A Tool for Automated Inference in Rule-Based Biological Models

by

Chelsea Voss

S.B., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© Chelsea Voss, MMXVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature redacted

Author.....

Department of Electrical Engineering and Computer Science

Signature redacted

Certified by ..

Armando Solar-Lezama

Associate Professor of Computer Science

Signature redacted

Accepted by ..

Dr. Christopher Terman

Chairman, Masters of Engineering Thesis Committee
A Tool for Automated Inference in Rule-Based Biological Models

by

Chelsea Voss

Submitted to the Department of Electrical Engineering and Computer Science on May 1, 2016, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Rule-based biological models help researchers investigate systems such as cellular signalling pathways. Although these models are generally programmed by hand, some research efforts aim to program them automatically using biological facts extracted from papers via natural language processing. However, NLP facts cannot always be directly converted into mechanistic reaction rules for a rule-based model. Thus, there is a need for tools that can convert biological facts into mechanistic rules in a logically sound way. We construct such a tool specifically for Kappa, a model programming language, by implementing Iota, a logic language for Kappa models. Our tool can translate biological facts into Iota predicates, check predicates for satisfiability, and find models that satisfy predicates. We test our system against realistic use cases, and show that it can construct rule-based mechanistic models that are sound with respect to the semantics of the biological facts from which they were constructed.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor of Computer Science
Acknowledgments

Thanks to Prof. Jean Yang for your incredible support and mentorship! Throughout this project, your guidance has been invaluable: from brainstorming directions for us to investigate, to your advice on Z3 and implementation, to helpful feedback on my code and writing.

Thanks to Prof. Armando Solar-Lezama for your support of this work and for great conversations giving me ideas about how to tackle the Iota implementation, opportunities to think over Syndra’s design, and advice on the design of this thesis.

Thanks to Prof. Walter Fontana for your guidance and for giving me insights into Kappa, rule-based biological modeling, and the trickiness involved in unpacking non-mechanistic rules into mechanistic rules.

Thanks to Prof. Peter Sorger for generously sponsoring this project, and thanks to John Bachman and Benjamin Gyori of the Sorger lab for collaborating with us on INDRA integration, for providing example INDRA statements for us to make inferences on, and and for fielding our many questions about what kind of tool would be the most useful!

Thanks to Héctor Medina for providing additional examples of inferences for Syndra to solve. Finally, thanks to Adrien Husson and Jean Krivine for the design of Iota, and for talking to me in person about Iota and answering my questions about how it works.
# Contents

## 1 Introduction
1.1 Biological modeling ......................................................... 10
1.2 Which abstraction level? Rule-based modeling .......................... 11
1.3 Programming models automatically ...................................... 13
1.4 The problem: turning facts into rules .................................. 14
1.5 The solution: logical inference over chemical semantics .......... 15

## 2 Related Work
2.1 Modeling with Kappa ......................................................... 19
2.2 Other modeling paradigms .................................................. 20
2.3 Analysis of Kappa ............................................................ 21
2.4 Big Mechanism ecosystem .................................................. 22
2.5 Bringing it all together ...................................................... 23

## 3 Predicates over biological models
3.1 Predicates over the space of models .................................... 26
3.2 Iota ................................................................................. 26
3.2.1 Semantics ..................................................................... 27

## 4 Architecture and implementation
4.1 Predicates ......................................................................... 32
4.2 Atomic Predicates ................................................................ 34
4.2.1 Wrapping atomic predicates ........................................ 35
List of Figures

1-1 Diagram of signaling pathways involved in cancer, illustrating a portion of the immense volume of knowledge in cellular and molecular biology. .......................................................... 11
1-2 Abstraction levels in biological models. ........................................ 12
1-3 Rule-based modeling: reaction rules are executed to simulate a run of the system. .................................................. 13
1-4 Inference rule that can combine two facts into a mechanistic rule. .. 16
1-5 Example inference rules that could be used to make logical deductions about collections of facts, in the naïve approach. .......... 16

2-1 The INDRA pipeline, with Syndra as intermediate analysis tool. . . 23
2-2 Two components: converting biological facts to predicates, convert-
ing predicates to models .............................................. 24

3-1 Semantics of Iota’s atomic predicates [16] ............................... 28
3-2 Semantics of Iota’s predicates [16] ........................................ 28

4-1 Subclasses of Predicate; see predicate.py for details. ................. 33
4-2 Subclasses of AtomicPredicate; see atomic_predicate.py for details. 35
4-3 Predicate semantics of each atomic predicate [16]. ................... 36
Chapter 1

Introduction

Computational biological models are tools that biology researchers can use to test their research hypotheses against current biological knowledge in silico, before deciding on experiments to run in vivo. These models are programs, written in specialized programming languages for making biological models. Without continuous manual efforts to keep them up to date, they are at risk of becoming stale when faced with the rising tide of new biological research results: software obsolescence strikes again!

Towards a method for more easily creating computational biological models, ongoing research projects try to use natural language processing (NLP) to extract facts from the scientific literature and to construct models automatically based on those facts. However, NLP does not make guarantees about correctness, and the facts extracted by NLP may be ambiguous and difficult to translate directly into the hard-and-fast mechanistic rules that modeling languages require. These NLP tools do not allow for logical inferences among disconnected facts, but such logical inferences are necessary in order to construct a complete model.

In order to solve this problem, we propose an approach that translates biological facts into predicates in a logic language designed for specifying the chemical behavior of models. Then, we demonstrate the feasibility of this approach by implementing Syndra, a tool for making logical inferences among biological facts in exactly this way, enabling the construction of complete and accurate biological models.
1.1 Biological modeling

A computational biological model is a program written in one of many model programming languages, intended to approximate the behavior of some biological system when executed. These models provide a tool that researchers can use to record knowledge about biological systems, to make predictions using that knowledge, and to inform the design of better experiments.

Computational biological models are a necessary component of modern biology's research toolkit. As cellular and molecular biology advances, with the constant publication of new experiments and new discoveries, it becomes more and more impractical as a researcher to read every new paper in order to keep up with the cutting edge of the field. However, understanding others' work is critical in order to perform one's own research effectively: with an up-to-date mental model of cellular and molecular biology, a researcher can design better experiments, avoid reinventing the wheel, and discover new results faster.

One way to have a more effective mental model is to consult external resources: the diagram in Figure 1-1, for example, provides a reference for signaling pathways involved in cancer. However, most such resources are static. Computational biological models have two advantages: (1) they can be changed to accommodate new discoveries, unlike books or posters; and (2) they can be executed in order to test hypotheses.

These models are especially useful for dealing with systems that are difficult to reason about in full. For systems as complex as the cell, making computational models allows researchers to reason and make predictions where keeping a mental model of the system might be impractical. Once existing knowledge about interactions of proteins and other molecular agents has been assembled into a computational model, researchers have that model as a reference to consult and a tool to interact with.
1.2 Which abstraction level? Rule-based modeling

Biological models have to make tradeoffs between being accurate and being simple to compute. At the smallest, most accurate level are models calculating the delicate atomic interactions among the amino acids in order to investigate protein folding. Getting bigger, we can trade off accuracy for simplicity in order to model a bigger population of proteins, by approximating proteins' behavior using actions like "binding" or "phosphorylation" or "degradation" and simulating over time the differential equations governing the interactions of proteins in an entire cell. Above this, we could abstract away the population sizes, ignoring quantity and only dealing with the state of each type of protein as a Boolean variable – active or inactive, on or off – and with interactions between proteins as simple activation or inhibition.

As we go up, we sacrifice fine-grained understanding in favor of simpler abstractions; at the far end, "genotype produces phenotype," for example, smooths over all of the layers of detail and mechanism that take place beneath its surface, while not offering much predictive power about those mechanisms.
As the adage goes, "All models are wrong, but some are useful." In order to solve problems involving entire cells or entire organisms, we must choose a level of abstraction to model at, because it is computationally infeasible to model large systems at the microscopic level of physics and physical interactions.

Researchers create biological models because of their usefulness in answering research questions, and researchers in different fields may want answers to different questions: this determines the abstraction level of their models. Physical simulations of protein interactions may be useful to a pharmaceutical company looking to design a new drug, or to a synthetic biologist looking to design a novel protein. Differential equations may be useful to a cancer biologist looking to discover the interactions among proteins that lead to the development of cancer in a single cell. Boolean networks, being the least computationally-intensive of these, may be useful to a researcher looking to investigate interactions among an entire population of cells.

Our work will specifically consider rule-based models of protein-protein interactions, which are well-suited for modeling large systems of proteins in a single cell and which find applications in cancer biology. These models consist of a list of rules; each rule defines an interaction that might take place among one or more chemical agents (usually proteins), and the reaction rate at which that interaction takes place. Reaction rules are suitable for abstracting over some of the microscopic details involved in how proteins interact, but might not be able to capture macroscopic
details such as cell-cell interactions. Executing the rule-based model consists of simulating the set of differential equations that this list of rules and reaction rates specifies.

Figure 1-3: Rule-based modeling: reaction rules are executed to simulate a run of the system.

1.3 Programming models automatically

Despite their usefulness, computational models can become obsolete over time as new discoveries are made. However, creating and updating models by hand is time-consuming: much like in traditional programming, the process is slow and human error can introduce bugs. For these reasons, biology could benefit from tools that facilitate the automatic programming of computational models, especially in a manner that automatically incorporates new scientific discoveries.

However, until biology has a bigger culture of reporting results in computationally friendly formats, data will only be available in the “legacy” format, scientific papers. Currently, there are a handful of research projects investigating the automated creation of executable biological models by using natural language processing (NLP) to scrape information from research papers. Although this is a promising technique for the automatic generation of models, there are still some obstacles to overcome before it can produce high-quality computational models for researchers. Our project aims to overcome those obstacles.
1.4 The problem: turning facts into rules

One obstacle is that facts extracted by NLP may not be able to feed directly into a rule-based model. Models need to be constructed from mechanistic rules: “Raf phosphorylates MEK at site Ser-222,” for example, is easy to transform into an executable model simulating levels of Raf and ERK and their states. In contrast, the facts extracted by NLP may take many forms, not all of which are clear-cut mechanistic rules. NLP may produce non-mechanistic rules, for example, such as the following facts about the Ras-Raf-MEK-ERK cancer pathway:

- **Active ERK1 phosphorylates RSK.** This seems mechanistic at first – phosphorylation reactions are common – but we’re missing one key piece: what does it mean for ERK to be active?

- **MEK phosphorylates the ERK protein family.** This is type error; which members of the ERK protein family does MEK phosphorylate, and by what mechanism?

- **Addition of EGF causes activation of ERK1.** This tells us that activity in one protein causes activity in another protein, but we don’t know how many causal steps take place in between; in fact, EGF only activates ERK1 through a pathway involving several intermediate receptors and signalling proteins.

Some facts produced by the NLP aren’t even “rules” at all, but are still useful for constructing a model and disambiguating other facts. We call these *domain knowledge*. Some examples:

- **When ERK1 is phosphorylated, it is active.** This is not a rule, but it helps us decode my earlier “Active ERK1 phosphorylates RSK” example.

- **ERK1 and ERK2 are in the ERK protein family.** This is not a rule, but it helps us decode my earlier “MEK phosphorylates the ERK protein family” example.
- **S151D-mutated ERK1 behaves as if always phosphorylated.** This only makes sense as an attachment to existing rules about how phosphorylated ERK1 behaves.

We want our final model to only contain mechanistic rules. For example, the above collection of non-mechanistic rules and domain knowledge is consistent with the following list of mechanistic rules:

- **MEK phosphorylates ERK1.**
- **MEK phosphorylates ERK2.**
- **Phosphorylated ERK1 phosphorylates RSK.**
- **Phosphorylated ERK2 phosphorylates RSK.**
- **S151D-mutated ERK1 phosphorylates RSK.**

The problem is that NLP output can be messy: it contains mechanistic rules, non-mechanistic rules, and domain knowledge, all of which must somehow be woven together in order to create an accurate model composed only of mechanistic rules. As a solution, we have created a tool that uses logical inference to deduce a set of clear-cut mechanistic rules that are consistent with the messier input facts NLP gives us.

### 1.5 The solution: logical inference over chemical semantics

A good computational model should contain all available facts pertaining to the system it models, including facts that can’t be immediately converted into the rules of a rule-based model. Our solution allows us to produce models that incorporate all of this available knowledge, by performing logical inference in order to deduce which mechanistic rules are implied by the available facts.
A naive implementation of this solution would be to come up with a set of deduction rules that can combine facts together in order to, ultimately, come up with the complete list of valid mechanistic rules. Consider, for example, these two facts from the above scenario:

- **Active ERK1 phosphorylates RSK.**
- **When ERK1 is phosphorylated, it is active.**

If I had the following inference rule in my arsenal, I would be able to deduce the conclusion I want to deduce: that phosphorylated ERK1 phosphorylates RSK.

Figure 1-4: Inference rule that can combine two facts into a mechanistic rule.

\[
\text{Active } A \text{ performs reaction } B. \quad \text{Phosphorylated } A \text{ is active.}
\]

\[
\text{Phosphorylated } A \text{ performs reaction } B.
\]

To complete the naïve approach, we would develop a set of inference rules describing the deductions that we can make among all of the kinds of facts that we might come across. Then we could add new facts to our list of facts by looking for matching inference rules, until there are no more facts to add. At that point, we could output the mechanistic rules that we had inferred, and be more confident in the completeness of our model.

Figure 1-5: Example inference rules that could be used to make logical deductions about collections of facts, in the naïve approach.

\[
A \text{ phosphorylates } B. \quad \text{Phosphorylated } B \text{ is active.}
\]

\[
A \text{ activates } B.
\]

\[
A \text{ binds } B. \quad B \text{ indirectly binds } C. \quad A \text{ binds } B. \quad A \text{ does not bind } B.
\]

\[
A \text{ indirectly binds } C. \quad \perp.
\]

\[
A \text{ binds } B. \quad A \text{ indirectly activates } B. \quad B \text{ indirectly inhibits } A.
\]

\[
A \text{ and } B \text{ exhibit negative feedback.}
\]
However, our quest will not be to attempt to enumerate all such possible inference rules as the naïve approach would. For one thing, this approach would not scale: if there are $n$ kinds of fact – that is, sentences with variables, like "$A$ activates $B$" or "$A$ binds $B$" – then there will be $\Omega(n^3)$ different candidate inference rules. And that's only for two-input inference rules like $x \land y \Rightarrow z$, not to mention three-input inference rules like $x \land y \land z \Rightarrow w$ and so on!

For another thing, this approach would be buggy: one could easily include an inference rule that is overly generous in what it permits, and make the entire implication system unsound.

To improve the naïve approach, we can give each kind of fact in these inference rules a definition, specified in a logical language that permits us to describe the interactions between chemical agents. Having done this, we can use this logical language in order to prove the soundness of individual inference rules, or in order to deduce properties of entire collections of facts. That way, the implications we prove will be grounded in a rigorous chemical semantics, and ensured to be sound within those semantics. Additionally, this solves our scaling problem: we only need to formalize the definitions of each of our $n$ kinds of fact, which is better than deliberating over $\Omega(n^3)$ inference rules.

*Syndra* is a tool we have created that executes this improved approach. It implements a logic language for reasoning about the semantics of biological models, and implements definitions for several kinds of facts as predicates in that logic language. From there, it can reason about implications between predicates in order to prove inference rules, and it can compute a rule-based model satisfying a collection of given biological facts.

With this tool, we are hopeful for a future where biological modeling becomes a scalable, accurate, and adaptable tool in the biology researcher’s toolkit.
Chapter 2

Related Work

Here, we will give an overview of related works in order to show that this project's goal of converting general biological facts into rule-based models has not before been solved, and that this project's approach of basing logical inference on the chemical semantics of facts has not before been tried. Particular works of interest include Kappa, a programming language for rule-based models; previous tools for static analysis on Kappa; and the current state of research into generating models automatically with NLP.

2.1 Modeling with Kappa

Kappa is one language that can be used to write rule-based models for modeling networks of interactions between proteins [11]. Kappa has syntax for indicating protein structures such as binding sites and enzyme active sites, and is capable of describing operations including binding, phosphorylation, and catalysis in its rules for reactions. Here is an example of a Kappa rule:

\[ A(\text{Site}_1^u), B(\text{Site}_2) \rightarrow A(\text{Site}_1^u!1), B(\text{Site}_2!1) @ 0.2 \]

This rule indicates that the structures on the left side—agent A with unphosphorylated site 1, and agent B with site 2—can become the structures on the right side—
the same agents, bound together (with binding sites indicated by $.1$). Furthermore, it specifies that this reaction occurs at a relative rate of 0.2 [2].

KaSim [4] is the compiler for Kappa: it stochastically simulates the evolution over time of a system governed by some set of Kappa rules, by choosing reactions to apply in proportion to their relative rates.

Kappa is a great choice of language for both rule-based models and for analysis of rule-based models because of its clearly-defined operational semantics. Furthermore, there is a great ecosystem of work built up around Kappa: in addition to the static analyses discussed below, there are tools like Kami [3], a tool for aggregating pieces of biological knowledge and related model components, and PySB [6], a Python framework for constructing biological models that can be compiled into Kappa and that makes it easy to write macros for Kappa models.

2.2 Other modeling paradigms

We have chosen to work with rule-based models, a modeling paradigm particularly suited for modeling protein signalling pathways on a single-cell level. However, there are other paradigms for biological modeling that similarly operate at the level of the cell.

Differential equations are one such modeling paradigm. Instead of writing down higher-level rules, it is possible to instead directly write down the differential equations that simulate a signalling pathway of choice. Although this could allow a model to be more fine-grained, perhaps permitting some differential equations that a rule-based modeling system would not, it comes at a cost: rule-based modeling languages allow for abstraction, composition, and modularity.

We choose to analyze rule-based models instead of differential equations because the trend in executable biology has been towards using rule-based models instead of differential equations exactly because abstraction makes programming these models easier. Furthermore, there are tools to add even more abstraction and modularity to Kappa: tools such as PySB, a Python library for manipulating Kappa
models, permit writing macro commands and parametrizable functions for Kappa models in Python [6]. Abstraction allows for code reuse without code duplication.

Boolean networks are another modeling paradigm. Unlike differential equations, which fall below rule-based models on the abstraction hierarchy, boolean networks are one abstraction higher. Instead of modeling populations of proteins in different states, boolean networks model the majority state of an entire population of proteins with a single boolean: active or inactive, on or off. Interactions then become simple all-or-nothing rules: A activates B all at once, B inhibits C all at once.

We choose to analyze rule-based models instead of Boolean networks because rule-based models permit mechanistic descriptions of biological systems; knowledge about the binding interactions between proteins, for example, can be encoded into a rule-based model, and used to inform more accurate simulations of the system's behavior. In contrast, a Boolean network would discard this information.

2.3 Analysis of Kappa

Our work is not the first research into logical analysis of Kappa models. Feret 2007 investigated reachability analysis for Kappa: this is an analysis that can determine which complexes of one or more molecules are reachable by some conceivable sequence of reactions, no matter how unlikely [14]. This analysis offers insight into what kinds of molecule complexes will be combinatorially possible according to the system's reaction rules, and can also be used to detect whether any reactions are impossible to trigger. Danos, Feret, Fontana, and Krivine 2008 elaborate on this work, developing an abstract interpretation for biological signalling networks in order to explore the set of complexes that are reachable by a Kappa ruleset [10].

In other work, Camporesi 2013 investigated techniques for reducing the combinatorial complexity of models with many possible molecular complexes, determining ways of coarse-graining models in order to trade off accuracy for efficiency [8].

Finally, Feret 2014 [15] notes that detecting symmetries in models can help to
reduce the complexity of simulating those models, and presents an abstract interpretation analysis for finding symmetries in Kappa graphs.

Our work is unique from these analyses because whereas their starting point is a complete Kappa model to analyze, we do not start with a complete Kappa model. Instead, we start with non-mechanistic biological facts, and must proceed from there to constructing a satisfactory Kappa model. Thus, we are tackling a different problem and must develop new tactics.

2.4 Big Mechanism ecosystem

As alluded to before, programming biological models by hand, fact by fact, takes time, and since research continues onward at a breakneck pace, it is unlikely that any such model might stay up-to-date for very long after it has been constructed.

With the Big Mechanism program, the U.S. Defense Department aims to fix this problem, by facilitating the automated creation of executable biological models [9]. The ultimate goal of this initiative is to research technologies that would enable the construction of dynamic models into which new research is immediately and automatically integrated. These models should be able to generate causal explanations for biological processes and aiding in the development of new research hypotheses. For now, Big Mechanism project focuses on constructing such a model for knowledge about signaling pathways in cancer biology [9].

Although languages like Biological Expression Language (BEL) or Biological Pathways Exchange (BioPAX) exist for allowing paper results to be described in a computer-readable syntax, most biological knowledge is not yet available in these computationally friendly formats. Instead, this knowledge is wrapped up in scientific papers. INDRA a project in the Big Mechanism ecosystem that scrapes facts from NLP of scientific literature, adds in facts from databases such as BEL and BIOPAX, then exports the mechanistic rules that it gathers to a model written in PySB, a Python modeling language that has Kappa as its backend. [6] Not all of the rules that INDRA collects are mechanistic, however; Syndra's role is to use logical
inference to deduce what rules may remain.

We have been closely collaborating with the INDRA developers in order to devise a tool that can meet their needs, and we demonstrate that our system can integrate with INDRA as a frontend. The techniques that we demonstrate in this work could be extended to work alongside other systems similar to INDRA.

Figure 2-1: The INDRA pipeline, with Syndra as intermediate analysis tool.

2.5 Bringing it all together

Our design choices are influenced by this ecosystem of related work and by the technologies that already exist for us to leverage. In particular, Syndra consists of two main components. The first component is translation of statements into predicates – in order to do this, we need a logic language to use to express these predicates. We choose *lota* for this purpose, a language designed by Adrien Husson and Jean Krivine specifically for the purpose of expressing predicates over sets of Kappa models [16]. After that, the second component is an implementation of the logic language so that we can use it to make deductions; we choose to use *the Z3*
**Theorem prover** as our tool for this. In the next two chapters, I will discuss the design and architecture of these two components of our system, and I will discuss how these two technologies (Iota and Z3) are utilized.

Figure 2-2: Two components: converting biological facts to predicates, converting predicates to models

<table>
<thead>
<tr>
<th>Mechanistic rules</th>
<th>Predicates over models</th>
<th>Space of possible models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-mechanistic rules</td>
<td>Domain knowledge</td>
<td></td>
</tr>
</tbody>
</table>

Ingredients:  
1: Logic language for specifying predicates over Kappa models  
2: Tool for analyzing predicates

![Iota](image1)  
![Z3](image2)
Chapter 3

Predicates over biological models

As a reminder, our overall goal is to be able to clean up collections of biological facts into rule-based models. We have chosen to approach this problem by converting each biological fact into a predicates over a space of possible rule-based models, so that we can use those predicates in order to make logical deductions.

Why predicates? With every new biological fact that we have, we should be able to add more nuance to the model that we’re building. The more information we have, the fewer candidate models there are: more and more models get eliminated for being inconsistent with the known facts. New biological facts constrain the space of possible models, and predicates are a way to express the nature of these constraints.

Additionally, each biological fact has a semantic meaning in terms of the chemical interactions it describes. We can’t get at this semantic meaning unless we have a language for describing it, but we need these meanings in order to reason deductively about sets of distinct but related biological facts. By using a formalized logic language for describing the space of possible models, with a clearly-defined semantics for each of the structures that logic language can express, we can go about this task in a rigorous way.

Here, I’ll discuss the details of Iota, the language we chose to describe predicates and to carry out this job of connecting biological facts to their chemical semantics.
3.1 Predicates over the space of models

A predicate is only true for some subset of all possible biological models; it constrains the space of possible biological models. By combining enough facts, we should find that the space becomes narrower and narrower, until there are only one or a few candidate models remaining. We find that we can convert biological facts, even those that are not mechanistic rules, into predicates – so, this approach allows us to extract a model from non-mechanistic facts.

If a predicate is satisfiable, that means that there exists some model or models under which it is true. We will call a predicate ambiguous if there are multiple such models, or unambiguous if there is only one. An unsatisfiable predicate is one which is not true for any model, and a valid predicate is one that is true for all models.

3.2 Iota

We need a logic language with clearly-defined semantics that allows us to express predicates over the space of possible biological models. Iota is such a language: it is a “meta-Kappa” designed by Adrien Husson and Jean Krivine, specially designed for the purpose of working with constraints over possible Kappa models [16]. Iota represents Kappa models as sets of rewrite rules on graphs, where each graph represents a possible state of a biological model.

This specialization is one reason we have chosen to implement Iota specifically: Iota predicates can be interpreted as Kappa models, without a need to translate between two different modeling languages or to design a logic language from the ground up. Another advantage of Iota is that its semantics are very precisely defined: the Iota paper provides rules for interpretation of Iota predicates, building up the set of models that a predicate represents by providing rules for breaking predicates apart and combining the interpretations of the sub-parts.
3.2.1 Semantics

Each Iota predicate can be interpreted as a set of models. In order to facilitate this, Iota devises a way of representing possible Kappa models. Each model is a set of rules: these are the reactions that may take place.

The data structure used to represent a rule is important. In Kappa, every rule represents a possible graph rewrite: for example, two previously-unbound agents may become bound, adding an edge between them. Iota represents the state of a system as a graph, with one type of edge for binding connections and another type of edge for parent-child relationships (as in the relationship of an enzyme to its active site). Iota also has a notion of an action, which is a change that may happen to a graph. The Iota paper describes how to construct any action that a Kappa rule may involve [16].

Thus, each Kappa rule is represented as a (graph, action) pair: the graph, as a template for what kind of original molecular state the rule may apply to, and the action, for what transformation is applied to that molecular state. The transformed graph – representing the post-rule molecular state – does not need to be specified, since it is uniquely determined by the combination of action with pre-rule graph. Each Iota predicate specifies a set of satisfying models, which is equivalently a set of sets of rules, which is equivalently a set of sets of (graph, action) pairs.

The Iota paper describes in detail the syntax and semantics of predicates in Iota, first by describing a set of atomic predicates which constrain over rules, then describing how to combine atomic predicates with other higher-level operations to create predicates which constrain over models. It also describes the interpretation of each of those predicates – mechanistic rules for constructing the set of models that satisfy a given predicate, by breaking the predicate down into smaller parts, interpreting those parts recursively, and combining the parts using the appropriate operation. The following figures depict how to interpret each predicate and atomic predicate according to the Iota semantics.
Figure 3-1: Semantics of Iota’s atomic predicates [16]

\([\top], := \mathcal{G} \times \mathcal{A}\)

\([\bot], := \emptyset\)

\([x = y], := \begin{cases} 
[\top] & \text{if } I(x) = I(y) \\
[\bot] & \text{otherwise}
\end{cases}\)

\([\Sigma_i(x)], := \{g \mid \lambda(I(x)) \in \Sigma_i \times \mathcal{A}\} \times \mathcal{A}\)

\([A(x)], := \{g \mid \lambda(I(x)) = A\} \times \mathcal{A}\)

\([x y], := \{g \mid \{I(x), I(y)\} \in P_x\} \times \mathcal{A}\)

\([x y], := \{\{g, a\} \mid \{I(x), I(y)\} \in P_x\}\}

\([\text{do}(x y)], := \mathcal{G} \times \{I(x), I(y)\}\)

\([x \sim y], := \{g \mid \{I(x), I(y)\} \in L_y\} \times \mathcal{A}\)

\([\overline{x \sim y}], := \{\{g, a\} \mid \{I(x), I(y)\} \in L_y\}\}

\([\text{do}(x \sim y)], := \mathcal{G} \times \{I(x)^{-1}(y)\}\)

\([\text{do}(x' y)], := \mathcal{G} \times \{I(x)^{-1}(y)\}\)

\([\text{has}(x)], := \{g \mid I(x) \in V_g\} \times \mathcal{A}\)

\([\overline{\text{has}(x)}], := \{\{g, a\} \mid I(x) \in V_g\}\}

\([\text{add}(x)], := \{\{g, \text{add}(I(x))\} \mid I(x) \notin V_g\}\)

\([\text{rem}(x)], := \{\{g, \text{rem}(I(x))\} \mid I(x) \in V_g\}\)

Figure 3-2: Semantics of Iota’s predicates [16]

\([P], := \{[P]\}\) for all atomic predicates \(P\).

\([\phi \lor \psi], := [\phi] \cup [\psi]\)

\([\phi \land \psi], := [\phi] \cap [\psi]\)

\([\phi \times \psi], := [\phi] \times [\psi]\)

\([-\phi], := \{S \mid S \in [\phi]\}\)

\([\forall x. \phi], := \begin{cases} 
\times_{a \in V} [\phi]_{I(\omega)}, & \text{if } \phi \text{ is choice-free.} \\
[\forall x. \psi] \cup [\forall x. \phi], & \text{if } \phi \text{ is } \psi\text{-hesitant.} \\
\emptyset, & \text{otherwise.}
\end{cases}\)

\([\exists x. \phi], := \begin{cases} 
\times_{a \in V} [\phi]_{I(\omega)}, & \text{if } \phi \text{ is choice-free.} \\
[\exists x. \psi] \cup [\exists x. \phi], & \text{if } \phi \text{ is } \psi\text{-hesitant.} \\
\emptyset, & \text{otherwise.}
\end{cases}\)

Note that the interpretation of an atomic predicate is a set of graph-action pairs.
\((g, a)\): atomic predicates thus constrain over possible Kappa rules. Similarly, note that the interpretation of a predicate is a set of sets of graph-action pairs, or equivalently a set of Kappa models: predicates constrain over possible Kappa models. Some of these interpretations are defined by combining the interpretations of two smaller predicates using operations such as \(\cup\) and \(\times_n\). For more information on some of these operators, consult their definitions in the Iota paper.

Having chosen Iota as a formalism to work with, we move on to the next step: architecting a system to implement the analysis of Iota predicates.
Chapter 4

Architecture and implementation

Now that we have chosen Iota as a logic language to use to describe chemical interactions, the primary task that remains is to implement Iota in such a way that we can check the satisfiability of Iota predicates. Ideally, we’d like our system to have an API that:

- allows the user to build predicates in Iota’s syntax
- allows the user to check a predicate’s satisfiability
- can construct a rule-based model satisfying any satisfiable predicate

With an API such as this one, we will have the ability to solve our motivating problem: we can build predicates to represent each of the facts that NLP gives us, and acquire the model satisfying those facts.

To build this API, we use the Z3 theorem prover to power our logical deductions. Z3 is an SMT solver developed by Microsoft Research [13], and can easily interface with Python via its Python bindings [5]. Z3 is well-suited for this problem because it permits the user to assert statements about collections of variables, then to check whether those assertions are satisfiable or not. The system is versatile enough to be used for everything from solving small systems of arithmetic constraints to program synthesis. Conveniently, Z3 is powerful enough to tackle problems described by first-order logic, and Iota was proven by its authors to be reducible to first-order
logic [16]. By designing Z3 datatypes to represent Iota's graph-action pairs, then translating Iota predicates into Z3 assertions over those datatypes, we have sufficient power to check the satisfiability of Iota predicates.

The code for my Iota implementation can be found at https://github.com/csvoss/syndra/tree/master/engine. Here I'll discuss how the implementation was built, and discuss some of the design decisions that were made.

4.1 Predicates

In order to implement this API, I designed an abstract class Predicate. Every instance $x$ of Predicate supports the following methods:

- $x$.get.predicate() returns a Z3 expression representing the assertion $x$
- $x$.check.sat() checks a predicate's satisfiability
- $x$.get.model() constructs a rule-based model satisfying this predicate
- $x$.check.implies(y) checks whether predicate $x$ implies predicate $y$

All of these methods are actually implemented in the abstract class Predicate. The only component of Predicate which is "abstract" is an abstract helper method, $x$.assert(pairset, interpretation). This is a private helper method, and it must be implemented differently by each subclass of Predicate. _assert is called by get.predicate; it translates the current Predicate object into a Z3 assertion, by recursively calling _assert down each branch of the syntax tree. The variable pairset in the inputs to _assert refers to a set of graph-action pairs for this predicate to assert over, and the variable interpretation refers to a mapping from variables to their values as nodes in the graph (the interpretation of each variable); see Datatypes for more details on these.

The subclasses of Predicate each represent an Iota operation, and can be combined together to form the syntax tree of an Iota predicate. Each subclass has a dif-
ferent interpretation of _assert that combines the predicates of its child branches in the syntax tree together in a unique way.

Figure 4-1: Subclasses of Predicate; see predicate.py for details.

ForAll(predicate)
Exists(predicate)
And(predicate1, ...)
Or(predicate1, ...)
Join(predicate1, ...)
DontKnow(predicate1, ...)
Not(predicate)
Implies(predicate1, predicate2)

For the most part, these (see Figure 4-1) are exactly as they are defined in the Iota paper. Note that Join is a special operator defined in the Iota paper, and represented there by the operator &. It is used to combine the actions of multiple predicates together. DontKnow is also a special operator defined in the Iota paper, and represented by the operator V; it is used when at least one predicate over the entire model must hold, as kind of an "or" over distinct situations. (In contrast, Or is used to combine two predicates which pertain to a single rule, and to require that at least one of those predicates must hold over that rule.) Finally, Implies is not defined in the Iota paper; I added this in to make it easier to check whether one Iota predicate implies another Iota predicate is true.

An example implementation of a subclass of Predicate is below. This subclass, DontKnow, accepts any number of child predicates for convenience's sake, automatically rearranging them into a binary tree of two-predicate DontKnow operations. Its _assert helper method is a simple Or of its two child predicates. This illustrates the role of Z3: each Iota predicate uses Z3 operations in order to create the appropriate constraint.

class DontKnow(Predicate):
    """'V' ("don't know" operator) two Iota predicates together.""
    def __init__(self, *preds):...
After constructing any predicate and using _assert to retrieve its representation as a Z3 constraint, we can use Z3 to check the satisfiability of that constraint and, if satisfiable, to obtain a model satisfying that constraint [13]. Predicate's implementations of get_model, check_sat, and check_implies do exactly this, delegating out to Z3 using a helper interface I wrote in solver.py.

### 4.2 Atomic Predicates

In Iota, each atomic predicate can also be used as a predicate. I defined an abstract class AtomicPredicate, and defined an automatic wrapper for each subclass of AtomicPredicate that generates a valid subclass of Predicate. The only method required by the AtomicPredicate abstract class is _assert(pair, interpretation), which must be implemented by each AtomicPredicate subclass. The variable pair refers to a graph-action pair datatype passed in for this predicate to assert over.

Each of these subclasses (see Figure 4-2, below) represents one of the atomic predicates as defined in the Iota semantics. Atomic predicates with "Pre" in the name are preconditions: constraints on the left-hand side of a reaction rule, the graph in a graph-action pair. Similarly, atomic predicates with "Post" in the name are postconditions: constraints on the right-hand side of a reaction rule, produced by applying action to graph for a given graph-action rule. "Do" is used to require that the action in a graph-action pair contain a certain atomic action component.

"Labeled" is used to require that agents have labels attached; for example, "phosphorylated" and "active" can be manipulated as labels. "Named" was not defined in Iota, but I include it here as a syntactic sugar over "Labeled" to make it easier to define labels that must be unique. "Named" can be used to require that a variable
Figure 4-2: Subclasses of AtomicPredicate; see atomic_predicate.py for details.

Top()
Bottom()
Equal(variable1, variable2)
PreLabeled(variable, string)
PostLabeled(variable, string)
PreUnlabeled(variable, string)
PostUnlabeled(variable, string)
PreParent(variable1, variable2)
PostParent(variable1, variable2)
DoParent(variable1, variable2)
PreLink(variable1, variable2)
PostLink(variable1, variable2)
DoLink(variable1, variable2)
DoUnlink(variable1, variable2)
Named(variable, string)
PreHas(variable)
PostHas(variable)
Add(variable)
Rem(variable)

used elsewhere in the formula has a specific protein name, or to ensure that two distinct variables refer to the same agent.

"Parent" and "Link" are used to require that two agents in a graph are bound together in either a parent-child edge (as, for example, a protein may be bound to its reactive side) or in a linking edge (as, for example, two separate proteins may be when they chemically bind together). "Has" simply asserts that a agent is present in a graph; "Add" and "Rem" assert that a rule's action involves adding or removing an agent from the graph.

4.2.1 Wrapping atomic predicates

The Iota paper provides a semantics for converting atomic predicates into predicates:
Figure 4-3: Predicate semantics of each atomic predicate [16].

\[[P]_I := \{[P]_I\}\] for all atomic predicates P.

Recalling that the interpretation of a predicate is a set of models, and the interpretation of an atomic predicate is a set of graph-action pairs, this indicates that in order to interpret an atomic predicate P as a predicate, we take the set consisting of one element: its interpretation as an atomic predicate.

I wrote an automatic wrapper function that uses each subclass of AtomicPredicate in order to define a separate valid subclass of Predicate. However, Predicate and AtomicPredicate each have different APIs for their respective _assert functions: Predicate asserts over a space of possible models, whereas instead AtomicPredicate asserts over a space of possible graph-action pairs. To implement this transformation in Python, I had to create subclasses of Predicate whose _assert helper methods delegated out to the _assert helper method of the appropriate subclass of AtomicPredicate.

### 4.3 Testing

Unit tests verifying the correctness of this implementation are defined in the files test_solver.py, test_atomic_predicate.py, and test_predicate.py. Some of these unit tests work by checking correctness for simple examples; others work by asserting that the relationships described by the Iota semantics for each Iota operator hold when combining two complicated predicates together with that operator.

### 4.4 Datatypes

Finally, in order to implement the conversion of each predicate to Z3 assertions, I needed to represent Iota's variables as variables in Z3, and for that I needed to define a Z3 datatype representing a biological model to apply constraints to, the
fundamental structures in an Iota model. This needed to be build up part-by-part, from the smallest components of the graph all the way up to an entire model.

- **Node**: an integer uniquely identifying a vertex in a graph

- **Edge**: a pair of Nodes

- **Nodeset**: an ArraySort (Z3 array) mapping Nodes to booleans

- **Edgeset**: an ArraySort mapping Edges to booleans

Note that because we need to reason about sets of sets, I could not represent sets of, for example, nodes as functions from Node to Bool; this would result in higher-order functions later on as I needed to reason about sets of sets of nodes, and Z3 does not allow the definition of higher-order datatypes. Instead, I used ArraySort to define a set of nodes as an array mapping node values to booleans. This has not made computation times noticeably slower.

- **Label**: an integer uniquely identifying each label such as “phosphorylated” or “active”

- **Labelset**: an ArraySort mapping Labels to booleans, for storing the set of labels that some agent might be labelled with

- **Labelmap**: an ArraySort mapping Nodes to Labelsets

Now, we have enough to start defining bigger datatypes:

- **Graph**: a custom Z3 datatype containing a Nodeset ‘has’ for the vertices it contains, an Edgeset ‘parents’ for parent-child edges, an Edgeset ‘links’ for chemical binding edges, and a Labelmap ‘labelmap’ for keeping track of which vertices have which labels

- **AtomicAction**: a custom Z3 datatype enumerating each of the atomic actions that Iota defines
- **Action**: an ArraySort mapping AtomicActions to booleans

- **Pair**: a custom Z3 datatype containing a graph and an action; constraints over a single Pair represent sets of graph-action pairs

- **Pairset**: a function from Pairs to booleans; constraints over a Pairset represent sets of sets of graph-action pairs, and thus a set of biological models

Our predicates define constraints over this last, biggest datatype, Pairset. Lastly, we have datatypes for the variables that Iota predicates constrain over, and for an *interpretation*, which is a candidate mapping from variables to nodes in the graph:

- **Variable**: a custom Z3 datatype with an attribute 'get_name' storing the variable's name, by unique integer identifier

- **Interpretation**: a function from Variables to Nodes

With all of these datatypes, we have enough structure built up in Z3 to construct AtomicPredicates and Predicates, and to evaluate their truth.

In the next chapter, I'll discuss how being able to construct and evaluate these predicates allows us to build models from NLP facts, and I'll show how we integrated this system with an existing tool for extracting NLP facts from biological papers.
Chapter 5

Applications and usage

Syndra can be applied to the problem of converting biological facts into rule-based models, solving the original problems with NLP model generation that motivated us to carry out this research. I’ve shown how we chose and implemented Iota, a logic language for describing predicates over biological models; here, I’ll discuss how we can use this tool with real output produced by NLP model generators in order to produce biological models.

5.1 Macros

One notable gap that we haven’t covered yet: we can convert predicates into models, but how do we convert biological facts into predicates?

In order to make it easier to create and reason about Iota predicates based on the higher-level English statements that NLP produces, Syndra defines a collection of macros. These are functions which produce Iota predicates for some common biological facts. Currently, the following macros have implementations specified by Syndra (see macros.py):

- \( \text{directly\_binds}(A, B) = A \text{ binds } B. \)
- \( \text{directly\_activates}(A, B) = A \text{ activates } B. \)
- \( \text{directly\_phosphorylates}(A, B) = A \text{ phosphorylates } B. \)
In order to define each of these macros, we choose a way to express the statement as an Iota predicate. For example, the implementation of the phosphorylation macro \texttt{directly phosphorylates}(A, B) requires that the model contain at least one rule in which on the left hand side, \(A\) is active and \(B\) is not phosphorylated, and on the right hand side, \(A\) is still active and \(B\) has become phosphorylated:

\[
\text{predicate.Exists} (\text{predicate.Implies} (\text{predicate.Named} (A, \text{name\_a}), \\
\text{predicate.Implies} (\text{predicate.Named} (B, \text{name\_b}), \\
\text{predicate.And}(\text{predicate.PreLabeled}(A, \text{ACTIVE}), \\
\text{predicate.PreUnlabeled}(B, \text{PHOSPHORYLATED}), \\
\text{predicate.PostLabeled}(A, \text{ACTIVE}), \\
\text{predicate.PostLabeled}(B, \text{PHOSPHORYLATED})))))
\]

Biology labs using computational tools often have different preferences for how to model certain systems and the assumptions that they make. Therefore, it is helpful that Syndra allows custom definition of macros over predicates. Researchers can choose their own preferred definitions, encode those definitions as predicates using macros, and have the resulting logical deductions still be sound according to the definitions so specified.

Unit tests for these macros can be found in \texttt{test\_macros.py}.

### 5.2 Deducing implications

Using these macros, Syndra can detect whether two or more biological facts \(x\) and \(y\) imply a third biological fact \(z\). We create a predicate for each biological fact using the appropriate macro, then check whether \(x \land y \Rightarrow z\) is valid – that is, check whether \(\neg(x \land y \Rightarrow z)\) is not satisfiable. The following sample code uses Syndra to perform this operation for the example of checking whether \(x = \text{"MEK phosphorylates ERK"}\) and \(y = \text{"phosphorylated ERK is active"}\) together imply \(z = \text{"MEK activates ERK"}\):
>>> from syndra.engine import macros, predicate

>>> x = macros.directly_phosphorylates("MEK", "ERK")
>>> y = macros.phosphorylated_is_active("ERK")
>>> z = macros.directly_activates("MEK", "ERK")

>>> x_and_y_imp'y;_z = predicate.Implies(predicate.And(x, y), z)

>>> print x_and_y_imp'y;_z.check_sat()
True

>>> print predicate.Not(x_and_y_imp'y;_z).check_sat()
False

5.3 Deducing inconsistencies

Syndra can also determine whether including two biological facts in a model together would be unsound. For example, it is not possible to combine the facts $x = \text{"MEK phosphorylates ERK"}$ and $\neg x = \text{"MEK does not phosphorylate ERK"}$ into a coherent model. We can make a predicate combining these two facts, and check that it is in fact unsatisfiable:

>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
>>> y = predicate.Not(i)

>>> x_and_y = predicate.And(x, y)

>>> print x_and_y.check_sat()
False

5.4 Interfacing with INDRA

We have used Syndra as a tool to assist in the logical analysis of models output by INDRA, an actual NLP-based automatic model generator and part of the Big Mechanism initiative [1]. INDRA gathers data by performing natural language processing on biology papers with the TRIPS parser and by including facts from databases
including BEL (Biological Expression Language) and BioPax. The INDRA developers presented us with ideas for analyses that Syndra could facilitate; the ability to check implications between different NLP-generated statements was one such idea. In order to do this for INDRA’s output, we converted INDRA statements – Python objects produced by the INDRA software, each of which represents an individual biological fact – into Iota predicates, by determining the appropriate macro to apply to each INDRA statement. From there, we can check implications between Iota predicates as usual.

For example, Syndra can take in a list of the following three INDRA statements, representing a small system in which protein MAPK1 is only activated if it is phosphorylated by MAP2K1 at both at Thr-183 and Tyr-185, and verify that they all imply the last INDRA statement, which asserts that MAP2K1’s kinase activity increases the kinase activity of MAPK1:

Phosphorylation(MAP2K1, MAPK1, PhosphorylationThreonine, 183)
Phosphorylation(MAP2K1, MAPK1, PhosphorylationTyrosine, 185)
ActivityModification(MAPK1, ['PhosphorylationThreonine',
  'PhosphorylationTyrosine'], ['183', '185'], increases, Activity)

↓

ActivityActivity(MAP2K1, Kinase, increases, MAPK1, Kinase)

The code supporting the conversion of INDRA statements into Iota predicates can be found at engine/statements_to_predicates.py.

Additionally, Syndra can take in a list of INDRA statements, convert them all to predicates, and return the corresponding model. Examples of how to do this are in interface_indra_to_syndra.py.

INDRA is under active development; as more features are added to INDRA, the library of macros supported by Syndra can easily be expanded to accommodate them.
Chapter 6

Conclusion

We've presented the design and implementation of Syndra, a tool for translating biological facts into Iota predicates in order to find complete models satisfying sets of biological facts. We described the design of Iota, a logic language devised for the purpose of specifying constraints over Kappa models. We implemented Iota predicates and Iota datatypes, and leveraged the Z3 bindings for Python in order to assert the predicates hold over the datatypes. We built macros describing common biological facts as Iota predicates, and showed that Syndra can reason correctly about implication relationships among these predicates. Finally, we can interface with INDRA and deduce a model from INDRA's statements.

In future work, we would like to see Syndra even better integrated into its purpose of converting facts from systems like INDRA all the way to Kappa or PySB models. Syndra outputs its models as Z3 datatypes; one low-hanging fruit would be to implement a function converting these Z3 models into PySB models.

In addition, the library of macros that Syndra supports could be expanded. There are a wide variety of concepts and pieces of domain knowledge that allow biologists to make inferences about the behavior of proteins: for example, a protein mutation that changes an amino acid to aspartic acid is "phosphomimetic," likely to make that site on the protein behave as if it is phosphorylated. We received a document listing several examples of these from one of our collaborators [17]; it would be great to demonstrate how Syndra can encode each of these domain knowledge.
examples into a predicate. With predicates for domain knowledge like these in hand, we would be able to make better inferences and better models.

More broadly, we are moving towards a future where research is increasingly carried out not by humans acting and reasoning alone, but by humans working together with the computational tools that assist them. Machine learning and AI are advancing very quickly, and it is not unlikely that in the future there will be advanced tools for reasoning about biological problems. Executable biological models are potentially very useful to this future: not only can they be executed, but they are also machine-readable, and thus easier for any potential future AI systems to hook into than human language alone.

With our work implementing Iota, we have developed a method by which biological knowledge can be converted from human-readable statements into machine-readable rules. This infrastructure could catalyze the creation of even greater computational tools in the future, tools which will build on freely-available rule-based biological models.
Bibliography


