A Tool for Automated Inference of Executable Rule-Based Biological Models

Chelsea Voss[1]

Computer Science Department
Massachusetts Institute of Technology
Cambridge, MA, USA [4]

Jean Yang[2]

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Walter Fontana[3]

Department of Systems Biology
Harvard Medical School
Boston, MA, USA

Abstract

Executable rule-based biological models can help researchers investigate and understand complex systems such as cellular signaling pathways, but often must be programmed by hand. Some research efforts aim to program them automatically using *facts* extracted from papers via natural language processing or other techniques. However, these facts cannot always be directly converted into the *mechanistic reaction rules* that rule-based models need. Challenges of processing these facts include removing redundancies, reasoning about facts that have the same meaning at different levels of abstraction, and detecting inconsistencies between statements for the purpose of disambiguation. Thus, there is a need for tools that can convert facts about signaling pathways into mechanistic rules in a logically sound way. We describe how to build an efficient and logically sound procedure for navigating collections of redundant or contradictory facts about signaling pathways using a first-order graph logic. Our tool can translate facts about signaling pathways into first-order logic predicates, check those predicates for satisfiability, and find models that satisfy those predicates. We test our system against a handful of use cases, and show that it can construct rule-based mechanistic models that are sound with respect to the semantics of the facts from which those models were constructed.

Keywords: executable biology, rule-based modeling, SMT solvers

[1] Email: csvoss@mit.edu
[2] Email: jyang2@cs.cmu.edu
[3] Email: walter_fontana@hms.harvard.edu
[4] Now affiliated with Pilot.com, Inc., San Francisco, CA, USA.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs

1 Introduction

Executable 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*. [10] Examples of executable models include logic programs, Boolean networks, and rule-based models. By *executable* models, we in fact refer specifically to *programs*, written in specialized programming languages designed specifically for making biological models. Unlike mathematical models such as systems of differential equations or correlation-based models, executable models allow experiments to be executed against novel input conditions, producing predictive power by encapsulating available human knowledge into executable code.

Executable models can be created to serve many areas of biology research. One topic to which they can be applied is *cellular signaling pathways*, the study of the network of protein-protein and protein-ligand interactions within cells. This area is particularly well-suited to executable modeling because *rule-based modeling*, a modeling technique which involves indirectly specifying differential equations by directly specifying transformations and binding interactions among reactants, provides an effective modeling strategy. [4] In addition to being executable, rule-based models of cellular signaling pathways are also conducive to *static analyses* such as reachability analysis [8] [5] or symmetry analysis [9]. Rule-based modeling will be our focus in this work.

Executable models are useful tools, but it's time-consuming for researchers to keep their models up-to-date with the continual pace of new *in vivo* research results. In many cases, a modeling program once written may progressively become more and more out of date with respect to current human knowledge, becoming obsolete software unless continual manual efforts are made to keep it up to date. Software obsolescence hinders computational biological models from being as useful as they could be.

Towards solving this hindrance, some ongoing research efforts [3] [13] [11] attempt to use natural language processing (NLP) in order to extract *facts* from the scientific literature and to construct models automatically based on those facts. However, we claim that this procedure alone cannot guarantee correctness. These extracted facts may not always be unambiguous: oftentimes acronyms, for example, can have many meanings which context disambiguates. Sometimes the full comprehension of one fact may first require the comprehension of many prerequisite facts, such as definitions and reaction mechanisms. There are interdependencies whose resolution may not always be clear-cut, so it is not always easy to translate natural language statements into the direct mechanistic rules that rule-based modeling languages require.

If this claim is correct, then there is a *gap* to be bridged between NLP-extracted facts and the final goal of complete and correct models of cellular signaling pathways. In response to this gap, we construct a tool that translates NLP-extracted facts into predicates in a first-order graph logic, then reasons about collections of those predicates in order to extract an executable rule-based biological model. Although we choose a specific ecosystem of NLP fact extraction system (INDRA [13]) and

rule-based model programming language (Kappa [4]) with which to integrate our tool, our tool demonstrates the feasibility of a more general strategy of approach: logical inference as a way to bridge the gap from NLP-extracted facts, enabling the construction of complete, correct, and up-to-date executable models.

2 Related Work

2.1 *Rule-based modeling*

Our work specifically pertains to *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. Executing the rule-based model consists of simulating the set of differential equations that this list of rules and reaction rates specifies.

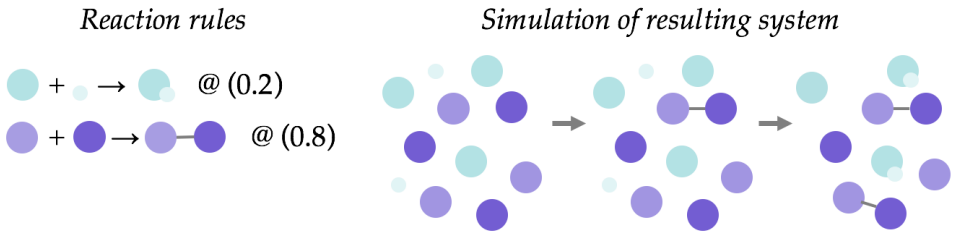

Fig. 1. Rule-based modeling: reaction rules are executed in order to simulate a run of the system.

Kappa is one programming language that can be used to write rule-based models for modeling networks of interactions between proteins [4]. We choose to implement our tool to work with Kappa models. Kappa is a great choice of language for both rule-based models and for static analysis of rule-based models because of its clearly defined operational semantics. Additionally, Kappa has precisely-defined syntax for indicating protein structures such as binding sites and enzyme active sites, and can describe operations such as binding, phosphorylation, and catalysis in its rules for reactions. [14]

Furthermore, there is a great ecosystem of work built up around Kappa: *KaSim* [16] is a compiler for Kappa which 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. *Kami* [15] is a tool for aggregating pieces of biological knowledge and related model components. Finally, *PySB* [17] is a a Python framework for constructing biological models that can be compiled into Kappa and that facilitates the creation of macros and abstractions for Kappa models.

2.2 *Prior static analyses of Kappa*

Our work is not the first research into 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 [8]. 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 signaling networks in order to explore the set of complexes that are reachable by a Kappa ruleset [5].

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 [2].

Finally, Feret 2014 [9] 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 we do not start with a complete Kappa model to analyze. 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. However, we conclude from these prior works that Kappa is a language well-suited to rigorous and logic-based static analyses.

2.3 NLP fact extraction

Programming biological models by hand – fact by fact – takes time, and since research continues onward at a breakneck pace, it is unlikely for a model to stay up-to-date for very long after it has been constructed. This is the software obsolescence problem, as mentioned. Various projects in the *Big Mechanism* program [3] aim to facilitate the automated creation of executable biological models, focusing in particular on modeling signaling pathways in cancer biology. The goal is to create models that can generate causal explanations for biological processes and aid in the development of new research hypotheses.

Even though languages like Biological Expression Language (BEL) [1] or Biological Pathways Exchange (BioPAX) [7] allow research 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.

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 these papers. *INDRA* one such project: it scrapes facts from NLP of scientific literature, adds in facts from BEL and BioPAX databases, then exports the rules that it gathers to a model written in PySB. [13] [11] Not all of the rules that INDRA collects are *mechanistic*, however, as explained below; this gets in the way of constructing complete models. The role of our tool, entitled *Syndra* (from "synthesis" + INDRA), is to use logical inference to deduce the remaining mechanistic rules.

We collaborated with the INDRA developers in order to devise a tool that could address 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.

2.4 Mechanistic rules

One obstacle to the NLP approach 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 straightforwardly mechanistic and therefore easy to transform into an executable model simulating reactions of proteins. In contrast, the facts extracted by NLP may take many forms, not all of which are clear-cut mechanistic rules. For example, NLP may discover *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 isn't precise enough: 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; for this example, EGF only activates ERK1 indirectly, through a pathway involving several intermediate receptors and signaling 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 because it doesn't describe a chemical reaction, but it gives us key knowledge for decoding separate statements such as "Active ERK1 phosphorylates RSK".
- **ERK1 and ERK2 are in the ERK protein family.** Similarly, this is not a rule, but it helps us decode "MEK phosphorylates the ERK protein family".
- **S151D-mutated ERK1 behaves as if always phosphorylated.** This piece of knowledge can only be decoded if there are already rules about how phosphorylated ERK1 behaves.

To construct an executable biological model, we will need mechanistic rules. For example, the above examples of non-mechanistic rules and domain knowledge are 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.**

In summary, NLP output can be messy: it contains mechanistic rules, non-mechanistic rules, and domain knowledge, all of which must 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 produces.

3 Bridging the gap using logical inference

In order to deduce which mechanistic rules are implied by the available facts, we devise a tool that performs logical inference over those facts.

First, we choose a logical language that permits us to describe the interactions between proteins in a cellular signaling pathway as logical predicates. We ensure that this language has a clearly-defined semantics that tie directly to the semantics of Kappa and allow us to express and analyze constraints over the set of all possible Kappa models.

Next, we implement definitions of properties about proteins that might be extracted by NLP from papers: for example, "phosphorylates" or "is activated". Each definition is implemented as a function which produces a predicate in our logical language. Having defined these properties, we can then express complete NLP-extracted facts as predicates. We implement functionality for converting facts from INDRA into our tool's representation of predicates in order to demonstrate the feasibility of this procedure

Finally, writing a *solver* allows us to automate reason about the relationships among these predicates so that we can make deductions about facts and produce complete Kappa models as a result. We implement this solver using an industrial-strength Satisfiability Modulo Theories (SMT) solver, with custom code tailored to the task of translating our logical language's predicates into SMT queries.

With all of these pieces put together, we can convert a collection of NLP-extracted facts into a set of mechanistic rules consistent with those facts, automating the construction of rule-based computational models.

4 Architecture

4.1 Predicates

While constructing a model, every new fact that we add should add a little more nuance. The more pieces of information we have, the fewer candidate models there are: more and more models get eliminated for being inconsistent with the known facts. Individual biological facts therefore *constrain* the space of possible models, and *predicates* in a logic are a natural way to represent those constraints.

By using a formalized logic language for describing the space of possible models, with a clearly-defined semantics for each of the structures which that logic language can express, we can thus go about the task of rigorously building models from natural language constraints. Here, we describe such a language.

First, some review of Kappa. Each Kappa model is a set of *rules*: these are the reactions that may take place. In Kappa, a rule is executed as a *graph rewrite*: for

example, two previously-unbound agents in the system's state may become bound, adding an edge between them. The "graph" in this case is anything that could be the state of a biological model in the middle of its execution.

We want the *interpretation* of a Syndra predicate to be a set of Kappa models. In order for predicates to be interpreted, we need a data structure we can use to represent putative Kappa models.

Data structures.

As a data structure for Kappa's graph rewrite rules, we will use a tuple of *pregraph* and *postgraph*. The pregraph is a representation of the left hand side of the Kappa rule; the postgraph is a representation of the right hand side of the Kappa rule. The data structure for a graph, in turn, must contain a bunch of information: a way to represent agents that are involved in the graph, a way to represent *linking* (or binding) and *site* (or parent-child) interactions among agents, and a way to label agents as having state modifiers such as "phosphorylated" or "active", any sort of variable carrying information that might helpfully inform the constraints of our logical solving procedure.

type *Node*	:=	an enum
type *Edge*	:=	a datatype with attributes [node1: *Node*, node2: *Node*]
type *Label*	:=	an enum
type *Graph*	:=	a datatype with attributes [has: Set⟨*Node*⟩, links: Set⟨*Edge*⟩, parents: Set⟨*Edge*⟩, labelmap: Mapping⟨*Node*, Set⟨*Label*⟩⟩]
type *Rule*	:=	a datatype with attributes [pregraph: *Graph*, postgraph: *Graph*]
type *Model*	:=	Set⟨*Rule*⟩

Fig. 2. Data structures for Syndra's representation of Kappa.

With these data structures in place, predicates over models can be represented as combinations of predicates over Kappa rules, which in turn are simply predicates over the pregraph and postgraph. As a consequence of this, our logic language is divided into three different tiers of predicate: one that constrains over models, one that constrains over rules, and one that constrains over graphs.

Semantics.

In the below figures, which describe the semantics and interpretations of these predicates, let $\mathscr{M}$ represent the set of all possible models, $\mathscr{R}$ represent the set of all possible rules, and $\mathscr{G}$ represent the set of all possible graphs. ϕ and ψ always represent predicates. Let the subscripts m, r, or g, when attached to ϕ or ψ, disambiguate whether that predicate is a model predicate, a rule predicate, or a graph predicate.

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.

$$
\begin{aligned}
\neg\phi_m &:= \{M \in \mathscr{M} \mid \neg\phi_m(M)\} \\
\phi_m \vee \psi_m &:= \{M \in \mathscr{M} \mid \phi_m(M) \vee \psi_m(M)\} \\
\phi_m \wedge \psi_m &:= \{M \in \mathscr{M} \mid \phi_m(M) \wedge \psi_m(M)\} \\
\phi_m \Rightarrow \psi_m &:= \{M \in \mathscr{M} \mid \phi_m(M) \Rightarrow \psi_m(M)\} \\
\text{ForAllRules}(\phi_r) &:= \{M \in \mathscr{M} \mid \forall R \in \mathscr{R}.\ R \in M \wedge \phi_r(R)\} \\
\text{ModelHasRule}(\phi_r) &:= \{M \in \mathscr{M} \mid \exists R \in \mathscr{R}.\ R \in M \wedge \phi_r(R)\} \\
\top_m &:= \{M \in \mathscr{M}\} \\
\bot_m &:= \varnothing
\end{aligned}
$$

Fig. 3. Semantics and interpretation of Syndra's *model predicates*. The interpretation of a model predicate consists of a set of models that satisfy that predicate.

$$
\begin{aligned}
\neg\phi_r &:= \{R \in \mathscr{R} \mid \neg\phi_r(R)\} \\
\phi_r \vee \psi_r &:= \{R \in \mathscr{R} \mid \phi_r(R) \vee \psi_r(R)\} \\
\phi_r \wedge \psi_r &:= \{R \in \mathscr{R} \mid \phi_r(R) \wedge \psi_r(R)\} \\
\phi_r \Rightarrow \psi_r &:= \{R \in \mathscr{R} \mid \phi_r(R) \Rightarrow \psi_r(R)\} \\
\text{PregraphHas}(R, \phi_g) &:= \{R \in \mathscr{R} \mid \phi_g(R.\mathsf{pregraph})\} \\
\text{PostgraphHas}(R, \phi_g) &:= \{R \in \mathscr{R} \mid \phi_g(R.\mathsf{postgraph})\} \\
\top_r &:= \{R \in \mathscr{R}\} \\
\bot_r &:= \varnothing
\end{aligned}
$$

Fig. 4. Semantics and interpretation of Syndra's *rule predicates*. Similarly, the interpretation of a rule predicate consists of a set of models that satisfy that predicate.

$$
\begin{aligned}
\text{Agent}(\mathsf{node}\text{: } \textit{Node}) &:= \{G \in \mathscr{G} \mid \mathsf{node} \in G.\mathsf{has}\} \\
\text{Bound}(\phi_g, \psi_g) &:= \{G \in \mathscr{G} \mid (\textit{Edge}(\mathsf{node1} = \phi_g.\mathsf{node},\ \mathsf{node2} = \psi_g.\mathsf{node}) \\
&\qquad \in G.\mathsf{links}) \wedge\ \phi_g(G) \wedge \psi_g(G)\} \\
\text{WithSite}(\phi_g, \psi_g) &:= \{G \in \mathscr{G} \mid (\textit{Edge}(\mathsf{node1} = \phi_g.\mathsf{node},\ \mathsf{node2} = \psi_g.\mathsf{node}) \\
&\qquad \in G.\mathsf{parents}) \wedge\ \phi_g(G) \wedge \psi_g(G)\} \\
\text{Labeled}(\phi_g, \mathsf{label}\text{: } \textit{Label}) &:= \{G \in \mathscr{G} \mid \phi_g(G) \wedge (\mathsf{label} \in G.\mathsf{labelmap}[\phi_g.\mathsf{node}])\} \\
\top_g &:= \{G \in \mathscr{G}\} \\
\bot_g &:= \varnothing
\end{aligned}
$$

Fig. 5. Semantics and interpretation of Syndra's *graph predicates*. Graph predicates are special: the interpretation of a graph predicate consists of two parts. One part is a set of graphs that satisfy that predicate, and one part is a "central node" which we use in order to enforce that a graph has the structure that we're building up. Let $\phi_g(G)$ be true if and only if a graph is in the set of graphs that is the interpretation of the predicate ϕ_g, and let $\phi_g.\mathsf{node}$ represent the central node.

Having chosen this formalism to work with, we then architect a system to implement the analysis of Syndra predicates.

5 Implementation

In order to most easily interface with INDRA and with PySB models, the Syndra predicate solver is written in Python and produces Python objects as output.

Satisfiability solving and logical deductions in the Syndra predicate solver is powered by the Z3 theorem prover. Z3 is an SMT solver developed by Microsoft Research [6], and can easily interface with Python via its Python bindings [18]. Z3 is well-suited for this problem because it permits making assertions about relationships among collections of variables, then checking whether those assertions are satisfiable or not. Z3 is powerful enough to tackle problems described by first-order logic.

We define Z3 datatypes such as `Node`, `Edge`, and `Graph` to represent portions of Kappa rules, then create a Python type `Predicate` with various subclasses for each component in the logic language; instances of `Predicate` each provide an interface to Z3 to make assertions over those datatypes, effectively compiling Syndra's first-order logic to Z3's first-order logic. This provides us with sufficient power to check the satisfiability of arbitrary Syndra predicates.

In the Syndra code, Rule predicates and model predicates are referred to as `Predicate`, with subclasses for each component of the rule predicate language and model predicate language. Graph predicates are referred to as `Structure`, with subclasses for each component of the graph predicate language.

Putting it all together.

Here is some sample code that instantiates a Syndra predicate as the variable `predicate`. This predicate asserts that the model has some rule in which a molecule named "enzyme" can bind to some molecule labeled "substrate" at a specific site of the enzyme:

```
import structure
import predicate
enzyme = structure.Agent("enzyme")
enzyme_site = structure.Agent("enzyme_site")
substrate = structure.Agent("substrate")
predicate = predicate.ModelHasRule(lambda r: predicate.And(
        predicate.PregraphHas(r, enzyme.with_site(enzyme_site)),
        predicate.PregraphHas(r, substrate),
        predicate.PostgraphHas(r, enzyme.with_site(
                                    enzyme_site.bound(substrate)))))
```

5.1 Predicate features

A `Predicate` object, once instantiated, permits the following operations:

- Check a predicate's satisfiability;

- Check whether some predicate X implies some other predicate Y;
- Output a rule-based model satisfying a predicate (if satisfiable).

Below I'll describe these operations, how to interface with them, and how they contribute to the overall goal of being able to combine collections of biological facts together into a sound executable model. These operations allow us to deduce inconsistencies, detect redundancies, and eliminate ambiguity during model construction.

The code for our tool can be found at https://github.com/csvoss/syndra.

Deducing inconsistencies

Syndra can determine whether including two biological facts in a model together would be unsound. For example, it is not possible to combine the facts x = "MEK phosphorylates ERK" and $\neg x$ = "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(x)
>>> x_and_y = predicate.And(x, y)
>>> print x_and_y.check_sat()
False
```

By checking the satisfiability of multiple predicates, we *deduce inconsistencies*: this allows us to flag to model-builders that something might be wrong with the NLP-extracted facts being used to construct the model.

Detecting redundancies

Syndra can also detect whether two or more biological facts x and y imply a third biological fact z. We create a predicate for each biological fact, then check whether $x \wedge y \Rightarrow z$ is valid – that is, check whether $\neg(x \wedge y \Rightarrow z)$ is not satisfiable. The following sample code uses Syndra to perform this operation for the example of checking whether x = "MEK phosphorylates ERK" and y = "phosphorylated ERK is active" together imply z = "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_imply_z = predicate.Implies(predicate.And(x, y), z)
>>> print x_and_y_imply_z.check_sat()
True
>>> print predicate.Not(x_and_y_imply_z).check_sat()
False
```

By deducing implications, we can detect redundancies: if a new fact is always implied by the existing facts, that means it is redundant.

Eliminating ambiguity

Finally, Syndra can extract a model satisfying a predicate or set of predicates. The following sample code illustrates:

```
>>> s = solver.MySolver()
>>> s.add(predicate)
>>> s.model()
```

It is also possible to extract more than one model satisfying a predicate: add a new constraint requiring that the model not be the one just extracted, and extract again.

By being able to extract Kappa models satisfying predicates, we solve two problems: not only is it useful to have the model itself, but if *multiple* models can satisfy the same set of predicates, we can conclude that there is still *ambiguity* which could be eliminated.

6 Applications and integrations

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. We've shown how we chose and implemented a graph logic language for describing predicates over biological models; here, we'll discuss how we can use this tool with real output produced by INDRA in order to produce biological models.

6.1 Macros

We have remarked that Syndra permits the direct construction of predicates; Syndra also permits the construction of predicates automatically from INDRA facts.

In order to make it easier to create predicates based on the higher-level English statements that natural language processing yields, Syndra defines a collection of *macros*. These are functions which output predicates for some common biological facts, parametrized on a few variable inputs. For example, `directly_activates(A, B)` outputs a predicate specifying that A activates B, and `phosphorylated_is_active(A)` outputs a predicate specifying that phosphorylated A must be active.

In order to define each of these macros, we choose a way to express the statement as a Syndra predicate. For example, the implementation of the phosphorylation macro `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:

```
directly_phosphorylates(A, B) =
predicate.Exists(predicate.Implies(predicate.Named(A, name_a),
        predicate.Implies(predicate.Named(B, name_b),
            predicate.And(predicate.PreLabeled(A, ACTIVE),
                predicate.PreUnlabeled(B, PHOSPHORYLATED),
                predicate.PostLabeled(A, ACTIVE),
```

```
        predicate.PostLabeled(B, PHOSPHORYLATED)))))
```

Different labs using computational tools often have different preferences for how to model certain systems and the assumptions that they make. If macros are created using an underlying logic language, then labs are able to customize according to their own preferred definitions and assumptions, encode those definitions as predicates using macros, and have the resulting logical deductions still be sound by Syndra's semantics.

We can use these macros in order to convert INDRA statements into Syndra predicates: pattern-matching on the INDRA Python object and then applying the right macro appropriately. This is in `statements_to_predicates.py`.

6.2 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 [13]. 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 Syndra predicates, by determining the appropriate macro to apply to each INDRA statement. From there, we can check implications between Syndra 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)
```

$$\Downarrow$$

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

The code supporting the conversion of individual INDRA statements into Syndra 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 be expanded to accommodate them.

7 Conclusion

Syndra is a tool for translating biological facts into predicates in order to find complete Kappa models satisfying sets of biological facts. We implemented predicates and 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 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 could see more work improving Syndra towards the goal of smoothly converting facts from systems like INDRA all the way to Kappa or PySB models. Currently, the models that Syndra outputs are instances of Z3 datatypes, representing satisfying values of the datastructures - enough information to describe a Kappa model - rather than Kappa models themselves, directly. This Z3 output contains enough information to be translated into a Kappa model; one important next step would be to implement that translation.

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. Syndra could be extended in order to encode a greater variety of pieces of domain knowledge into its predicates. 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. It is likely that in the future there may be even more advanced computational systems for reasoning about biological problems. Executable biological models are potentially a useful resource to this future: not only can they be executed, but they are also *machine-readable*, and thus easier for new computational tools to integrate with than human research results alone.

With our work implementing this procedure, 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, if it becomes possible to construct and maintain rule-based biological models automatically.

8 Acknowledgements

Much inspiration for the logic language that Syndra uses came from conversations with and reading the unpublished work of Adrien Husson and Jean Krivine, who devised a first-order logic that functions as a "meta-Kappa," formalizing predicates over the space of possible rule-based Kappa models. [12]

Thanks to Peter Sorger, John Bachman, and Benjamin Gyori, for many conversations about INDRA and its functionality, and for the Sorger lab's generous spon-

sorship of some of this work in spring of 2016.

References

[1] Biological Expression Language (BEL). http://openbel.org/. Accessed: 2017-07-15.

[2] F. Camporesi, J. Feret, and J. Hayman. Context-Sensitive Flow Analyses: A Hierarchy of Model Reductions. In A. Gupta and T. Henzinger, editors, *Eleventh Conference on Computational Method in Systems Biology (CMSB'13)*, number 8130 in LNBI, pages 220–233. Springer, 2013.

[3] P. R. Cohen. DARPA's Big Mechanism program. *Phys Biol*, 12(4):045008, Jul 2015.

[4] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-Based Modelling of Cellular Signalling. L. Caires and V.T. Vasconcelos (Eds.): CONCUR 2007, LNCS 4703, pp. 17-41, 2007. doi:10.1007/978-3-540-74407-8_3.

[5] V. Danos, J. Feret, W. Fontana, and J. Krivine. Abstract Interpretation of Cellular Signalling Networks. In F. Logozzo, D. A. Peled, and L. D. Zuck, editors, *Proceedings of the Ninth International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'2008*, volume 4905 of *Lecture Notes in Computer Science*, pages 83–97, San Francisco, USA, 7–9 January 2008. Springer, Berlin, Germany.

[6] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] E. Demir, M. P. Cary, S. Paley, K. Fukuda, C. Lemer, I. Vastrik, G. Wu, P. D'Eustachio, C. Schaefer, J. Luciano, F. Schacherer, I. Martinez-Flores, Z. Hu, V. Jimenez-Jacinto, G. Joshi-Tope, K. Kandasamy, A. C. Lopez-Fuentes, H. Mi, E. Pichler, I. Rodchenkov, A. Splendiani, S. Tkachev, J. Zucker, G. Gopinath, H. Rajasimha, R. Ramakrishnan, I. Shah, M. Syed, N. Anwar, O. Babur, M. Blinov, E. Brauner, D. Corwin, S. Donaldson, F. Gibbons, R. Goldberg, P. Hornbeck, A. Luna, P. Murray-Rust, E. Neumann, O. Ruebenacker, O. Reubenacker, M. Samwald, M. van Iersel, S. Wimalaratne, K. Allen, B. Braun, M. Whirl-Carrillo, K. H. Cheung, K. Dahlquist, A. Finney, M. Gillespie, E. Glass, L. Gong, R. Haw, M. Honig, O. Hubaut, D. Kane, S. Krupa, M. Kutmon, J. Leonard, D. Marks, D. Merberg, V. Petri, A. Pico, D. Ravenscroft, L. Ren, N. Shah, M. Sunshine, R. Tang, R. Whaley, S. Letovksy, K. H. Buetow, A. Rzhetsky, V. Schachter, B. S. Sobral, U. Dogrusoz, S. McWeeney, M. Aladjem, E. Birney, J. Collado-Vides, S. Goto, M. Hucka, N. Le Novere, N. Maltsev, A. Pandey, P. Thomas, E. Wingender, P. D. Karp, C. Sander, and G. D. Bader. The BioPAX community standard for pathway data sharing. *Nat. Biotechnol.*, 28(9):935–942, Sep 2010. http://www.biopax.org/.

[8] J. Feret. Reachability Analysis of Biological Signalling Pathways by Abstract Interpretation. In T. Simos, editor, *Proceedings of the International Conference of Computational Methods in Sciences and Engineering, ICCMSE '2007, Corfu, Greece*, number 963.(2) in American Institute of Physics Conference Proceedings, pages 619–622, Corfu, Greece, 25–30 September 2007. American Institute of Physics.

[9] J. Feret. An Algebraic Approach for Inferring and Using Symmetries in Rule-based Models. *Electronic Notes in Theoretical Computer Science*, 316:45 – 65, 2015. 5th International Workshop on Static Analysis and Systems Biology (SASB 2014).

[10] J. Fisher and T. A. Henzinger. Executable cell biology. Nature Biotechnology 25, pp. 1239-1249, 2007. doi:10.1038/nbt1356.

[11] B. M. Gyori, J. A. Bachman, K. Subramanian, J. L. Muhlich, L. Galescu, and P. K. Sorger. From word models to executable models of signaling networks using automated assembly. *bioRxiv*, 2017. http://www.biorxiv.org/content/early/2017/03/24/119834.

[12] A. Husson and J. Krivine. Rule-based biological modeling in a logical setting, 2015. Unpublished work.

[13] Integrated Network and Dynamical Reasoning Assembler (INDRA). https://github.com/sorgerlab/indra. Accessed: 2016-05-18.

[14] An Introduction to Kappa Syntax. http://www.kappalanguage.org/syntax.html. Accessed: 2016-04-27.

[15] Kami: Annoted knowledge representation translation into Kappa (Knowledge Aggregator and Model Instantiator). https://github.com/Kappa-Dev/Kami. Accessed: 2016-04-27.

[16] KaSim: Command line stochastic simulator for Kappa models. https://github.com/Kappa-Dev/KaSim. Accessed: 2016-04-27.

[17] C. F. Lopez, J. L. Muhlich, J. A. Bachman, and P. K. Sorger. Programming biological models in Python using PySB. http://msb.embopress.org/content/9/1/646, 2013. doi:10.1038/msb.2013.1.

[18] The Z3 Theorem Prover. https://github.com/Z3Prover/z3. Accessed: 2016-05-18.