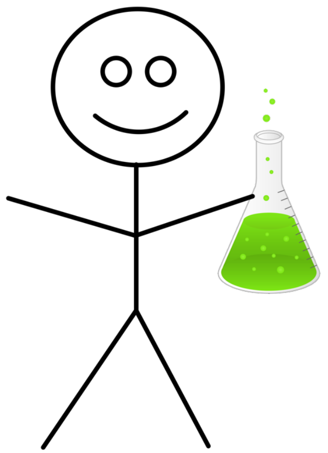
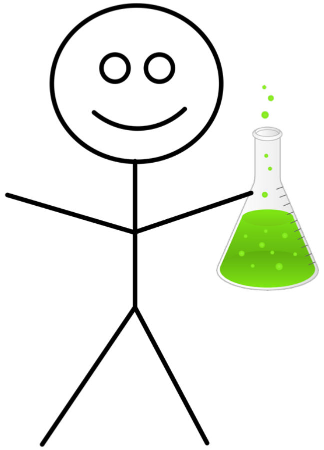
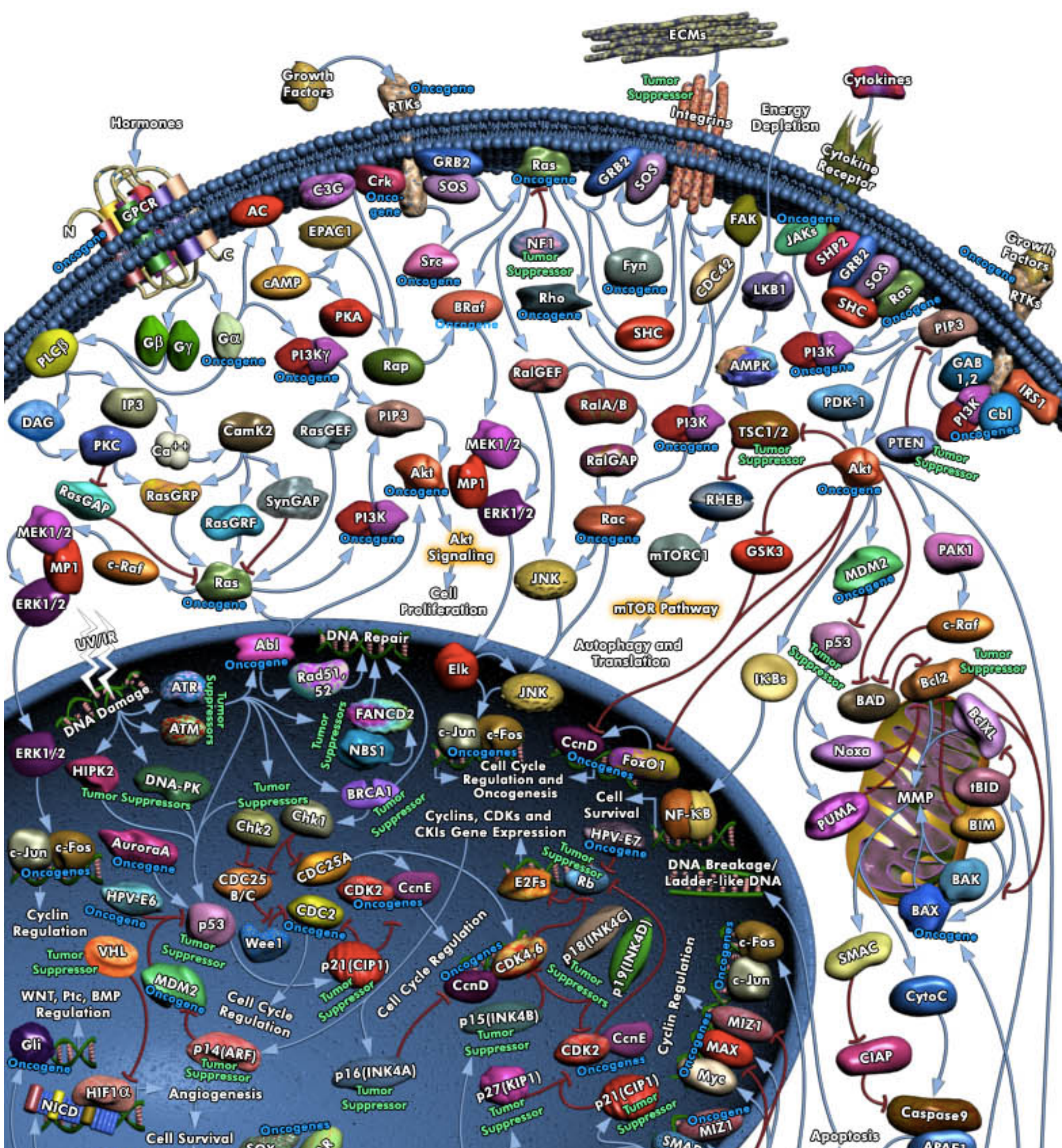
A Tool for Automated Inference of Executable Rule-Based Biological Models

Chelsea Voss, Jean Yang, Walter Fontana

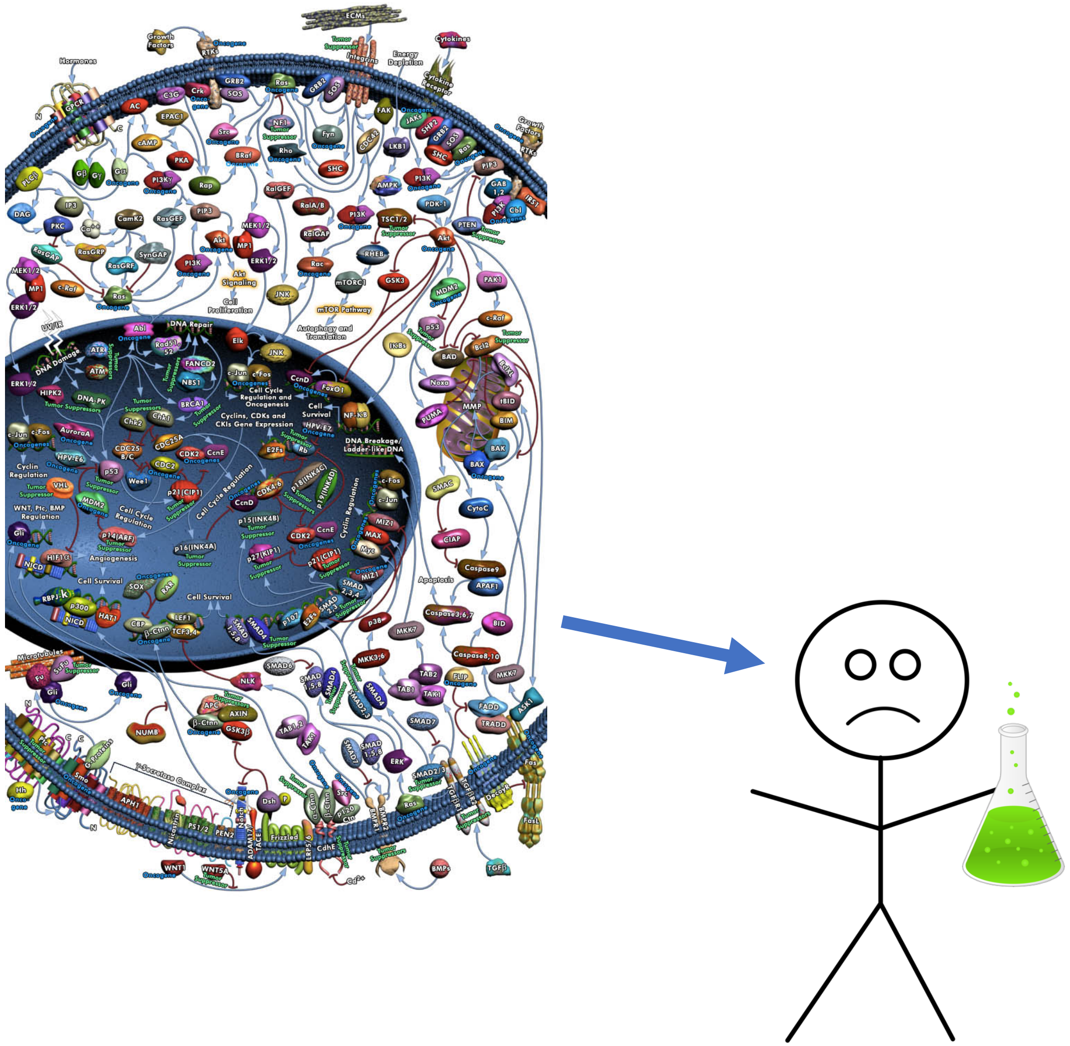
Static Analysis in Systems Biology, 2017

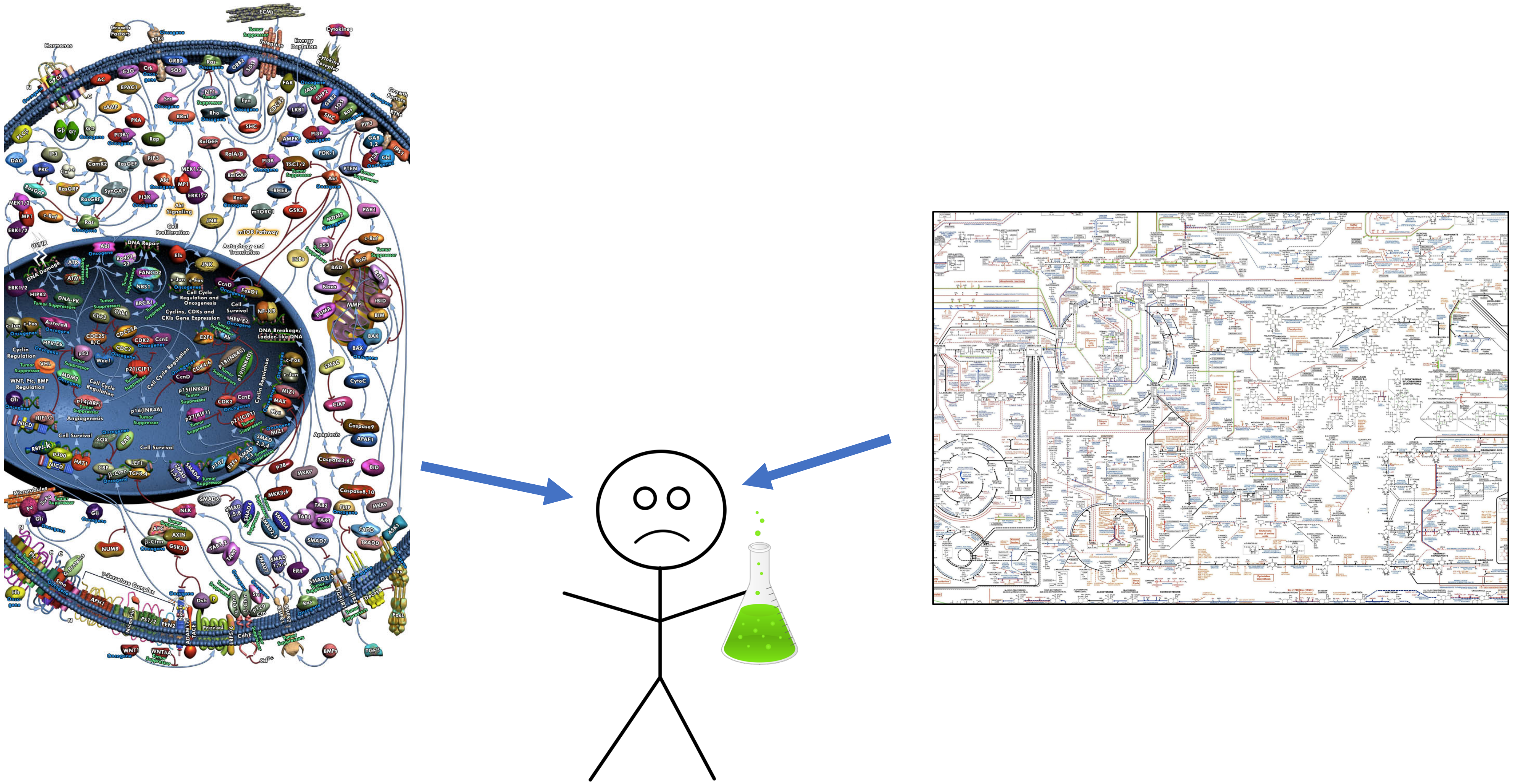


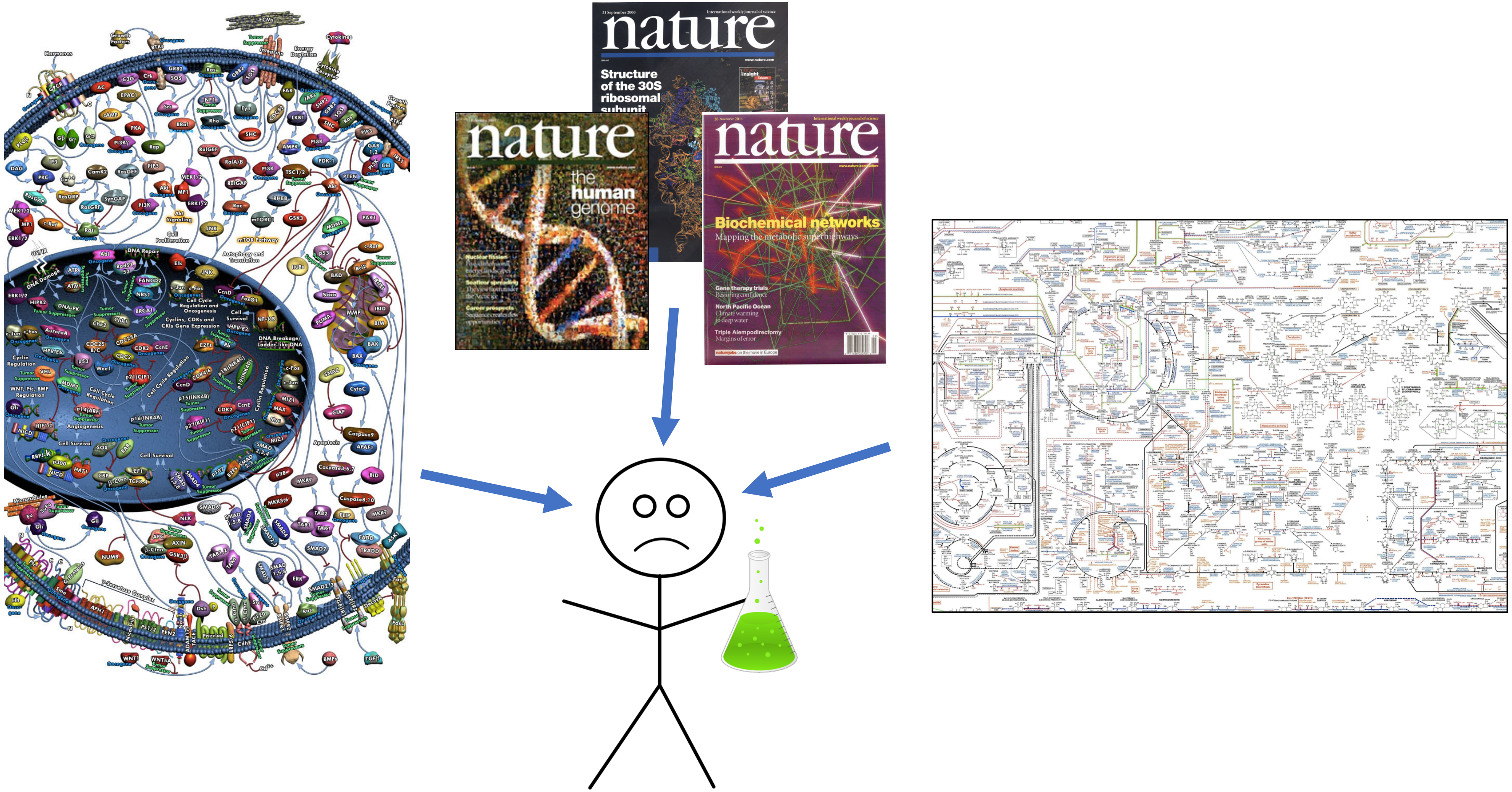




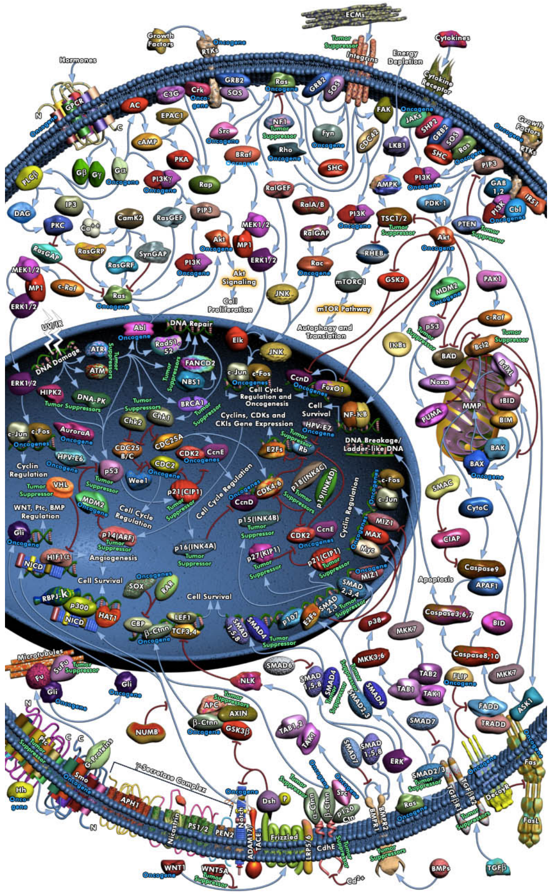
ECMs
Growth Factors
RTKs
Oncogene
Tumor Suppressor
Integrins
Cytokines
Energy Depletion
Cytokine Receptor
Hormones
GPCR
N
C
C3G
Crk
GRB2
SOS
Ras
FAK
JAKs
SHP2
SHC
AC
EPAC1
Src
NF1
Fyn
CDC42
LKB1
cAMP
PKA
BRaf
Rho
Gβ
Gγ
Gα
PLCβ
PI3Kγ
Rap
RalGEF
AMPK
PI3K
PIP3
GAB 1,2
IRS1
Cbl
DAG
IP3
CamK2
RasGEF
MEK1/2
RalA/B
TSC1/2
PDK-1
PTEN
PKC
Ca++
RasGRP
SynGAP
Akt
MP1
RalGAP
RHEB
RasGAP
RasGRF
ERK1/2
Akt Signaling
Rac
mTORC1
GSK3
MDM2
PAK1
c-Raf
Cell Proliferation
JNK
mTOR Pathway
Autophagy and Translation
UV/IR
DNA Damage
Abl
DNA Repair
Rad51, 52
ATR
ATM
Tumor Suppressors
FANCD2
NBS1
Elk
c-Jun
c-Fos
Oncogenes
CcnD
FoxO1
IKBs
p53
BAD
Bcl2
BclXL
HIPK2
DNA-PK
BRCA1
Chk1
Chk2
Cell Cycle Regulation and Oncogenesis
Cell Survival
NF-KB
Noxa
tBID
MMP
PUMA
BIM
Cyclins, CDKs and CKIs Gene Expression
HPV-E7
DNA Breakage/ Ladder-like DNA
AuroraA
HPV-E6
CDC25 B/C
CDC25A
CDK2
CcnE
E2Fs
Rb
BAK
BAX
Cyclin Regulation
VHL
CDC2
Wee1
p21(CIP1)
CDK4,6
p18(INK4C)
p19(INK4D)
Cell Cycle Regulation
Cyclin Regulation
SMAC
CytoC
WNT, Ptc, BMP Regulation
MIZ1
MAX
Myc
Gli
p14(ARF)
p15(INK4B)
p16(INK4A)
p27(KIP1)
CIAP
HIF1α
NICD
Angiogenesis
Cell Survival
Caspase9
Apoptosis

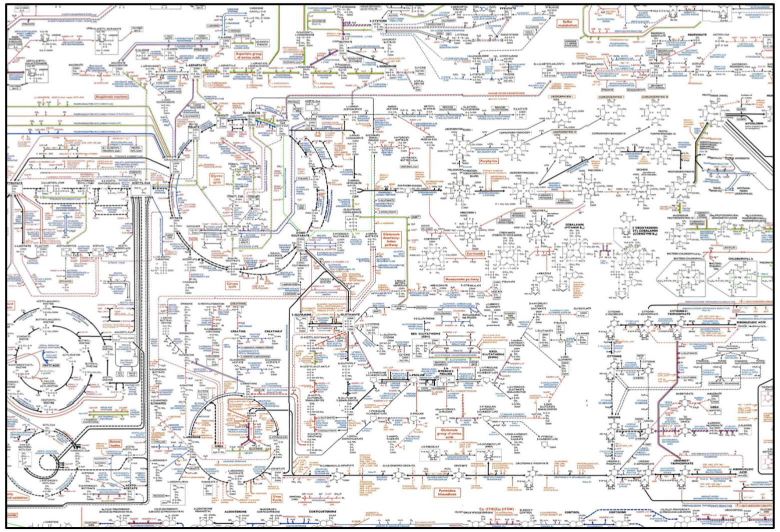


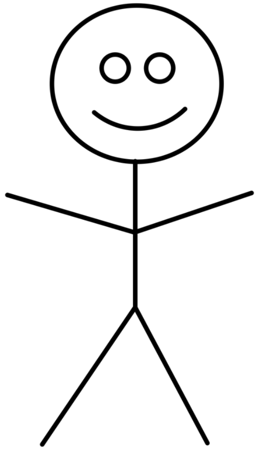



nature
Structure of the 30S ribosomal subunit
nature
the human genome
nature
Biochemical networks
Mapping the metabolic superhighways
Gene therapy trials
Restoring confidence
North Pacific Ocean
Climate warming in deep water

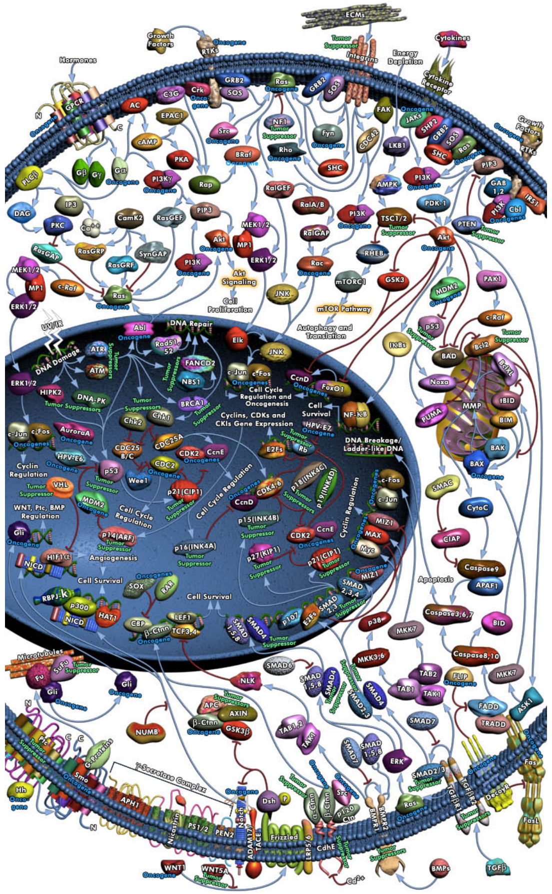
The need for biological models

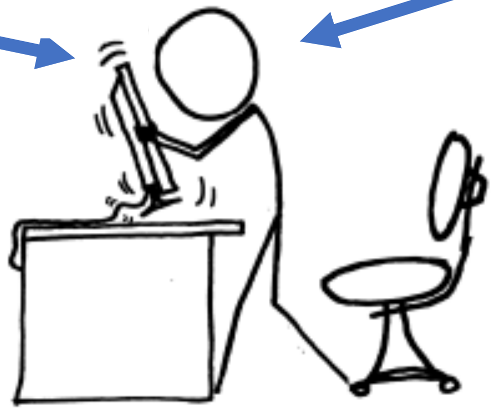




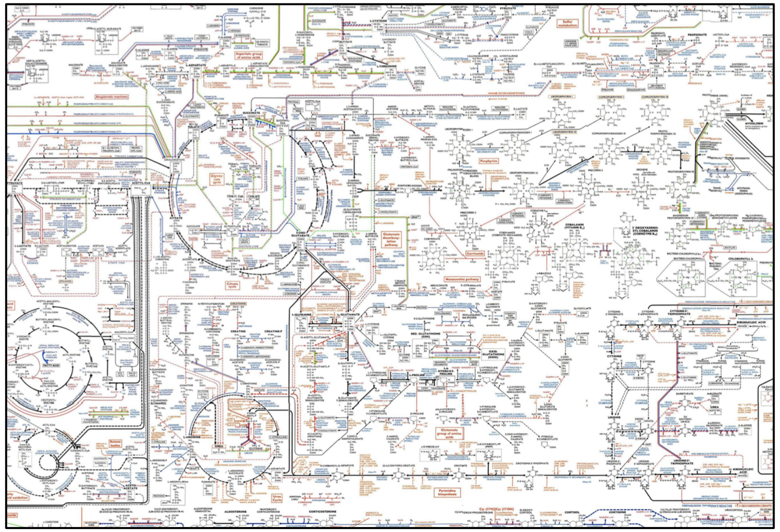


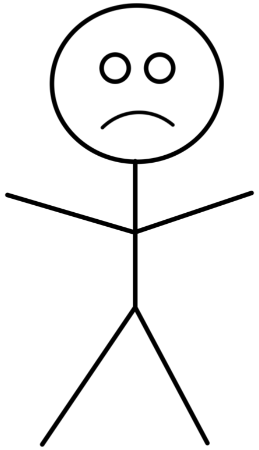
executable models for
"in silico" experimentation



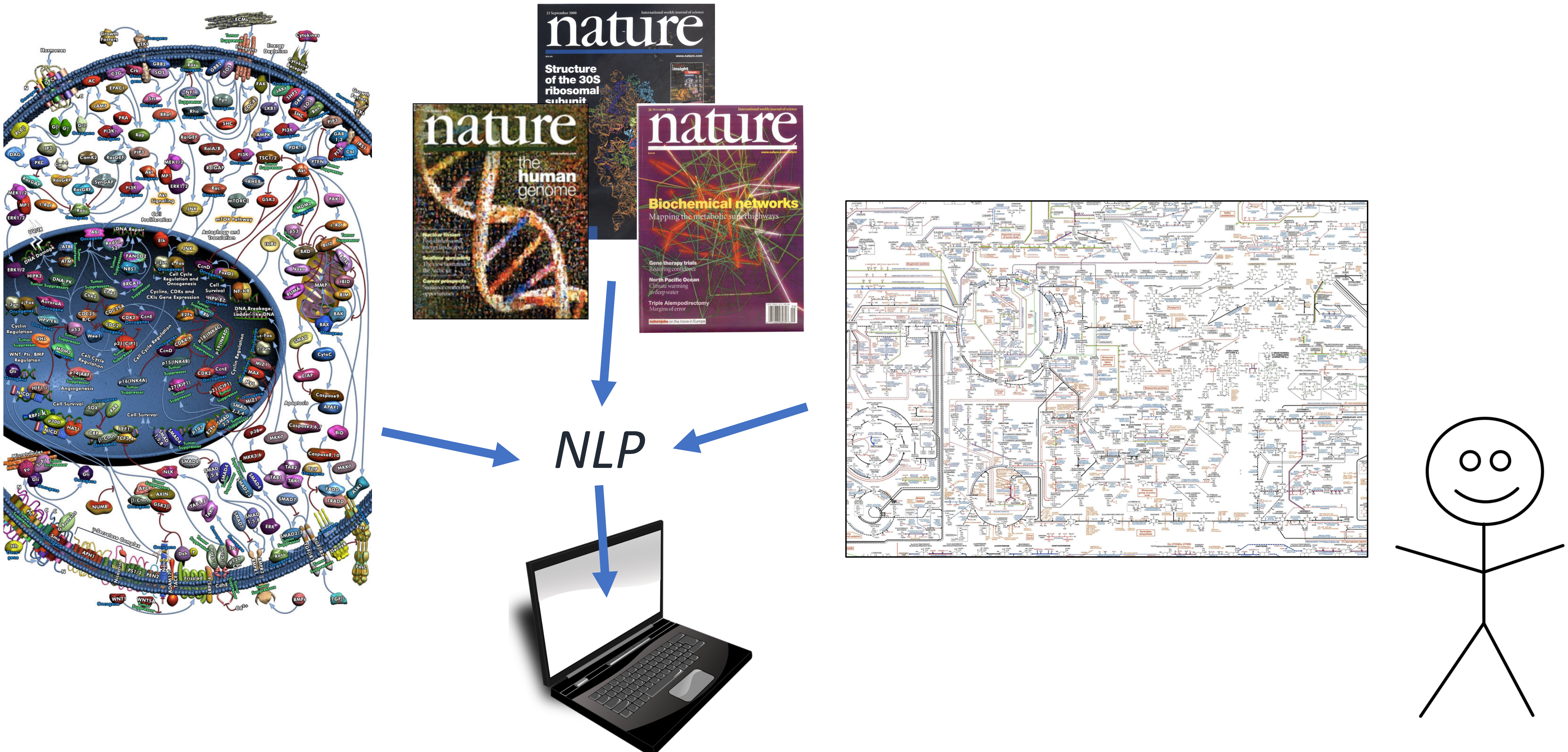


programming is hard





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

-
-
-

??? $\longrightarrow$

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

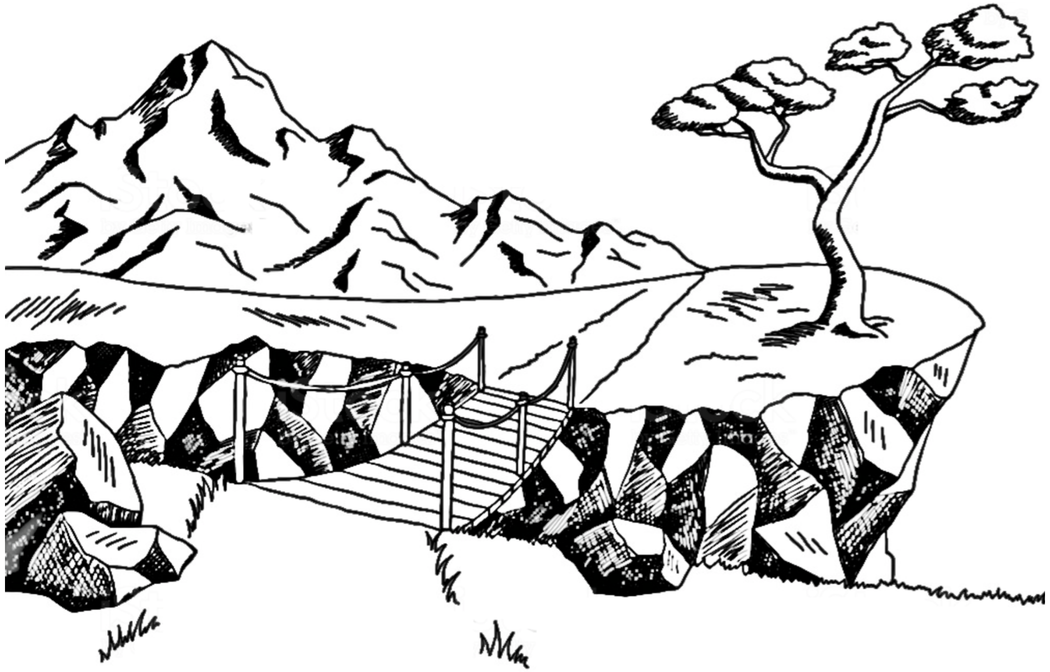
Space of possible *models*

Our contribution

Mechanistic rules
Non-mechanistic rules
Domain knowledge

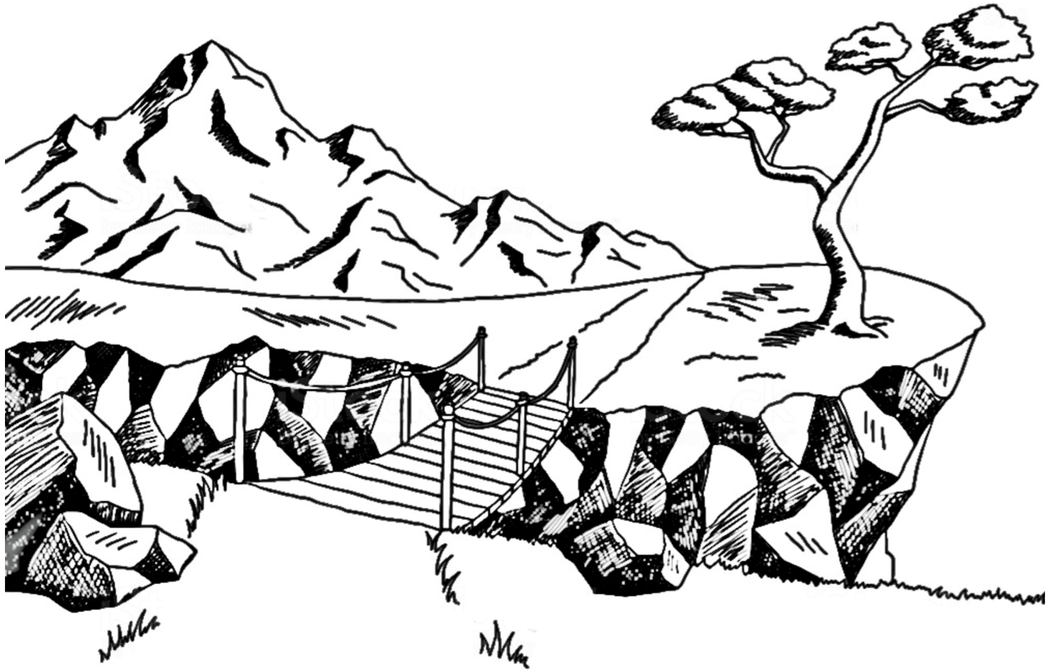
→

Space of possible models



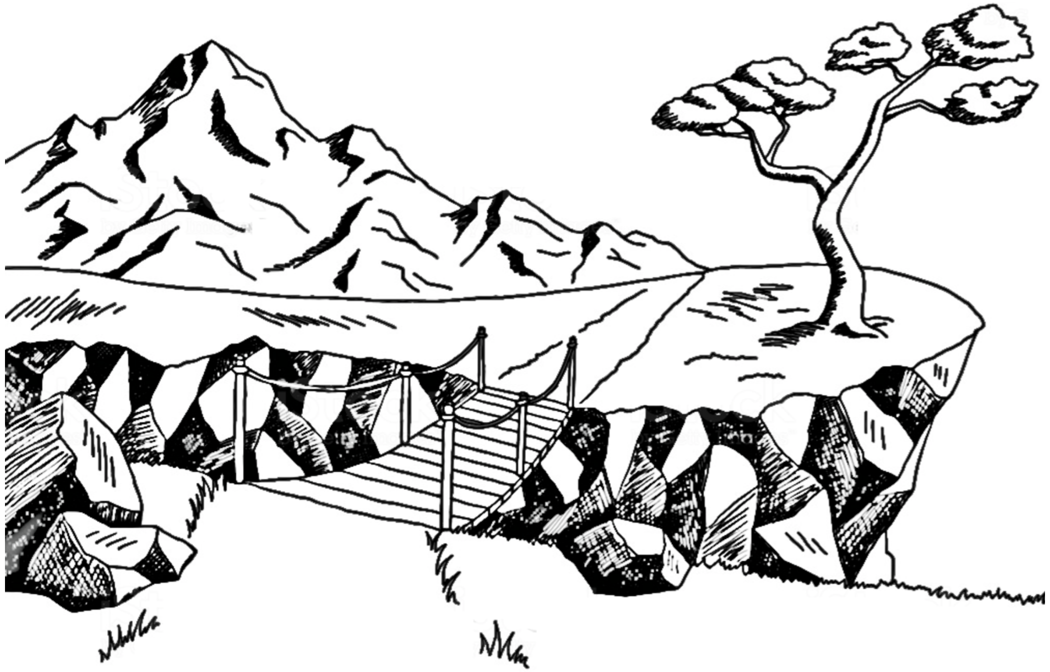
Our contribution: how it works

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

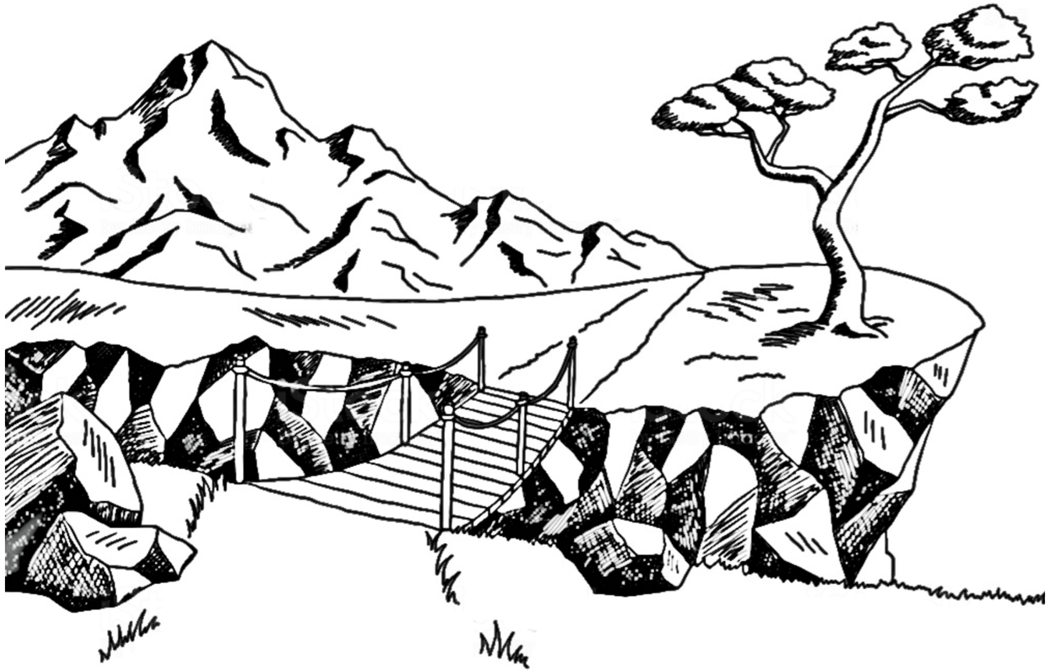
Mechanistic rules
Non-mechanistic rules
Domain knowledge

3: Create predicates →

1: Predicates over models

2: Implement interpretation →

Space of possible models



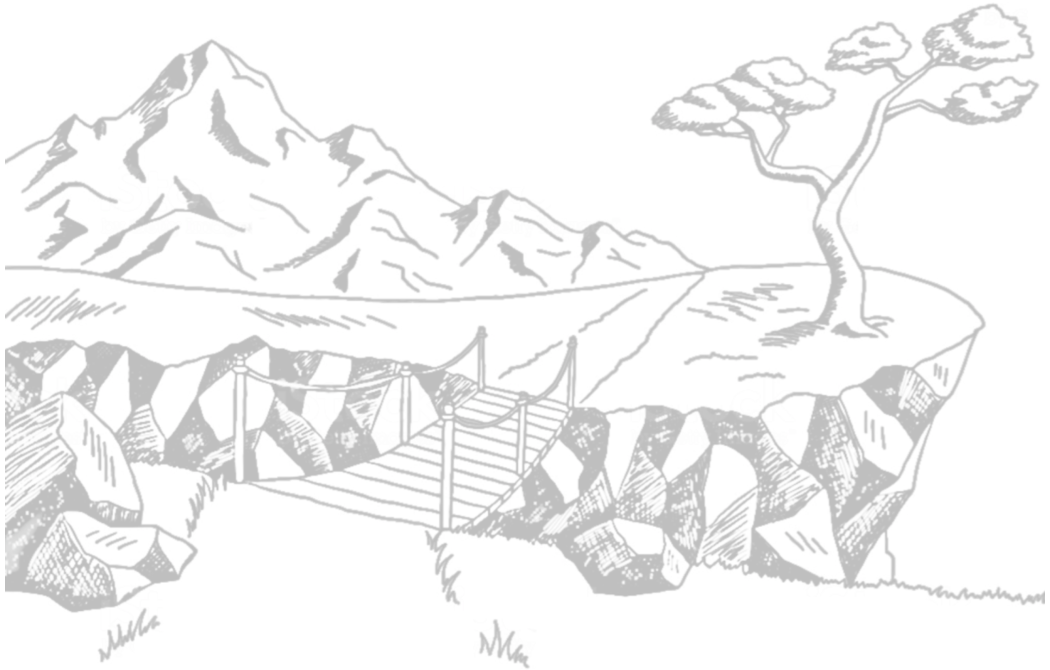
Mechanistic rules
Non-mechanistic rules
Domain knowledge

3: Create predicates →

1: Predicates over models

2: Implement interpretation →

Space of possible models



1: Predicates over models, in a logic

First, **choose a modeling language.**

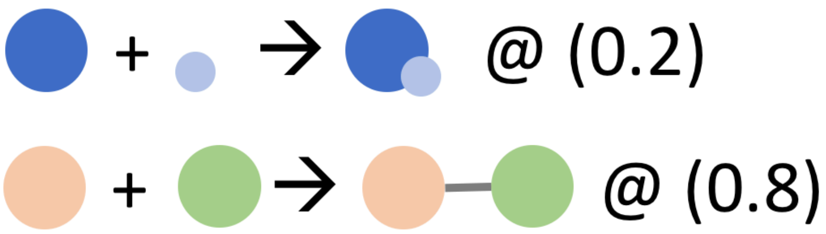
1: Predicates over models, in a logic

First, choose a modeling language: **Kappa.**

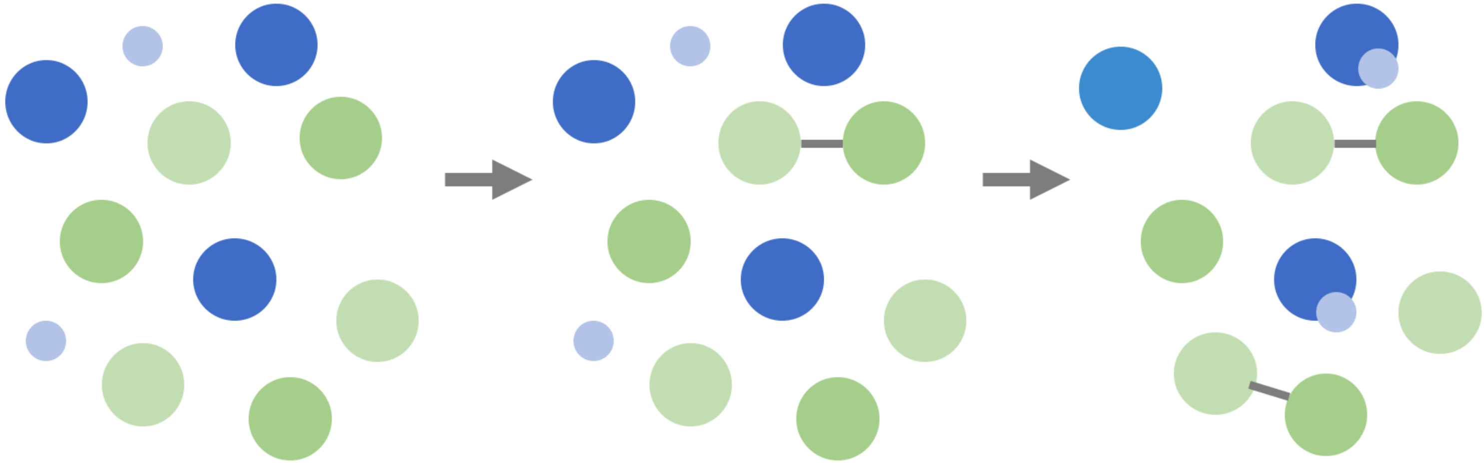
1: Predicates over models, in a logic

First, choose a modeling language: Kappa.

Kappa rules



Simulation of resulting system

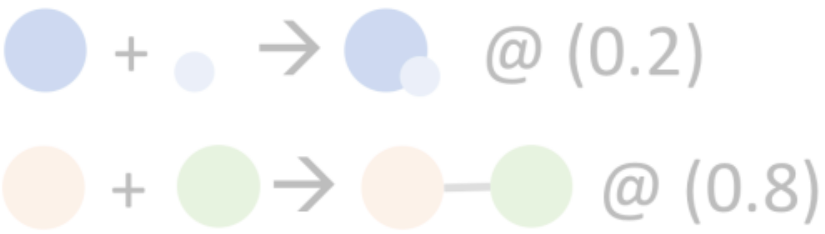


1: Predicates over models, in a logic

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.***

$$
\begin{array}{rll}
E & ::= \varepsilon \mid a \;,E & \text{(expression)}\\
a & ::= \emptyset \mid N(\sigma) & \text{(agent)}\\
N & ::= A \in \mathcal{A} & \text{(agent name)}\\
\sigma & ::= \varepsilon \mid s,\sigma & \text{(interface)}\\
s & ::= n^{\lambda} & \text{(site)}\\
n & ::= x \in \mathcal{S} & \text{(site name)}\\
\lambda & ::= \epsilon \mid i \in \mathbb{N} \mid N@n \mid - & \text{(binding state)}
\end{array}
$$

(a) Syntax.

$$E \;, A(\sigma,s,s',\sigma') \;, E' \equiv E \;, A(\sigma,s',s,\sigma') \;, E'$$
$$E \;, a \;, a' \;, E' \equiv E \;, a' \;, a \;, E'$$
$$E \equiv E \;, \emptyset$$
$$\frac{i,j \in \mathbb{N} \text{ and } i \text{ does not occur in } E}{E[i/j] \equiv E}$$
$$\frac{i \in \mathbb{N} \text{ and } i \text{ occurs only once in } E}{E[\epsilon/i] \equiv E}$$

(b) Congruence.

$$
\begin{array}{rcl}
& & \emptyset \models_{E_0} \emptyset\\
\lambda = \lambda_l & \Longrightarrow & n^{\lambda} \models^N_{E_0} n^{\lambda_l}\\
& & n^i \models^N_{E_0} n^-\\
& & n^i \models^N_{E_0} n^{\alpha_{E_0}(i,N@n)}\\
& & \sigma \models^N_{E_0} \varepsilon\\
s \models^N_{E_0} s_l \wedge \sigma \models^N_{E_0} \sigma_l & \Longrightarrow & s,\sigma \models^N_{E_0} s_l,\sigma_l\\
\sigma \models^N_{E_0} \sigma_l & \Longrightarrow & N(\sigma) \models_{E_0} N(\sigma)\\
& & E \models_{E_0} \varepsilon\\
a \models_{E_0} a_l \wedge E \models_{E_0} E_l & \Longrightarrow & a \;, E \models_{E_0} a_l \;, E_l
\end{array}
$$

(c) Matching.

$$
\begin{array}{rcl}
\emptyset[a_r] = a_r & & a_r[\emptyset] = \emptyset\\
\lambda_r \in \mathbb{N} \cup \{\epsilon\} \Longrightarrow n^{\lambda}[n^{\lambda_r}] & = & n^{\lambda_r}\\
n^{\lambda}[n^-] & = & n^{\lambda}\\
n^{\lambda}[n^{N@n}] & = & n^{\lambda}\\
\sigma[\varepsilon] & = & \sigma\\
(s,\sigma)[s_r,\sigma_r] & = & s[s_r],\sigma[\sigma_r]\\
N(\sigma)[N(\sigma_r)] & = & N(\sigma[\sigma_r])\\
E[\varepsilon] & = & E\\
(a \;, E)[a_r, E_r] & = & a[a_r] \;, E[E_r]
\end{array}
$$

(d) Replacement.

$\alpha_E(i, A@n) = A'@n'$ where the site n' in the agent A' is the unique site distinct from the site n in A in the pattern E that is tagged with i.

(e) Look-up function.

$$\frac{\begin{cases} E_0, E_1 \text{ are mixtures, } r = E_\ell, E_r \in R\\ E_0 \equiv E_0',\; E_0' \models_{E_0'} E_\ell,\; E_0'[E_r] \equiv E_1 \end{cases}}{E_0 \longrightarrow_r E_1}$$

(f) Transitions.

Fig. 3. Syntax and operational semantics.

[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 <graph, 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 <graph, action> 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 = Agent('a')
b = Agent('b')
p = And(
        ModelHasRule(lambda r:
            PregraphHas(r, a.bound(b))),
        ModelHasRule(lambda r:
            PostgraphHas(r, a.unbound(b))))
```

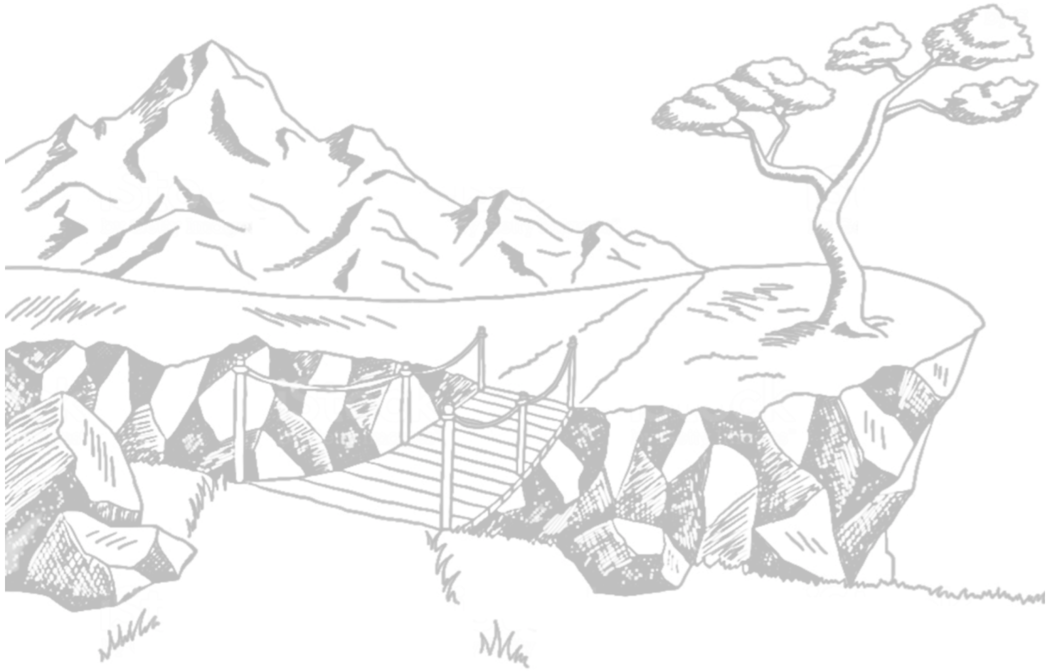
Mechanistic rules
Non-mechanistic rules
Domain knowledge

3: Create predicates

1: Predicates over models

2: Implement interpretation

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

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

z3 Microsoft Research

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

z3 Microsoft Research

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

Is this formula satisfiable?

```
(declare-fun x () Int)
(assert (>= 5 x))
(check-sat)
(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
  - 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?

```
(declare-fun x () Int)
(assert (>= 5 x))
(check-sat)
(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
  - High-performance satisfiability solver
  - Wide variety of datatypes supported: arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers

Microsoft Research

z3

Is this formula satisfiable?

```
(declare-fun x () Int)
(assert (>= 5 x))
(check-sat)
(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
  - High-performance satisfiability solver
  - Wide variety of datatypes supported: arithmetic, fixed-size bit-vectors, extensional arrays, datatypes, uninterpreted functions, and quantifiers

Microsoft Research

z3

Is this formula satisfiable?

```
(declare-fun x () Int)
(assert (>= 5 x))
(check-sat)
(get-model)
```

→

```
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/\Q is unsatisfiable, then Q is inconsistent with the existing facts)
  - Detect **redundancy** (if Q is a new fact, and P => Q, then Q is redundant)
  - Detect **ambiguity** (if model M satisfies predicate P, and P/\¬(model=M) is satisfiable, then P has multiple solutions)

Usage example: Inconsistency checking

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

Usage example: Inconsistency checking

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
```

Usage example: Redundancy checking

```
>>> from syndra.engine import macros, predicate
>>> x = macros.directly_phosphorylates("MEK", "ERK")
>>> y = macros.phosphorylated_is_active("ERK")
```

Usage example: Redundancy checking

```
>>> 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

```
>>> 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)
```

Usage example: Redundancy checking

```
>>> 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()
```

Usage example: Redundancy checking

```
>>> 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
```

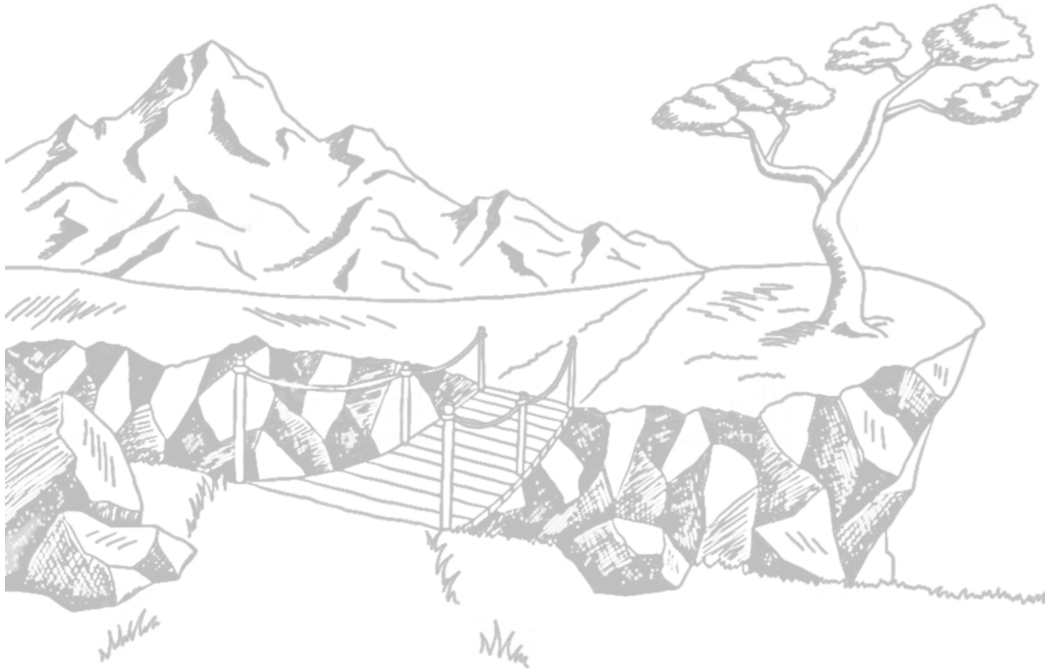
Mechanistic rules
Non-mechanistic rules
Domain knowledge

3: Create predicates →

1: Predicates over models

2: Implement interpretation →

Space of possible models



3: Tools for creating predicates

- **Macros**

3: Tools for creating predicates

- **Macros**

A phosphorylates B →

```
PreLabeled(A, phosphorylated) ∧
PreUnbound(A, B) ∧
PostLabeled(A, phosphorylated) ∧
PostBound(A, B)
```

3: Tools for creating predicates

- **Macros**

A phosphorylates B → PreLabeled(A, phosphorylated) $\wedge$ PreUnbound(A, B) $\wedge$ PostLabeled(A, phosphorylated) $\wedge$ 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]

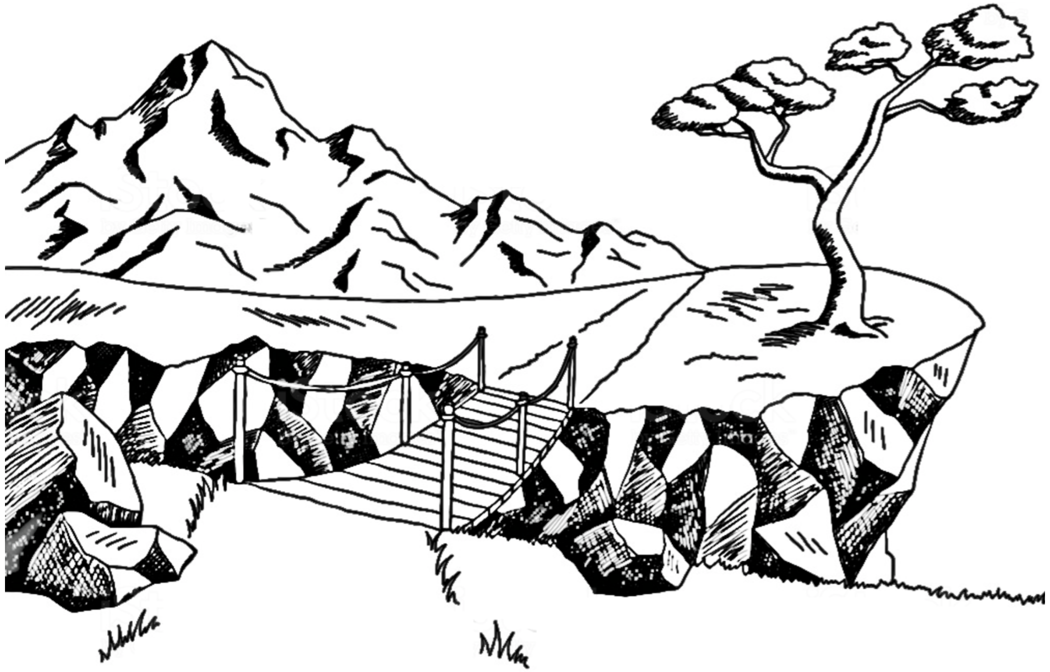
Mechanistic rules
Non-mechanistic rules
Domain knowledge

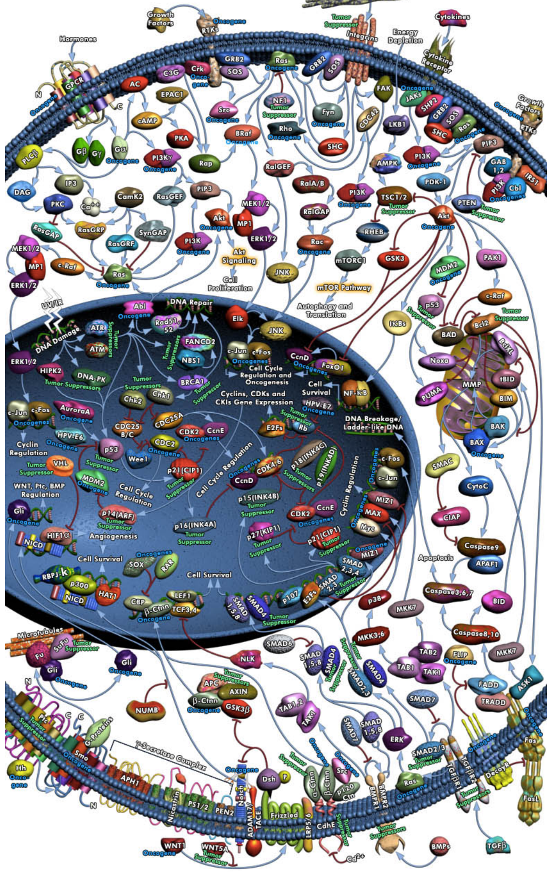
3: Create predicates →

1: Predicates over models

2: Implement interpretation →

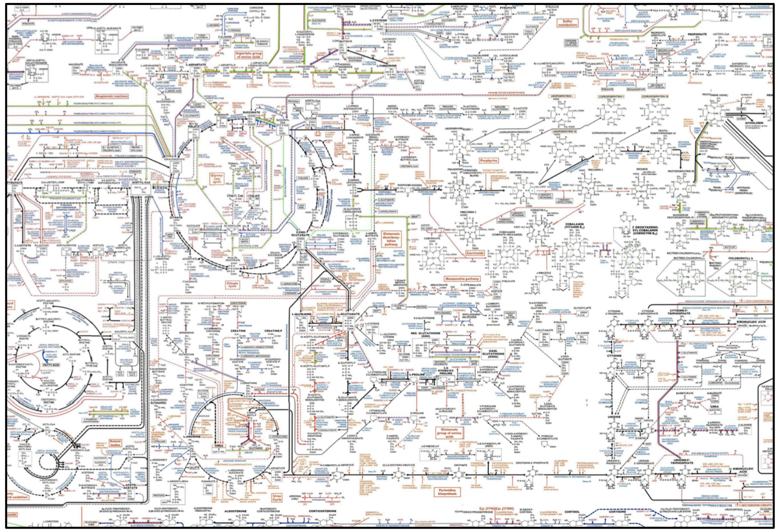
Space of possible models

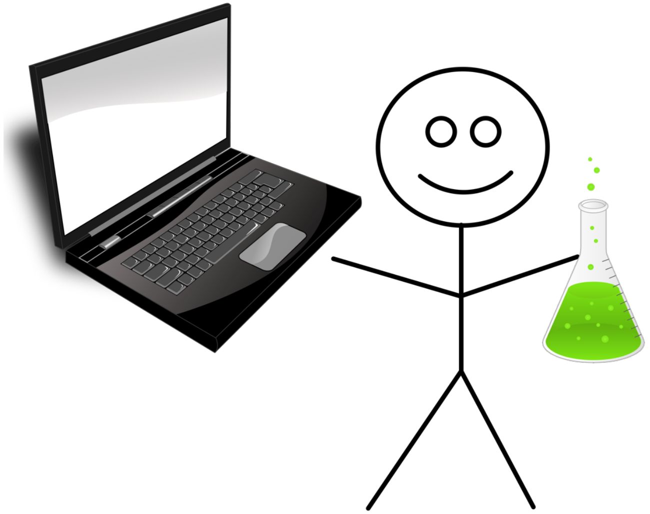


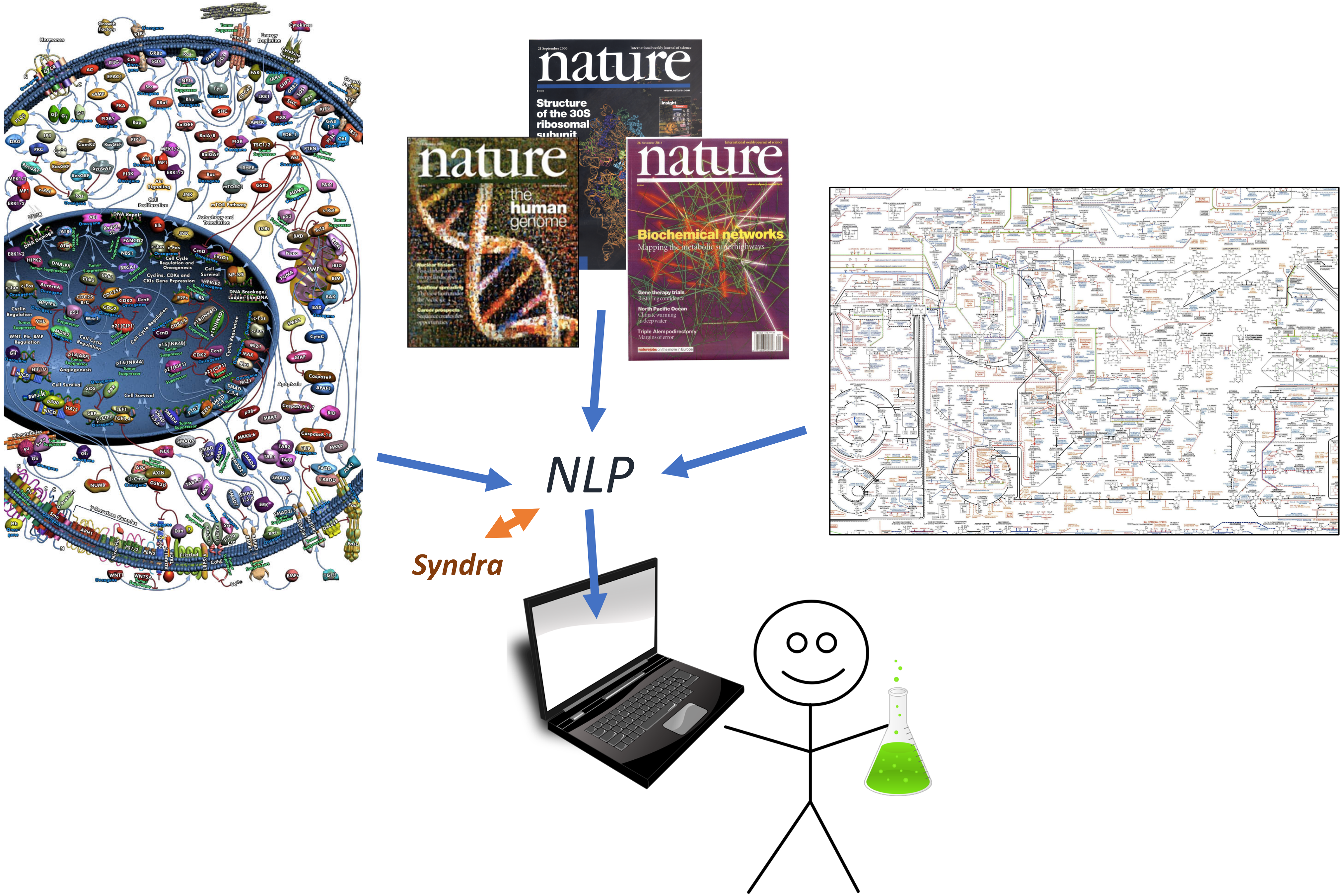



nature
Structure of the 30S ribosomal subunit
nature
the human genome
nature
Biochemical networks
Mapping the metabolic superhighways
Gene therapy trials
North Pacific Ocean
Triple Alempodirectomy

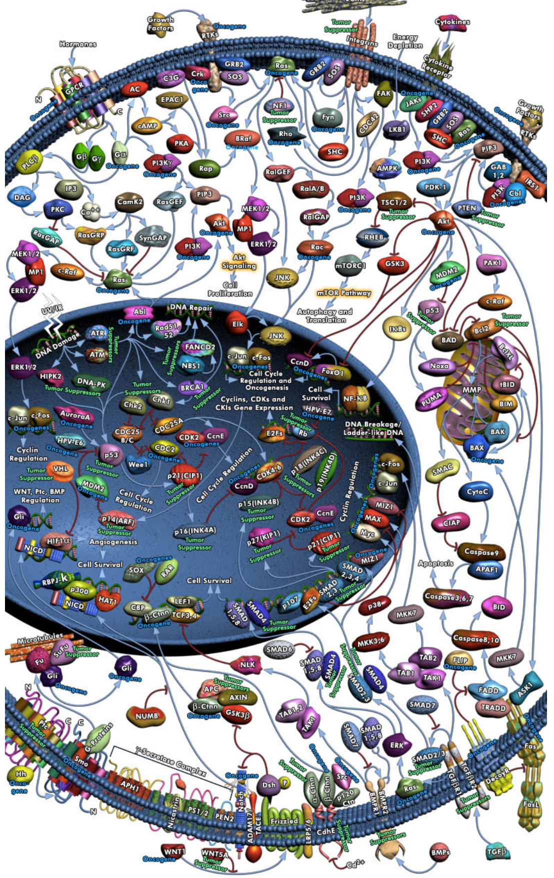
NLP

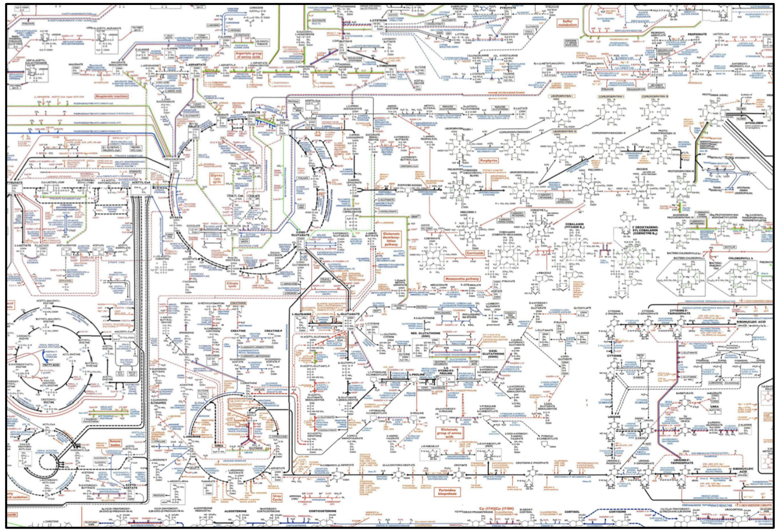






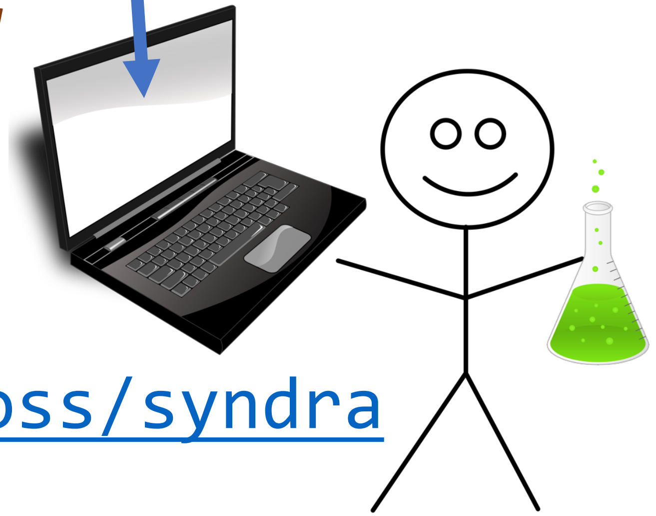
nature
Structure of the 30S ribosomal subunit
nature
the human genome
nature
Biochemical networks
Mapping the metabolic superhighways
Gene therapy trials
North Pacific Ocean
NLP
Syndra





NLP

Syndra



https://github.com/csvoss/syndra