Constraint Programming for Graph Reduction in Systems Biology

Steven Gay, Francois Fages, Sylvain Soliman

October 16, 2012

Reaction Model Reduction

- Reaction Model
- Model Reduction
- Reaction Graphs

Graph Editing Operations

- Delete and Merge Operations
- Coding as a Subgraph Epimorphim Problem

Searching a mapping

- Bruteforce
- Constraint Logic Programming

Constraintes team

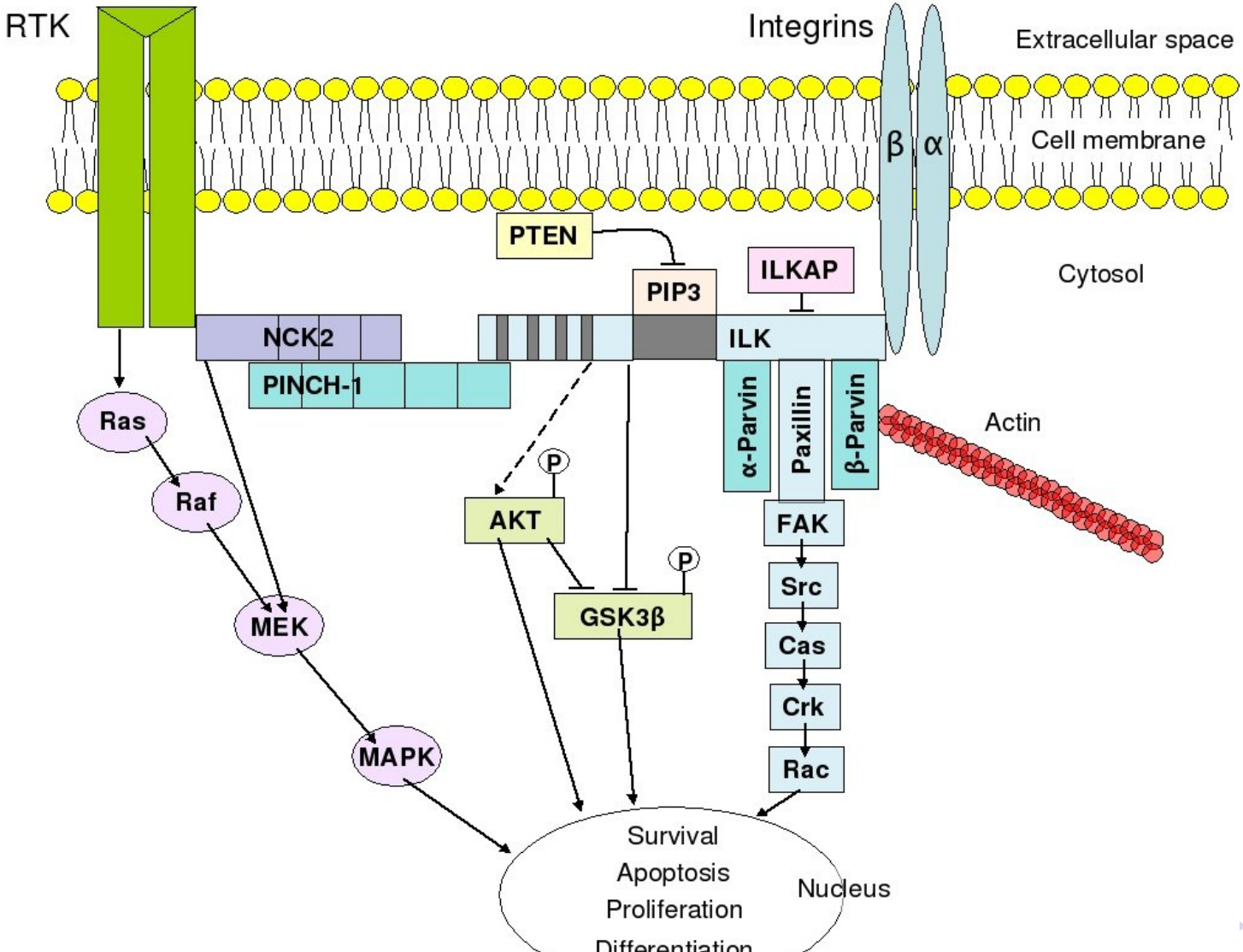
We study constraint programming and systems biology.

A unifying feature is development of *modelling languages*.

Systems Biology

Systems Biology studies interactions in big models.

Example (Molecular Cell Biology)



Reaction Model

Reaction models are used in systems biology.

- Model of a cell = { Reactions }
- Reaction = Molecules + Parametrized Rates

Example (Michaelis-Menten Reaction)

Highschool notation

$$E + S \underset{k_d}{\overset{k_c}{\rightleftharpoons}} ES \overset{k_p}{\rightarrow} E + P$$

BIOCHAM reactions

kc*[E]*[S] for E + S ⇒ ES.
kd*[ES] for ES ⇒ E + S.
kp*[ES] for ES ⇒ E + P.

Fixing *parameters* and *initial concentrations*, models are *simulated*.

Model Reduction

Large models store biological knowledge

- Kohn's model = 800 reactions, 500 molecular species

Small models are better fit to work on:

- level of abstraction fitting relative importance of parts
- Model-Parametrize-Test workflow : parametrization and simulation computationally expensive

Modelers use kinetic reductions : reduce M, get M'.

Reduction as a binary relation : is model M reducible to M'?

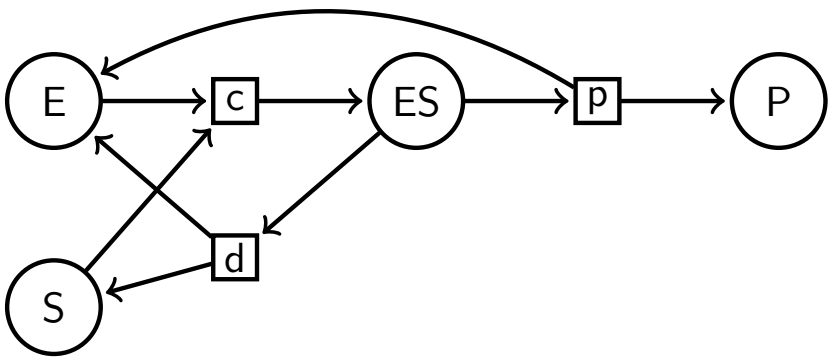
Reaction Graphs

Definition

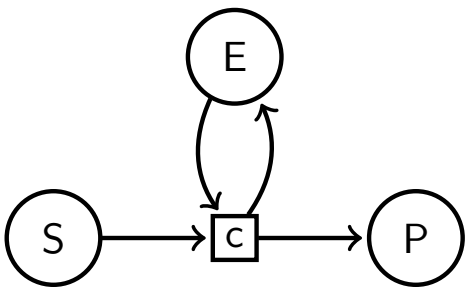
A *reaction graph* is a triple (S, R, A), with $A \subseteq (S \times R) \cup (R \times S)$. S is the set of *molecular species* of the graph, R is the set of *reactions*.

Example (Michaelis-Menten expanded and reduced)

$$E + S \rightleftharpoons ES \rightarrow E + P$$



$$E + S \rightarrow E + P$$



Model reduction by graph operations

What happens when we abstract from kinetic conditions?

$$\begin{array}{ccc} M & \rightarrow & M' \\ {\scriptstyle \text{abstraction}}\downarrow & & \\ G & \underset{?}{\rightarrow} & G' \end{array}$$

We define a model reduction to be a sequence of elementary operations:

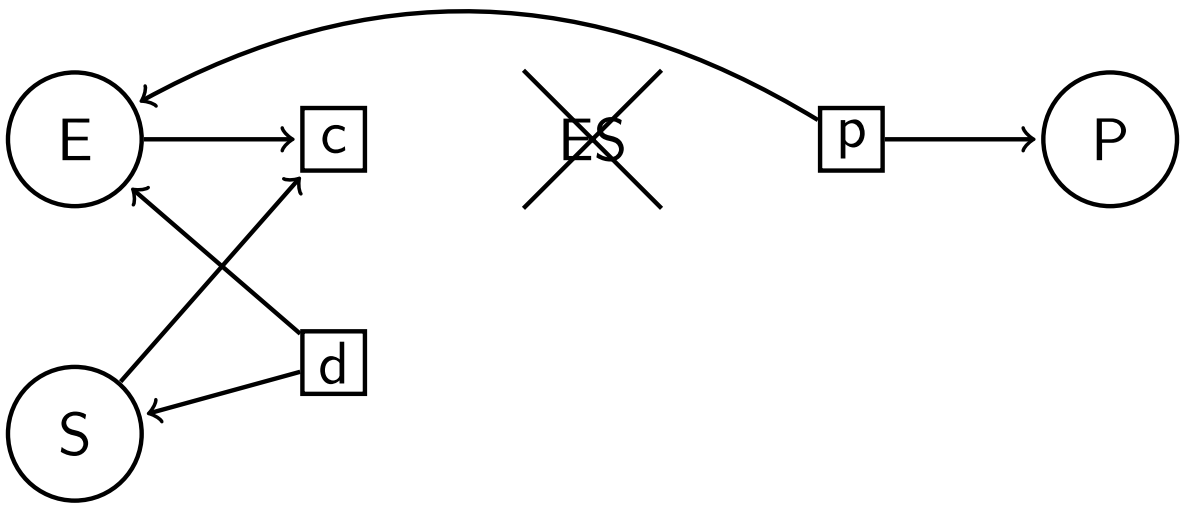
- Node deletion
- Node merging

Species Deletion

This removes a species from the model.

- Remove every arc linking the species and any reaction
- Remove the species' node from the graph

Example

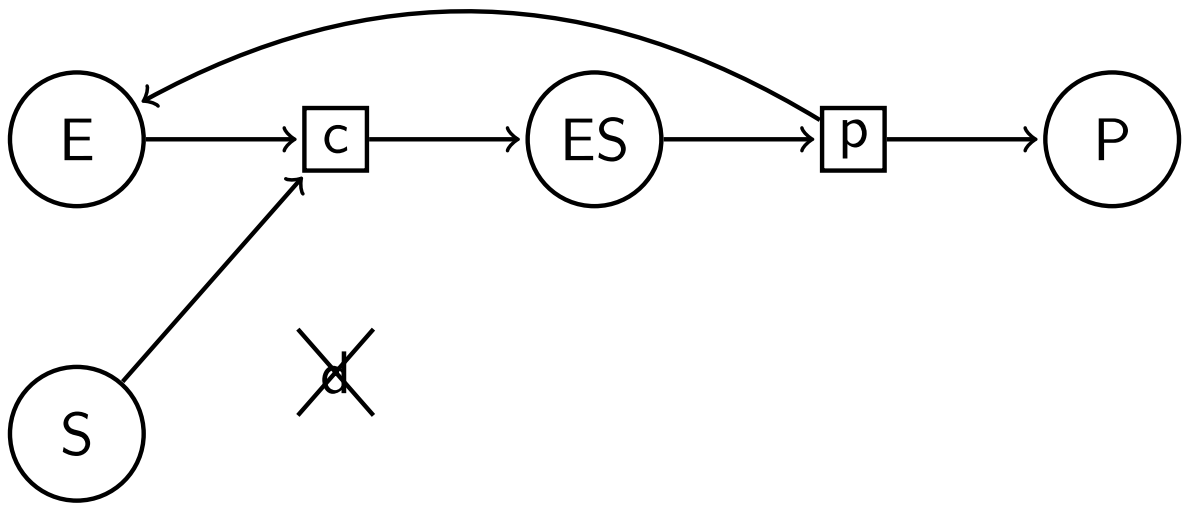


Reaction Deletion

This removes a reaction from the model.

- Remove every arc linking the reaction and a species
- Remove the reaction's node from the graph

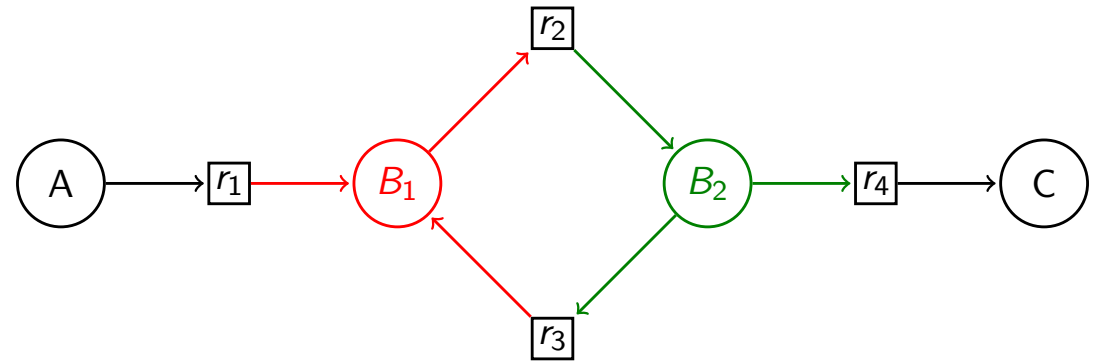
Example

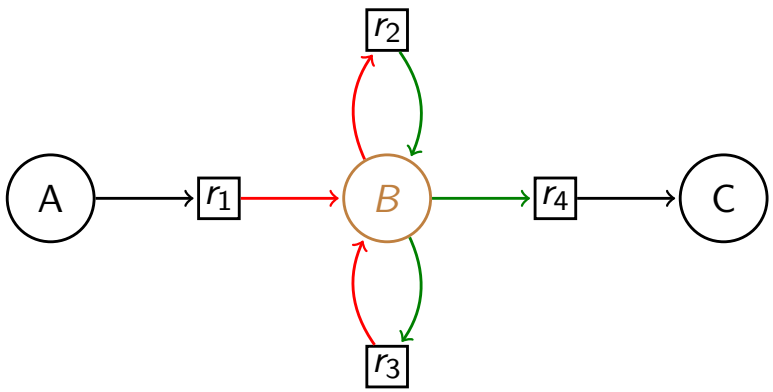


Species Merging

This merges several species $S_1 \ldots S_n$ into one:

- Create a new species node S
- For every reaction linked with an S_i, link it with S
- Delete every S_i

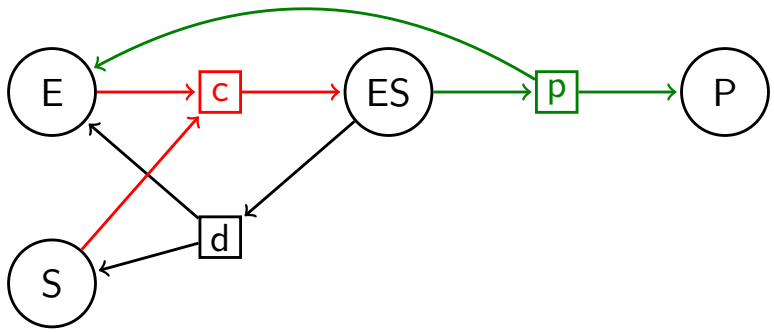


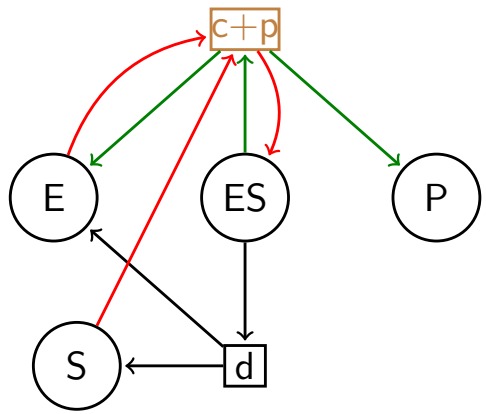


Reaction Merging

This merges several reactions $R_1 \ldots R_n$ into one:

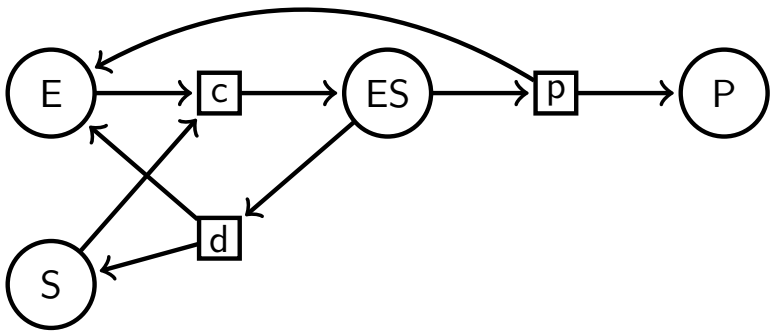
- Create a new species node R
- For every reaction linked with an R_i, link it with R
- Delete every R_i



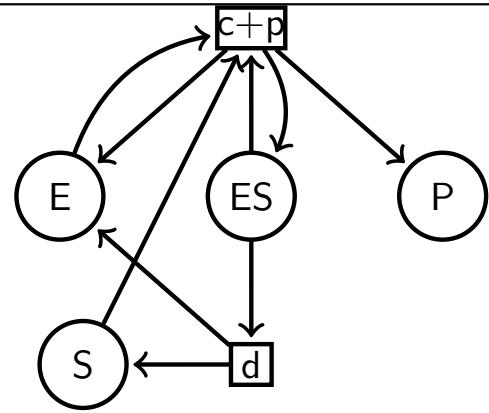


Finishing the Michaelis-Menten example

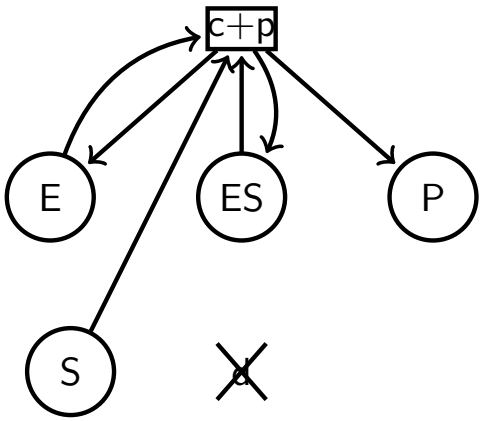
From the expanded Michaelis-Menten mechanism:



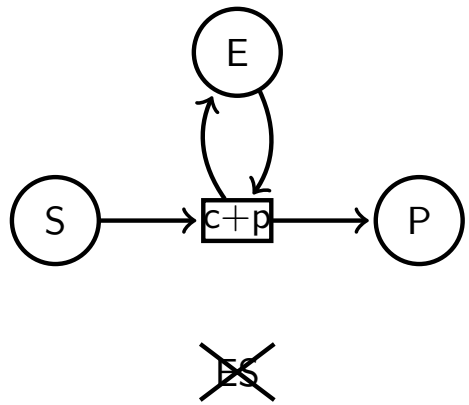
Merge c with p



Delete d



Delete ES

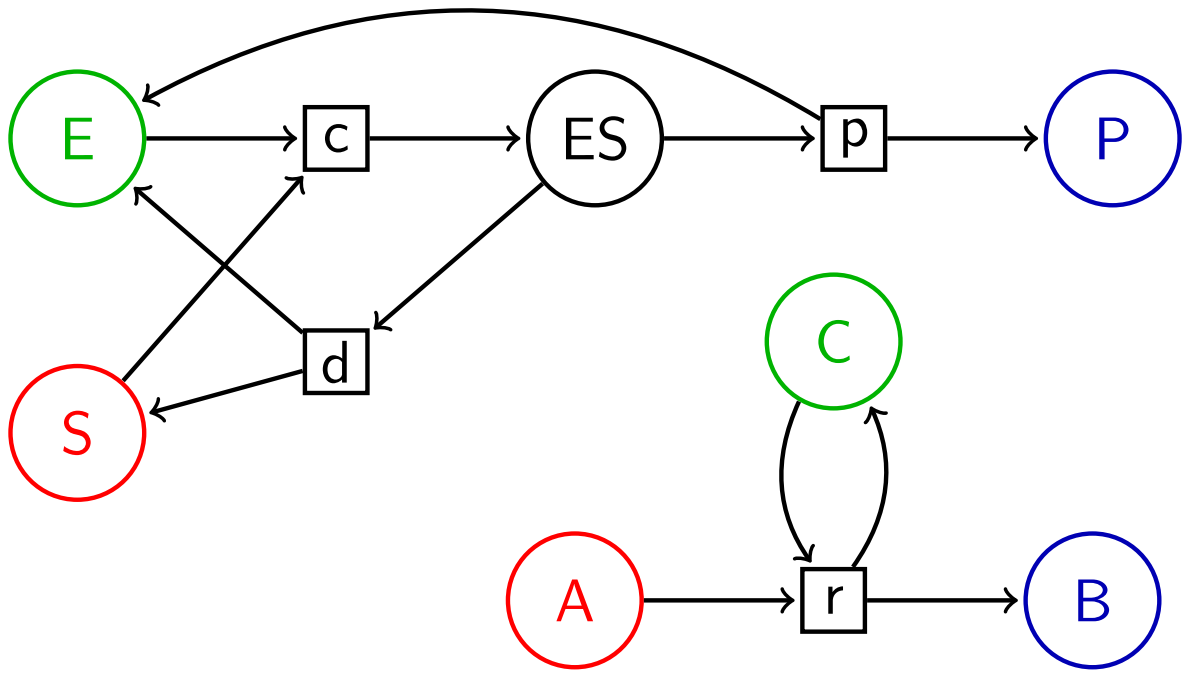


Reduced Michaelis-Menten

Model Reductions as Subgraph Epimorphisms

Sequences of deletions/mergings can be seen as a surjective mapping from one graph to another:

$E \to C$
$S \to A$
$P \to B$
$c \to r$
$p \to r$
$d \to \bot$
$\mathrm{ES} \to \bot$



Subgraph Epimorphisms

Definition (SEPI)

A *subgraph epimorphism* μ from $G = (N, A)$ to $G' = (N', A')$ is a mapping $\mu : N_0 \longrightarrow N'$,

- **Morphism:** $\forall (x, y) \in A \cap N_0 \times N_0, (\mu(x), \mu(y)) \in A'$
- **Node Surjection:** $\forall x' \in N', \exists x \in N, \mu(x) = x'$
- **Arc Surjection:**
  $\forall (x', y') \in A', \exists (x, y) \in A, (\mu(x), \mu(y)) = (x', y')$

In our setting, $\mu(S \cap N_0) \subseteq S'$, $\mu(R \cap N_0) \subseteq R'$.

Theorem (SEPI coding)

$$\exists \textit{ sequence of deletions/mergings transforming } G \textit{ into } G'$$
$$\textit{iff}$$
$$\exists \textit{ subgraph epimorphism } \mu : G \longrightarrow G'$$

Removes sequence symmetries, easier to work with.

The Subgraph Epimorphism Problem

Theorem

Given G, G', the problem :
$\exists$ subgraph epimorphism $\mu : G \longrightarrow G'$
is NP-complete.

NP-complete means there are hard instances for which the *best known* possible algorithms are as costly as *trying every possible solution* ...

Generate and test, a.k.a. Bruteforcing

Generate every function $\mu : G \longrightarrow G'$ and test SEPI conditions:

```
for every TargetNode in G' {
  image[0] = TargetNode
  for every TargetNode in G' {
    image[1] = TargetNode
    for ...

       if is_sepi(image, graph, graph') return image;

  }
}
return NULL;
```

This does not work on large instances!

Constraint Programming

Constraint programming allows the reversal of generate-and-test:

```
...
make_map(Images, TargetNodes),
is_sepi(SourceGraph, TargetGraph, Images),
...
```

to constrain and search:

```
...
constrain_sepi(SourceGraph, TargetGraph, Images),
search_map(Images, TargetNodes),
...
```

which is more efficient.

Logic programming in Prolog

Prolog is a logic-based language made for non-deterministic programming

Prolog $\supseteq$ Predicates + Backtracking

Some code first . . . calls are sequenced with ',':

```
test :-
  format("Hello "),
  format("world~N"),
  member(X, [1, 3, 4]),
  format("X = ~w~N", [X]).
```

Output:

```
Hello world
X = 1
```

Non-deterministic programming

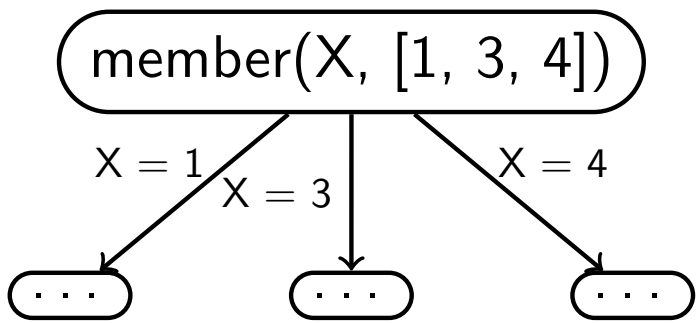
Predicates $\sim$ Functions with variable number of outputs

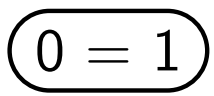
- can have zero to infinite *successes*

Example

member(X, [1, 3, 4])

$0 = 1$

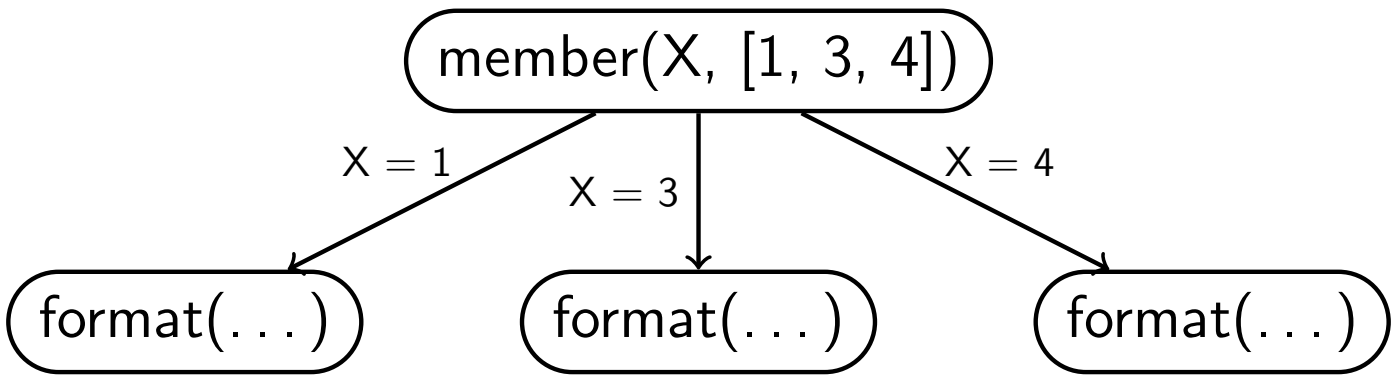




Evaluation Strategy

Successes are sequenced from first to last.

```
test :-
  member(X, [1, 3, 4]),
  format("X = ~w~N", [X]).
```



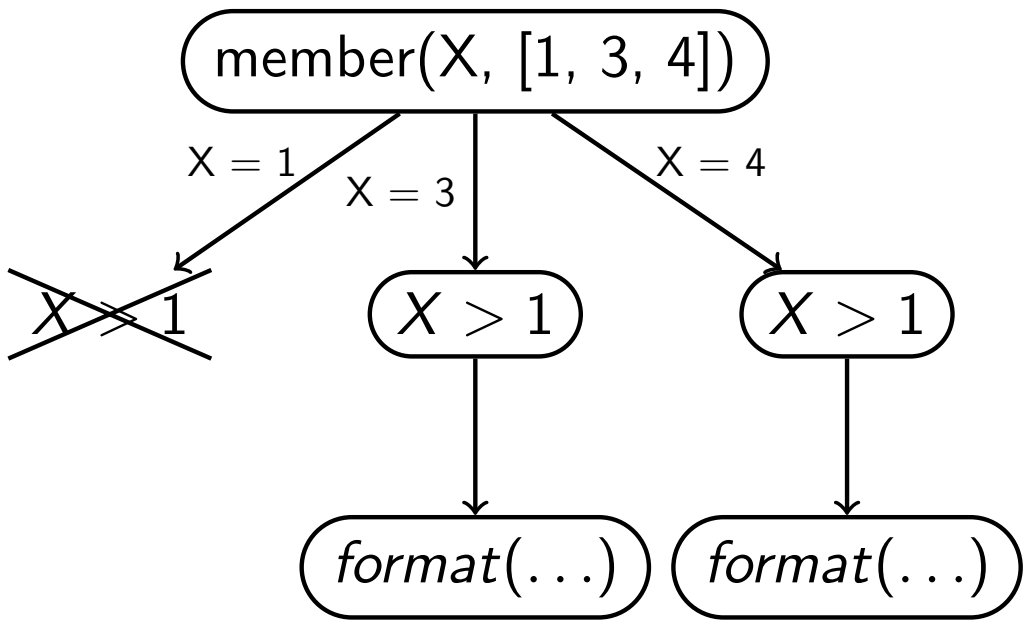
Output:

```
X = 1
```

Failure, Backtracking

When a predicate has no successes, it *fails*. Execution backtracks to the state of the last choice point, and executes the next choice:

```
test :-
  member(X, [1, 3, 4]),
  X > 1,
  format("X = ~w~N", [X]).
```



Non-deterministic programming

What does this do?

```
test :-
  member(X, [1, 3, 4]),
  format("X = ~w~N", [X]),
  X > 3.
```

Non-deterministic programming

What does this do?

```
test :-
  member(X, [1, 3, 4]),
  format("X = ~w~N", [X]),
  X > 3.
```

Output:

```
X = 1
X = 3
X = 4
```

Bruteforcing SEPI 1

These mechanisms make bruteforcing easy to code:

```
sepi :-
  ...
  member(Image0, TargetNodes),
  member(Image1, TargetNodes),
  ...
  member(ImageN, TargetNodes),
  is_sepi(Images, SourceGraph, TargetGraph),
  ...
```

The program is easier to write, but not more efficient!

Constraint Programming

Failure can be guessed before a candidate is completely generated. Constraints tell when failure will happen, thus taking generate-and-test:

```
...
make_map(Images, TargetNodes),
is_sepi(SourceGraph, TargetGraph, Images),
...
```

and replacing it by constrain and search:

```
...
constrain_sepi(SourceGraph, TargetGraph, Images),
search_map(Images, TargetNodes),
...
```

CLP(FD)

CLP(FD) introduces finite domain variables to LP.

- FD variables are given a domain
  - domain = { possible values }
- Constraints are added on FD variables
  - propagators watch their variables
- An assignment is *searched*
  - variables must be in domain, values must satisfy constraints

```
fd_domain(F, [1, 3, 4]).
F #> 1,
fd_labeling(F),
format("F = ~w~N").
```

Output:
F = 3

```
member(X, [1, 3, 4]).
X > 1,
format("X = ~w~N").
```

Output:
X = 3

FD variables know their domain

CLP(FD) introduces finite domain variables to LP.

- **FD variables are given a domain**
  - domain $=$ { possible values }
- Constraints are added on FD variables
  - propagators watch their variables
- An assignment is *searched*
  - variables must be in domain, values must satisfy constraints

```
fd_domain(F, [1, 3, 4]).
format("F = ~w~N").
```

Output:

```
F = _#2(1:3:4)
```

```
format("X = ~w~N").
```

Output:

```
X = _2
```

Constraints restrict domains

CLP(FD) introduces finite domain variables to LP.

- FD variables are given a domain
  - domain $=$ { possible values }
- **Constraints are added on FD variables**
  - propagators watch their variables
- An assignment is *searched*
  - variables must be in domain, values must satisfy constraints

```
fd_domain(F, [1, 3, 4]).
F #> 1,
format("F = ~w~N").
```

Output:

```
F = _#2(3:4)
```

```
X > 1,
format("X = ~w~N").
```

Output:

```
uncaught exception:
error(instantiation_error,(>)/2)
```

FD variables' values can be enumerated on

CLP(FD) introduces finite domain variables to LP.

- FD variables are given a domain
  - domain = { possible values }
- Constraints are added on FD variables
  - propagators watch their variables
- **An assignment is searched**
  - variables must be in domain, values must satisfy constraints

```
fd_domain(F, [1, 3, 4]).
F #> 1,
fd_labeling(F),
format("F = ~w~N"),
fail.
```

Output:

```
F = 3
F = 4
no
```

Propagation (see Thierry Martinez's talk)

Deductions are made dynamically:

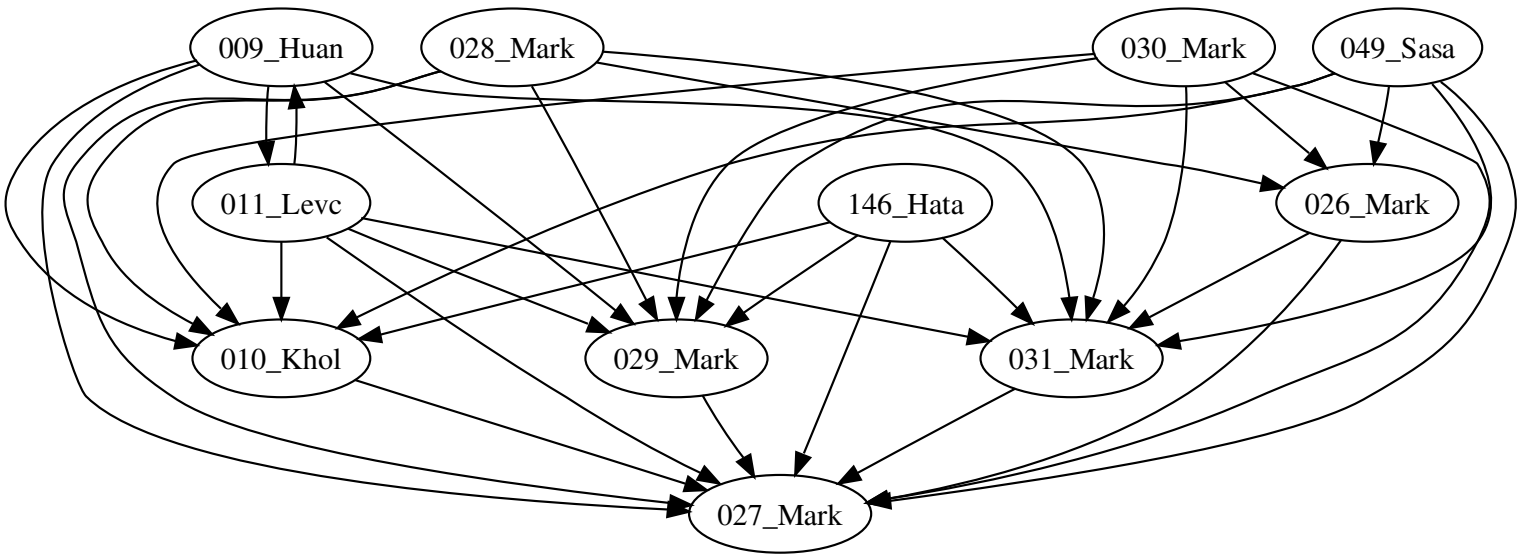
- FD variable is modified $\Rightarrow$ propagators woken up
- Propagators remove impossible values from domains
- Search fails when a domain is empty

```
fd_domain(F, [1, 2, 3]).
fd_domain(G, [1, 2, 3]).
format(...),                    F = _1#(1:2:3), G = _2#(1:2:3)

F #> G,
format(...),                    F = _1#(2:3), G = _2#(1:2)

G = 2
format(...).                    F = 3, G = 2
```

Final Word On SEPI

Problem : Find an assignment from 40 nodes to 20.
Bruteforce : 20^{40} possible assignments to check.
CLP : $<$ 1s most of the time!

Example (MAPK models as a hierarchy)



Thank you!