrent systems [21] because a model must reproduce all observed experiments, captured in the $angelic(P)$ correctness condition, which is a 2QBF formula. This makes the synthesis problem a 3QBF problem. Typical synthesis problems are 2QBF.

We develop an algorithm for the 3QBF synthesis problem with a two-part counterexample-guided inductive synthesis (CEGIS) algorithm. (An inductive synthesizer produces a program that is correct on a small sample of inputs; this candidate program is then verified on remaining inputs.) Compared to the classical CEGIS algorithm, where an inductive synthesizer communicates with a verifier [22], the two-part CEGIS algorithm communicates with two verifiers, one for each of the two correctness conditions, and collects two kinds of counterexamples, one from each verifier. From the demonic verifier, our algorithm collects an input-schedule pair (m_i, s_i), while from the angelic verifier it collects input-output pairs (m_i, f_i). By collecting these counterexample, the two-part CEGIS algorithm decomposes the 3QBF problem into two 1QBF (*i.e.*, SAT) solvers (inductive synthesizer and the demonic verifier) and one 2QBF solver (the angelic verifier). 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 $P^?$).

Example 2. The partial program $P^?$ for the desired weak consensus is shown in Figure 3(a). We wish to synthesize update functions for the receivers and the delay components. These update functions control how these components react to signals from the base station and the peer sensor. The synthesized functions from our tool are shown in Figure 4. 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 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. (A model P becomes invalid under a new experiment (m_{n+1}, f_i) if $P(m_{n+1}, s) \neq f_i$ for all schedules s.) 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. First, we define aggregate outcomes and ambiguity:

DEFINITION 1 (Aggregate outcome). *Let P be a model and m an input configuration. Then the* aggregate outcome *of P on m, denoted $P[[m]]$, is the set of outcomes over all schedules: $P[[m]] \doteq \{P(m,s) \mid \forall s \in S\}$*

A specification E is ambiguous under biological assumptions expressed in a partial programs $P^?$ if we can find two models that disagree on some new experiment. Of course, one of these models would become invalid given the new experiment.

DEFINITION 2 (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.*, $m \in M \setminus dom(E)$. In our case study, we show that the specification E is unambiguous given provided biological assumptions (*i.e.*, , there is no need for more experiments at the desired level of modeling). We have also shown that removing some historically important experiments indeed makes the specification ambiguous, permitting alternative explanations for how cells coordinate.

We develop an algorithm for the *alternative model query* that, given an existing (perhaps previously synthesized) model P and a partial program $P^?$ stating biological assumptions, finds an input configuration m and a new model P^h such that $P[[m]] \neq P^h[[m]]$, or shows that no such h and m exist.

Now consider a research scenario where one wants to validate a set of experiments performed in the literature. Is it possible to identify the smallest set of experiments whose replication is sufficient to yield a non-ambiguous specification? We support a *minimization query* that computes a minimal non-ambiguous specification E_m, *i.e.*, $\neg Amb(E_m, P^?) \wedge \neg\exists E', E' \subset E_m \wedge \neg Amb(E', P^?)$. (This query is defined when E is non-ambiguous, of course.)

In our case study, we should that, under our assumptions $P^?$, one does not need to replicate about 90% of experiments. This result suggests that computing which experiments to perform might reduce unnecessary laboratory work.

Example 3. We examined the example in Section 2.4 using our query for alternative models. Supposing we relaxed the specification and did not care about the outcome on the case $N1 = L$, $N2 = L$. We ask our synthesizer to generate models under this relaxed specification with the output from the example in 2.4 as the model to differentiate against. 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 4(a',b',c'). When we ask for a mutation that distinguishes among the models, the synthesizer produces the omitted row.

We have also explored the minimization query. Our synthesizer prunes E down to the first three rows of Figure 3(b) as a minimally ambiguous specification. This is somewhat surprising but it gives substantial insight into the problem, as the user can now understand that the specification of the three transmit OFF cases was redundant, while the user may have presumed that it was necessary. More importantly, the minimization demonstrates that even though redundant rows were present, the specification was at least consistent, which may be useful diagnostics.

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 our language is a *component*. In the context of biological models, a component may represent genes or proteins in a living cell. We denote the set of all components in a program by $Comp$. Components are connected via a set of directed *edges* $Edges : Comp \times Comp$. Edges model channels of communication between cell components. For each component c, we say a component c' is an *input component* 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 is associated with a state σ_c that takes values from a finite domain L_c. In biology, the component state corresponds to a discretized activation/concentration level of a component. Each component c is also associated with an update function, denoted f_c, that updates its state σ_c. The function f_c has domain $L_c \times \Pi_{c' \in Input_c} L_{c'}$ and range L_c. In biology, these update functions model the behavior of proteins based on their own activation level and the level of other components that influence them. The update function for a component is chosen from within 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.

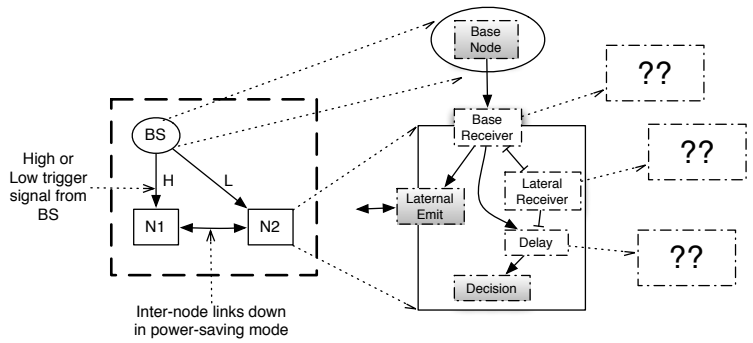
A *cell* is a set of components. In biology, this corresponds to an actual cell that contains biological components. Within a cell, we have a synchronous execution model, i.e. all components of a cell update their state simultaneously using their update function. The state of a cell $\bar{\sigma}$ is 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*. 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 in the execution of the program on this input configuration. 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 very 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.

EXAMPLE 1. *For the example in Section 2.4 the user declares two different types of Cells. One the BS, and another for N1 and N2. In the BS they declare a component Base Node, and for N1, N2 they declare components Base Receiver, Laternal Emit, Lateral Receiver, Delay, and Decision. They leave the update functions for Base Receiver, Lateral Receiver, and Delay undefined as they contain the complex behavior. The remaining Lateral Emit, Decision, and Base Node are trivial functions that either relay, or reach constant values. They also specify the connections as indicated in Figure 3(a).*

Operational semantics Figure 5 shows the small-step semantic rules for the execution of the program. Here, we assume that the program starts in the initial state $\bar{\bar{\sigma}}_{init}$, and that the program 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 structure of the system. The schedule S, with a schedule step s, partitions the cells to the set $enabled$ (for whom $s(cell) = 1$) and $disabled$ (for whom $s(cell) = 0$). For the disabled cells the state remains unchanged.

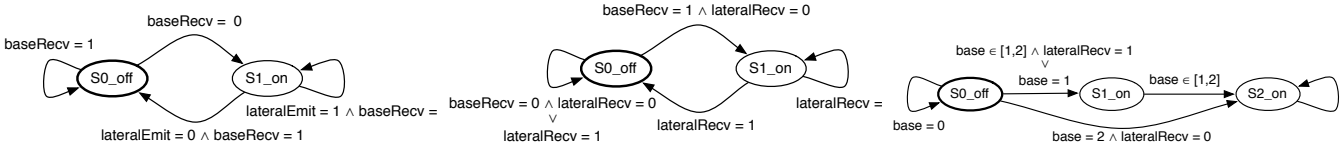


(a)

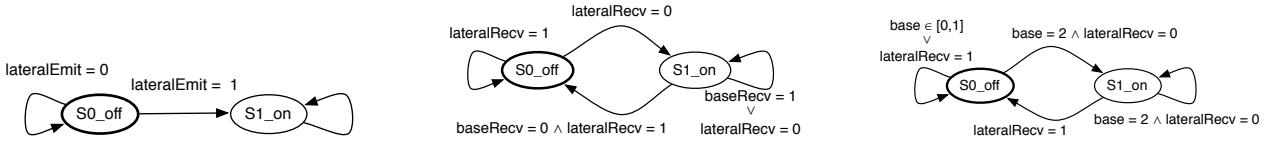
Base station trigger	Tx Radio	N1	N2
N1=H, N2=H	ON	C	D
		D	C
		C	C
N1=L, N2=L	ON	C	D
		D	C
		C	C
N1=H, N2=L	ON	C	D
N1=L, N2=H	ON	D	C
N1=H, N2=H	OFF	C	C
N1=L, N2=L	OFF	C	C
N1=H, N2=L	OFF	C	C

(b)

Figure 3. (a) Template of connections in distributed system. The top node is the base station, and the bottom nodes are distributed sensors whose transmit radios might be switched off leading to no horizontal communication links. (b) The specification giving required outcomes for nodes N1 and N2 under a range of Scenarios of base station trigger signals and cases of transmit radio on the ode being ON or OFF. C = Commit, D = Delegate.



(a) Lateral receiver (b) Delay (c) Base receiver



(a') Lateral receiver' (b') Delay' (c') Base receiver'

Figure 4. Output of the synthesizer, i.e., graphical representations of the update functions, for the example in Section 2.4 for the unknown components. Also shown is the update function for the alternative model generated using Q3 to differentiate from the previous model under ambiguous specification, i.e., removing row 2 of Figure 3(b).

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

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

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

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

Figure 5. Small-step semantics for system execution. RUN-SYSTEM runs a schedule by advancing the cells according to each microstep in the schedule. ADVANCE-CELLS rule updates the states of cells, depending on the current microstep 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

For the enabled cells, each component is advanced by reading the relevant neighbors and stepping based on the update functions.

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 upto 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 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 succintly 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[1]. The following lemma holds:

LEMMA 1 (Fisher et al. [15]). *A micro-schedule exists iff 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 been intractable with micro-steps.

4. Translating Programs into Formulas

We now describe how to translate a program $(Cells, Edges)$ to a SMT formula which enables various verification and synthesis algorithms. We first give rewrite rules that translate the concrete execution of the program to a symbolic execution. We then describe additional constraints that encode biological domain knowledge to be used in synthesis of programs.

4.1 Translation of Program Execution

The translated SMT formula for the partial program $(Cells, Edges)$ 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 can hold one of the three values "$<$", "$>$" and "$=$". Channel variables $channel_{t,c_1,c_2}$ and $channel_{t,c2,c1}$ are asserted to be consistent in the following way:

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

- 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.

Translation rules for compiling the program $(Cells, Edges)$ to a SMT formula are shown in Figure 6.

4.2 Domain-Specific Constraints on Update Functions

The translation above does not impose any restrictions on the structure of the update functions that are left unspecified by the user. In biology, formulating a hypothesis typically involves stating high-level invariants about whether a component *activates* or *inhibits* another one. We found that asserting constraints that encode these invariants based on user annotations on components and edges between them is crucial for ensuring that the structure of update functions agree with the invariants. In the following, we describe two kinds of invariants that encode monotonicity properties on update functions.

When modeling biological systems, component states typically encode chemical concentrations, and for a such component c, the values L_c can be annotated to have a total order $\leq$ over them.

We define a partial function $label$ from $Edges$ to the set $\{activating, inhibiting\}$, that annotates edges in a program $P^?$ as either activating or inhibiting. Intuitively, if there is an activating edge from component c_1 to component c_2, then an increase in σ_{c_1} should not have 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}.

Given a component c with update function $f_c : L_c \times L_{c_1} \times \ldots \times L_{c_n} \rightarrow L_c$, we define the partial order $\preceq$ on elements of $L_{c_1} \times \ldots \times L_{c_n}$ in the following way:

$$\begin{array}{l} (v_1, \ldots, v_n) \preceq (u_1, \ldots, u_n) \\ \quad \doteq \ \forall i \in \{1, \ldots, n\}. \\ \qquad (label((c_i, c)) = activating \wedge v_i \leq u_i) \vee \\ \qquad (label((c_i, c)) = inhibiting \wedge v_i \geq u_i) \end{array}$$

Intuitively, $\preceq$ is a partial order on the strength of the input values to a component. We use this partial order to assert two types of constraints on the structure of the update function f_c. The first one encodes that, given a value of σ_c, a stronger value of the input will not have the effect of decreasing the value of σ_c, i.e., the updates through f_c should be corresponding higher:

$$\begin{array}{c} \forall i_1, i_2 \in L_{c_1} \times \ldots \times L_{c_n}. \forall v \in L_c. \\ i_1 \preceq i_2 \Rightarrow f_c(v, i_1) \leq f_c(v, i_2) \end{array}$$

The second constraint 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, the greater the activation level of the component is, the greater the updated value will be:

$$\begin{array}{c} \forall i \in L_{c_1} \times \ldots \times L_{c_n}. \forall v_1, v_2 \in L_c. \\ v_1 \leq v_2 \Rightarrow f_c(v_1, i) \leq f_c(v_2, i) \end{array}$$

5. Synthesis and Querying Spaces of Models

In this section, we present algorithms for synthesizing models from partial programs, as well as querying spaces of these models for

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

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

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

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

Figure 6. Translation rules for symbolic execution of programs.

ambiguity analysis of specifications. The algorithms we describe leverage the translation to formulas presented in Section 4.

In the translated execution of a program, holes, schedules and input configurations are symbolic. The space for holes and the space for schedules are typically very large. However, specifications are typically wet lab experiments which are sparse and inherently small ($O(10^2)$). This enables efficient solving algorithms that unroll quantifications of configuration inputs over specifications.

In the following, we refer to the symbolic output parameter of translating the run of program P with input m and schedule s as $P(m, s)$, and to the specification as the partial function E.

5.1 Verifying Programs

The correctness condition, presented in Section 2, is defined as:

$$correct(P) \doteq demonic(P) \wedge angelic(P)$$

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 for demonic schedules for the first, and a verifier V_a for angelic schedules for the second.

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

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

To check this property, we attempt to disprove it by searching for a demonic schedule that produces an unobserved output for an input in the domain of E. Given the observation that there is a small set of input values in the domain of E, we solve this formula by unrolling the existential quantification over this set, and querying symbolically for a demonic schedule. The containment $P(m, s) \notin E(m)$ is expressed by unrolling for 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 outputs in the set that m maps to must be observable, i.e. appear in some execution of P on m:

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

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 by unrolling values of the domain $dom(E)$, again based on the assumption that this domain is small. We perform unrolling by formulating 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 output 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.

5.2 Synthesizing Programs

In our language, it is possible to define a program sketch $P^?$ 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 described above. Our procedure leverages two verifiers V_d and V_a to check correctness properties $demonic$ and $angelic$ respectively, in order to solve the following synthesis problem:

$$\exists h.\, demonic(P^h) \wedge angelic(P^h)$$

This formula is in 3QBF, due to the quantifier alternation $\exists\forall\exists$ when considering $angelic(P^h)$ within the quantification over h. We solve it by performing counterexample-guided inductive synthesis, using a synthesizer that solves a 1QBF formula and that communicates with the two verifiers to either validate a candidate model or to add new counterexamples to consider for inductive synthesis. The solver architecture can be seen in Figure 7.

More 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:

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

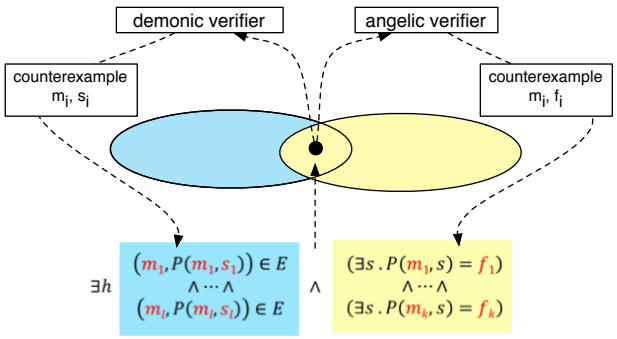


Figure 7. The synthesizer consists of three communicating solvers. The two verifiers generate two kinds of counterexamples, and the synthesizer generates models that satisfy the constraints for all counterexamples.

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 synthesize a new candidate after updating the counterexample sets CE_1 and CE_2 based on the values returned by the verifiers. 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 now 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 outputs Obs by searching for angelic schedules leading the program to produce an output a formerly unobserved output. We express containment of the observed output by unrolling for each value in the Obs set inferred so far, and solve the following formula at each step:

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

If the above formula is satisfiable, we obtain an output that we add to the observed output set Obs, and attempt to solve the formula with the updated set. If it is unsatisfiable, we have obtained all observable outputs 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 a different 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, we can state this query formally as:

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

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 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: (i) It can either contain an output value not in $P_1[[m]]$, or (ii) 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 output not seen in $P_1[[m]]$ can be found by augmenting the synthesis query described in Section 5.2 with the constraint $P_2^h[[m]] \setminus P_1[[m]] \neq \emptyset$. The resulting 3QBF formula is handled using the same mechanism of a synthesizer communicating with two verifiers to perform inductive synthesis. The formula that is solved in this case is defined as following:

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

This formula is satisfiable if and only if there exists a completion of program P_2^h that decides an output not in $P[[m]]$.

Case 2. Alternatively, P_2^h may be found by attempting to synthesize a model that always produces outputs 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 output set is a strict subset of observable outputs of P_1 on input m.

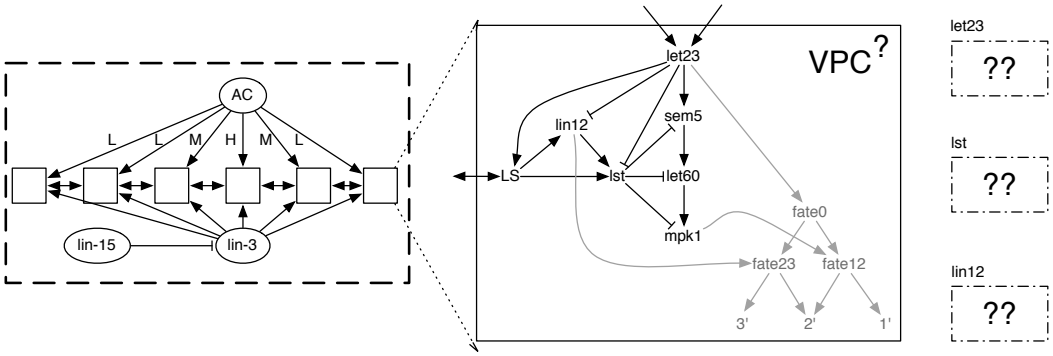
5.3.2 Minimization

In a context where performing experiments is an expensive process, a researcher may want to obtain a minimal non-ambiguous specification that is sufficient for validating 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 a partial program $P^?$. We achieve this by considering each element m in the domain of E in order, and maintaining a set of inputs that were not marked as redundant yet.

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 nonredundant input values. We leverage the *alternative model* query described in Section 5.3.1 to search for such a model. If synthesis fails, m is marked as redundant. If synthesis succeeds, then 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.



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

Figure 8. (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].

6. Case Study

6.1 *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) take on specific fates. These interactions are informally found in biological literature from which we develop our template $VPC^?$. The template is shown in Figure 8(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 do not want to see complex behavior in them. Additionally, introducing holes in these nodes leaves too much freedom to the synthesizer to generate models that 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 very well understood behavior colludes with the other nodes to give models that are hard to explain to the biologist. 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 (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 do manually, even after considerable effort.

The specification consists of 48 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 8(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 9(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*.

6.1.1 Specification ambiguity for *C. elegans* VPC models

Next we analyzed the ambiguity in the specification. The big 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 experimented 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 44 outcomes are asserted. We found that under this relaxed specificaiton *all* four options of inhibition coming from any node of the cascade work.

Then using the alternative model query from Section 2.6, we asked for a model including any one of the 4 remaining outcomes. The synthesizer eliminates two that have inhibition emanating from let60 and mpk1. This was very significant since it validated, and 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 muta-

tion 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 could take hours to days to perform) as they now know that mutation experiments will not 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 2.6 for each of the VPC queries, with suprising results. We infer for the space we are searching over, only 4 experimental observations suffice to yield a unique model. This set contains a non-deterministic outcomes, and additionally others that together constrain the system enough to yield the unique model that is explained by the full 48.

Wet-lab predictions Our exploration demonstrated that (a) let23 is not the only possibility for inhibition, but sem5 is as well, (b) let60 and mpk1 cannot play that role, (c) the models using let23 and sem5 cannot be distinguished observationally. These suggest a possible inhibition from sem5, that cannot be distinguished through mutation experiments, and so 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 as its underlying constraint solver. We interface with Z3 through the Scala^{Z3} library. Our framework consists of 5K lines of code.

We show performance results for the evaluation of our synthesis procedure in Figure 10(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 very efficient.

In Figure 10(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 that just synthesis. This is because multiple synthesis and verification queries are solved in the process of minimization. But compared to the amount of time this saves the biologist, i.e., months-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 [4]. Rizk *et al.*find parameters for ODE models by optimizing a notion of continuous degree of satisfaction of temporal logic formulas. 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 [5]. Time series data of concentrations is not available in our setting, so these approaches do not apply to the inference of our models.

Synthesis algorithms for concurrent systems. Our synthesis algorithm extends the synthesis algorithm for concurrent data structures [21]. That work showed how to extend the CEGIS algorithm [22] 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 paraglide project developed synthesizers for concurrent data structures by deriving them from high-level specifications [24]. 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 has been applied to model various biological systems [6–8, 11, 17], and all such modeling efforts have severely demonstrated the need for a synthesis system. Various other formal method techniques, abstract interpretation [10], petri nets [9], boolean networks [18], and process algebras [20]. 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 algorithms for synthesizing concurrent models from experiments that perform adversarial 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 the space of plausible models, ascertaining that a model is the sole biological explanation whenever possible under given biological assumptions. We have carried out a significant case study, synthesizing a model that we previously found difficult to produce by hand.

References

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

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

[3] 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.

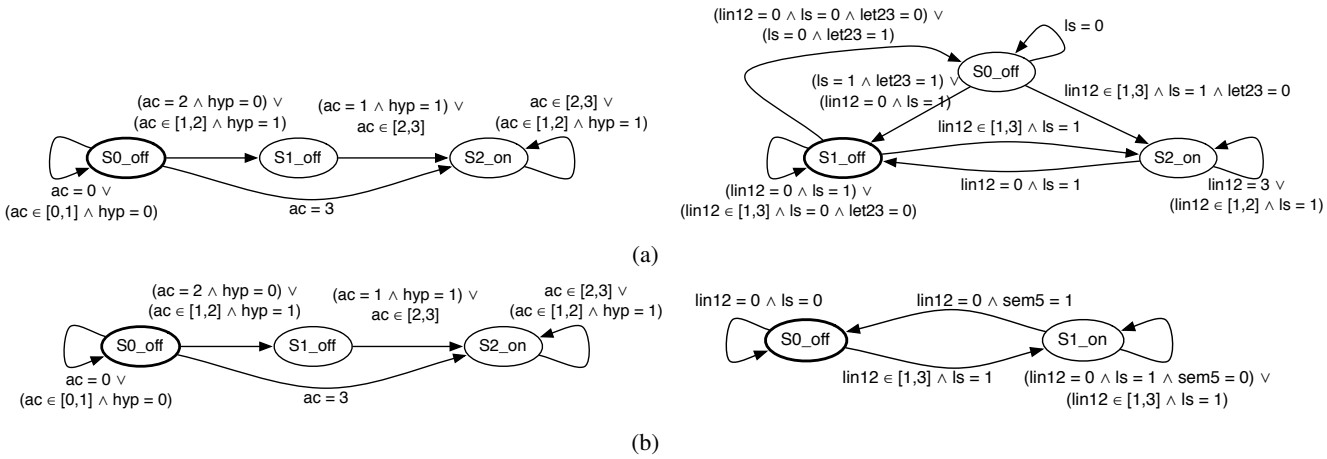


Figure 9. Synthesized update functions for two different connection topologies, for Let23, and Lst. (a) The topology with Lst and Lin12 inhibited by Let23. The template allows for three Let23 states and three Lst states. (b) The topology with Lst and Lin12 inhibited by Sem5. The template allows for three 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 10. All times are in seconds, and memory usage is in gigabytes. (a) Evaluation results for synthesis. The number of levels 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.

[4] 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.

[5] 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.

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

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

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

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

[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] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

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

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

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

[25] A. S. Yoo, C. Bais, and I. Greenwald. Crosstalk between the EGFR and LIN-12/Notch pathways in C. elegans vulval development. *Science*,