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

Chelsea Voss, Jean Yang, Walter Fontana
Static Analysis in Systems Biology, 2017
The need for biological models

 executable models for “in silico” experimentation
programming is hard
The need for computer-generated models
Some NLP output requires logical inference
Some NLP output requires logical inference

Executable model needs:
  Mechanistic rules
Some NLP output requires logical inference

NLP produces:  

- Mechanistic rules
- Non-mechanistic rules
- Domain knowledge

Executable model needs:  

- Mechanistic rules
Some NLP output requires logical inference

**NLP produces:**

* Mechanistic rules
  - Non-mechanistic rules
    - *Domain knowledge*

**Executable model needs:**

* Mechanistic rules

???
Some NLP output requires logical inference

NLP produces:

Mechanistic rules
- MEK phosphorylates ERK1

Non-mechanistic rules
- 
- 

Domain knowledge
- 
- 

Executable model needs:

Mechanistic rules

???
Some NLP output requires logical inference

NLP produces:

*Mechanistic rules*
- MEK phosphorylates ERK1

*Non-mechanistic rules*
- MEK phosphorylates the ERK protein family
- Active ERK phosphorylates RSK

*Domain knowledge*

Executable model needs:

*Mechanistic rules*
Some NLP output requires logical inference

NLP produces:

*Mechanistic rules*
- MEK phosphorylates ERK1

*Non-mechanistic rules*
- MEK phosphorylates the ERK protein family
- Active ERK phosphorylates RSK

*Domain knowledge*
- When ERK1 is phosphorylated, it is active
- S151D-mutated ERK1 behaves as if always phosphorylated
- ERK1 and ERK2 are in the ERK protein family

Executable model needs:

*Mechanistic rules*
Some NLP output requires logical inference

**NLP produces:**

*Mechanistic rules*
- MEK phosphorylates ERK1

*Non-mechanistic rules*
- MEK phosphorylates the ERK protein family
- Active ERK phosphorylates RSK

*Domain knowledge*
- When ERK1 is phosphorylated, it is active
- S151D-mutated ERK1 behaves as if always phosphorylated
- ERK1 and ERK2 are in the ERK protein family

**Executable model needs:**

*Mechanistic rules*
- MEK phosphorylates ERK1
- MEK phosphorylates ERK2
- Phosphorylated ERK1 phosphorylates RSK
- Phosphorylated ERK2 phosphorylates RSK
- S151D-mutated ERK1 phosphorylates RSK
Mechanistic rules
Non-mechanistic rules
Domain knowledge
Mechanistic rules
Non-mechanistic rules
Domain knowledge

Models
Mechanistic rules
Non-mechanistic rules
Domain knowledge

Space of possible models
Our contribution

Mechanistic rules
Non-mechanistic rules
Domain knowledge

Space of possible models
Our contribution: how it works

Mechanistic rules
Non-mechanistic rules
Domain knowledge

1: Predicates over models

Space of possible models
Our contribution: how it works

Mechanistic rules
Non-mechanistic rules
Domain knowledge

1: Predicates over models

2: Implement interpretation

Space of possible models
Our contribution: how it works

1: Predicates over models
2: Implement interpretation
3: Create predicates

Mechanistic rules
Non-mechanistic rules
Domain knowledge

Space of possible models
1: Predicates over models

2: Implement interpretation

3: Create predicates

Mechanistic rules
Non-mechanistic rules
Domain knowledge

Space of possible models
1: Predicates over models, in a logic

First, **choose a modeling language**.
1: Predicates over models, in a logic

First, choose a modeling language: **Kappa.**
1: Predicates over models, in a logic

First, choose a modeling language: Kappa.

Kappa rules

\[
\begin{align*}
\text{+ } & \rightarrow \text{ } \at \text{ } (0.2) \\
\text{+ } & \rightarrow \text{ } @ \text{ } (0.8)
\end{align*}
\]

Simulation of resulting system
1: Predicates over models, in a logic

First, choose a modeling language: Kappa.

Why Kappa?

Kappa rules

Simulation of resulting system
1: Predicates over models, in a logic

First, choose a modeling language: Kappa.

Why Kappa?

Well-defined operational semantics allow us to reason precisely.

[Figure due to Danos et al. 2009: Abstracting the ODE Semantics of Rule-Based Models: Exact and Automatic Model Reduction.]
1: Predicates over models, in a logic

First, choose a modeling language: Kappa.
Second, devise a logic for quantifying over models.
1: Predicates over models, in a logic

First, choose a modeling language: Kappa.

Second, **devise a logic for quantifying over models.**

Datatypes:

- **Graphs** represent the state of a Kappa system
- **Rules** are sets of \(<\text{graph}, \text{action}>\) pairs
  - action rewrites graph, creates new graph
- **Models** are sets of rules

[Conversations with Husson & Krivine, 2015-2016]
1: Predicates over models, in a logic

First, choose a modeling language: Kappa.
Second, devise a logic for quantifying over models.

Datatypes:
• **Graphs** represent the state of a Kappa system
• **Rules** are sets of \(\langle\text{graph, action}\rangle\) pairs
  • action rewrites graph, creates new graph
• **Models** are sets of rules

Predicates:
• **Atomic predicates** specify a set of rules
• **Predicates** specify a set of models

[Conversations with Husson & Krivine, 2015-2016]
Atomic predicates

class AtomicPredicate:

    Top
    Bottom
    Equal
    PreLabeled, PostLabeled
    PreUnlabeled, PostUnlabeled
    PreParent, PostParent
    PreLink, PostLink
    PreHas, PostHas
    Add, Rem
    DoLink, DoUnlink
    DoParent, DoUnparent
    Named
**Atomic predicates**

class AtomicPredicate:

- Top
- Bottom
- Equal
- PreLabeled, PostLabeled
- PreUnlabeled, PostUnlabeled
- PreParent, PostParent
- PreLink, PostLink
- PreHas, PostHas
- Add, Rem
- DoLink, DoUnlink
- DoParent, DoUnparent
- Named

**Predicates**

class Predicate:

- And
- Not
- Or
- Implies
- ModelHasRule
- ForAllRules
- Top
- Bottom
Example predicate syntax tree

\[ a = \text{Agent('a')} \]
\[ b = \text{Agent('b')} \]
\[ p = \text{And(} \]
\[ \quad \text{ModelHasRule(lambd}a\text{. \text{r:}} \]
\[ \quad \text{PregraphHas(}r, a.\text{.bound}(b))\text{)}, \]
\[ \quad \text{ModelHasRule(lambd}a\text{. \text{r:}} \]
\[ \quad \text{PostgraphHas(}r, a.\text{.unbound}(b)))\text{)} \]
1: Predicates over models
2: Implement interpretation
3: Create predicates

Mechanistic rules
Non-mechanistic rules
Domain knowledge

Space of possible models
2: Implement interpretation of predicates

- Solving predicates in this logic is *reducible to first-order logic*
2: Implement interpretation of predicates

- Solving predicates in this logic is reducible to first-order logic
- Workhorse: Z3 Theorem Prover
2: Implement interpretation of predicates

• Solving predicates in this logic is reducible to first-order logic
• Workhorse: Z3 Theorem Prover
2: Implement interpretation of predicates

- Solving predicates in this logic is **reducible to first-order logic**
- **Workhorse: Z3 Theorem Prover**
  - Demo at [http://rise4fun.com/z3](http://rise4fun.com/z3)
2: Implement interpretation of predicates

• Solving predicates in this logic is reducible to first-order logic

• Workhorse: Z3 Theorem Prover
  • Demo at http://rise4fun.com/z3
  • High-performance satisfiability solver
2: Implement interpretation of predicates

• Solving predicates in this logic is reducible to first-order logic

• Workhorse: Z3 Theorem Prover
  • Demo at http://rise4fun.com/z3
  • High-performance satisfiability solver
  • Wide variety of datatypes supported: arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers
2: Implement interpretation of predicates

- Solving predicates in this logic is reducible to first-order logic
- Workhorse: Z3 Theorem Prover
  - Demo at [http://rise4fun.com/z3](http://rise4fun.com/z3)
  - High-performance satisfiability solver
  - Wide variety of datatypes supported: arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers

Is this formula satisfiable?

1. `(declare-fun x () Int)
2. (assert (>= 5 x))
3. (check-sat)
4. (get-model)
2: Implement interpretation of predicates

- Solving predicates in this logic is reducible to first-order logic
- Workhorse: Z3 Theorem Prover
  - Demo at [http://rise4fun.com/z3](http://rise4fun.com/z3)
  - High-performance satisfiability solver
  - Wide variety of datatypes supported: arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers

Is this formula satisfiable?

```
1 (declare-fun x () Int)
2 (assert (>= 5 x))
3 (check-sat)
4 (get-model)
```
2: Implement interpretation of predicates

- Solving predicates in this logic is reducible to first-order logic

- Workhorse: **Z3 Theorem Prover**
  - Demo at [http://rise4fun.com/z3](http://rise4fun.com/z3)
  - High-performance satisfiability solver
  - Wide variety of datatypes supported: arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers

---

Is this formula satisfiable?

1 (declare-fun x () Int)
2 (assert (>= 5 x))
3 (check-sat)
4 (get-model)

sat
2: Implement interpretation of predicates

• Solving predicates in this logic is reducible to first-order logic

• Workhorse: **Z3 Theorem Prover**
  • Demo at [http://rise4fun.com/z3](http://rise4fun.com/z3)
  • High-performance satisfiability solver
  • Wide variety of datatypes supported: arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers

Is this formula satisfiable?

<table>
<thead>
<tr>
<th>No.</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(declare-fun x () Int)</td>
</tr>
<tr>
<td>2</td>
<td>(assert (&gt;= 5 x))</td>
</tr>
<tr>
<td>3</td>
<td>(check-sat)</td>
</tr>
<tr>
<td>4</td>
<td>(get-model)</td>
</tr>
</tbody>
</table>

sat

(model

(define-fun x () Int 5)

)
2: Implement interpretation of predicates

• Solving predicates in this logic is reducible to first-order logic

• Workhorse: **Z3 Theorem Prover**

• Using Z3 to interpret our predicates
  • Declare **Z3 datatypes** to represent
  • **Recursively build** Z3 predicates from our predicate classes
  • Use (check-sat) and (get-model)
2: Implement interpretation of predicates

- Solving predicates in this logic is reducible to first-order logic
- Workhorse: Z3 Theorem Prover
- Using Z3 to interpret our predicates

- Value added:
  - Extract models
  - Detect inconsistencies (if $P$ is our facts so far and $Q$ is a new predicate, and $P \land \neg Q$ is unsatisfiable, then $Q$ is inconsistent with the existing facts)
  - Detect redundancy (if $Q$ is a new fact, and $P \implies Q$, then $Q$ is redundant)
  - Detect ambiguity (if model $M$ satisfies predicate $P$, and $P \land \neg (\text{model}=M)$ is satisfiable, then $P$ has multiple solutions)
Usage example: Inconsistency checking

```python
>>> from syndra.engine import macros, predicate
```
Usage example: Inconsistency checking

```python
>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
```
Usage example: Inconsistency checking

>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
>>> y = predicate.Not(x)
Usage example: Inconsistency checking

```python
>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
>>> y = predicate.Not(x)
>>> x_and_y = predicate.And(x, y)
```
Usage example: Inconsistency checking

```python
>>> 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()
```
Usage example: Inconsistency checking

```python
>>> 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
```
Usage example: Redundancy checking

>>> from syndra.engine import macros, predicate
Usage example: Redundancy checking

```python
>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
```
Usage example: Redundancy checking

```python
>>> from syndra.engine import macros, predicate

```
Usage example: Redundancy checking

```python
>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
>>> y = macros.phosphorylated_is_active("ERK")
>>> z = macros.directly_activates("MEK", "ERK")
```
Usage example: Redundancy checking

```python
>>> from syndra.engine import macros, predicate

```
Usage example: Redundancy checking

```python
>>> 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()
```
from syndra.engine import macros, predicate

```python
>>> 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
1: Predicates over models

2: Implement interpretation

Space of possible models

3: Create predicates

Mechanistic rules
Non-mechanistic rules
Domain knowledge
3: Tools for creating predicates

• Macros
3: Tools for creating predicates

• Macros

A phosphorylates B

\[
\text{PreLabeled}(A, \text{phosphorylated}) \land \\
\text{PreUnbound}(A, B) \land \\
\text{PostLabeled}(A, \text{phosphorylated}) \land \\
\text{PostBound}(A, B)
\]
3: Tools for creating predicates

- **Macros**

  A phosphorylates B

  \[
  \text{PreLabeled}(A, \text{phosphorylated}) \land \\
  \text{PreUnbound}(A, B) \land \\
  \text{PostLabeled}(A, \text{phosphorylated}) \land \\
  \text{PostBound}(A, B)
  \]

- **directly_phosphorylates**
- **phosphorylated_is_active**
- **directly_activates**
- **negative_residue_behaves_as_if_phosphorylated**
3: Tools for creating predicates

• Macros

• Interface with INDRA

[INDRA: Gyori et al. *From word models to executable models of signaling networks using automated assembly*. 2017]
3: Tools for creating predicates

• Macros

• Interface with INDRA
  • indra.statements.Phosphorylation
  • indra.statements.Activation
  • indra.statements.ActiveForm

[INDRA: Gyori et al. From word models to executable models of signaling networks using automated assembly. 2017]
Mechanistic rules
Non-mechanistic rules
Domain knowledge

3: Create predicates

1: Predicates over models

2: Implement interpretation

Space of possible models
https://github.com/csvoss/syndra