Answer Set Programming

Torsten Schaub

University of Potsdam

FMCAD@Cambridge

Rough Roadmap

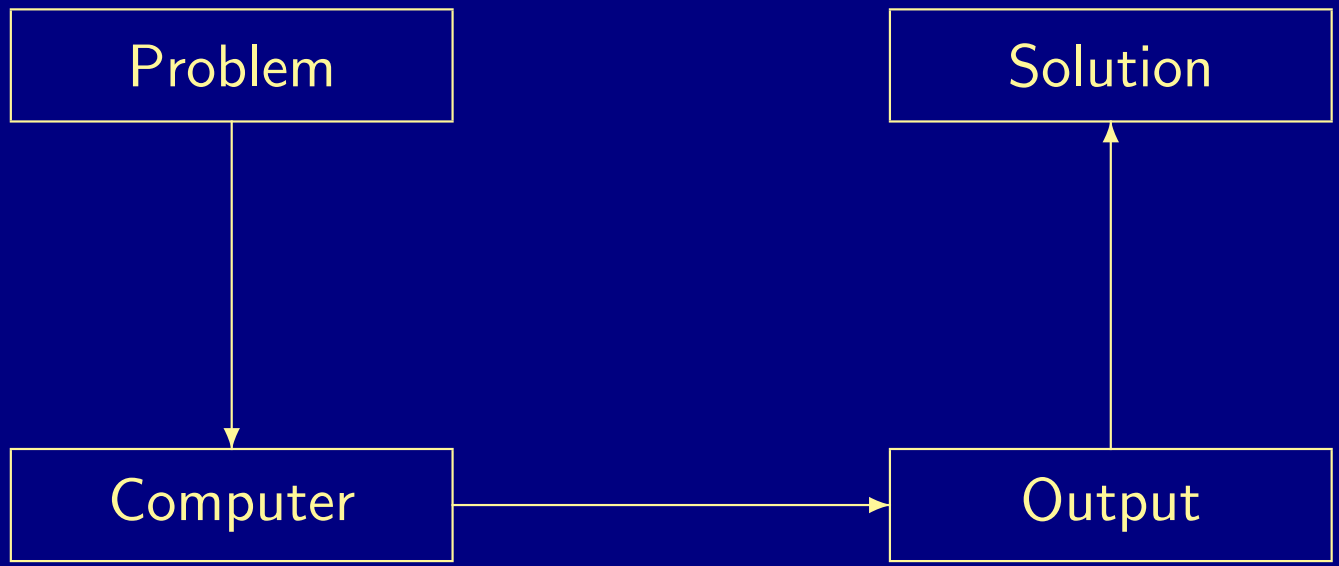
1. Motivation
2. Introduction
3. Modeling
4. Systems
5. Conclusion

Motivation: Overview

Outline

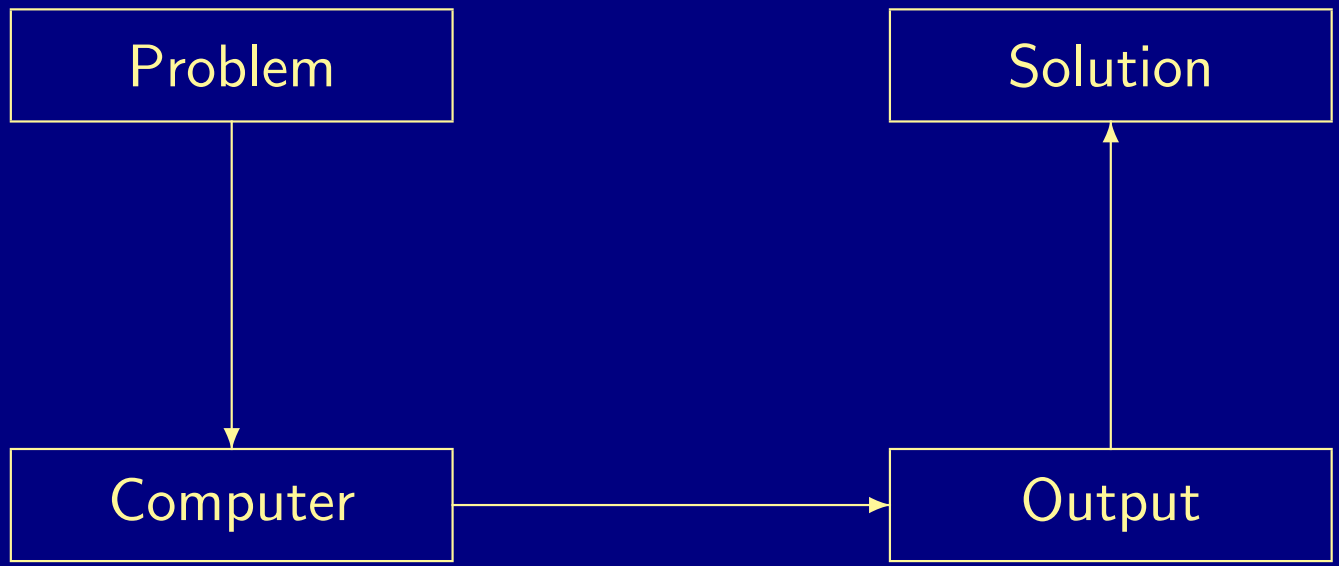
Informatics

"What is the problem?" versus *"How to solve the problem?"*



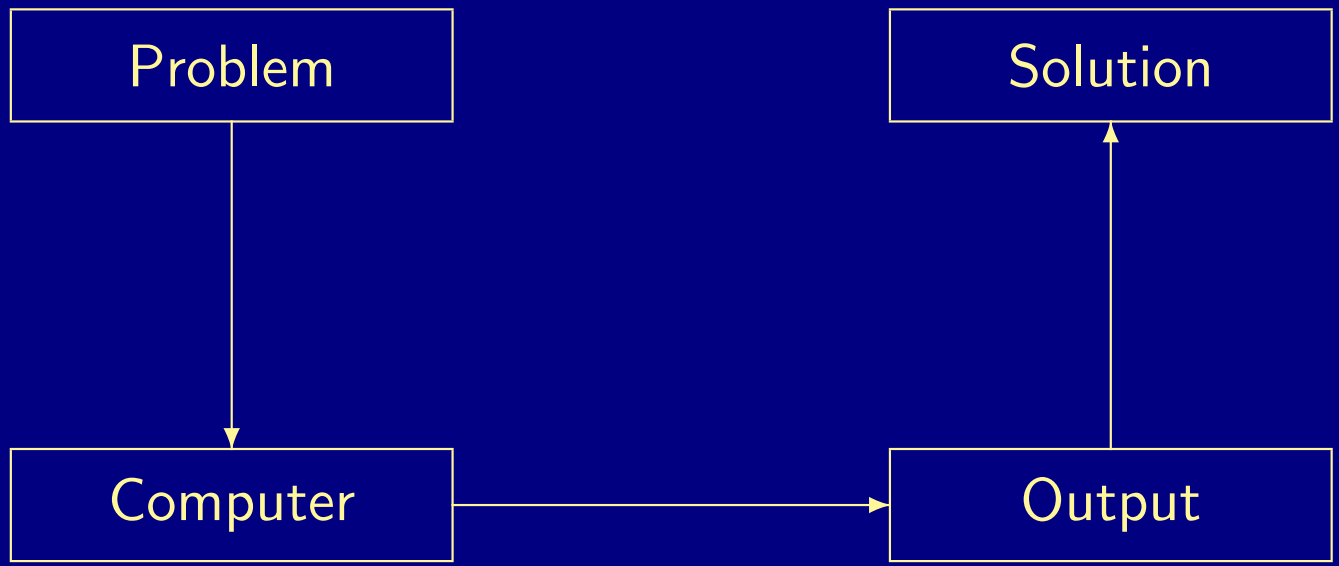
Informatics

"What is the problem?" versus *"How to solve the problem?"*



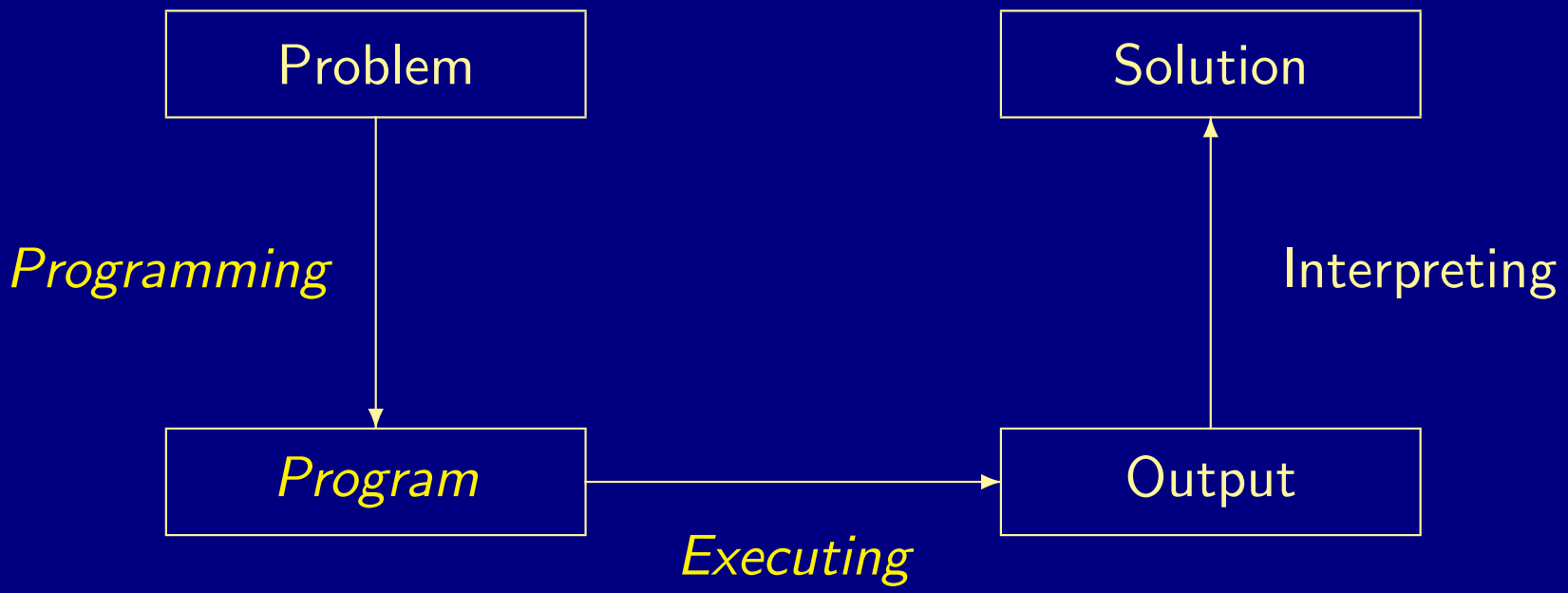
Traditional programming

"What is the problem?" versus *"How to solve the problem?"*



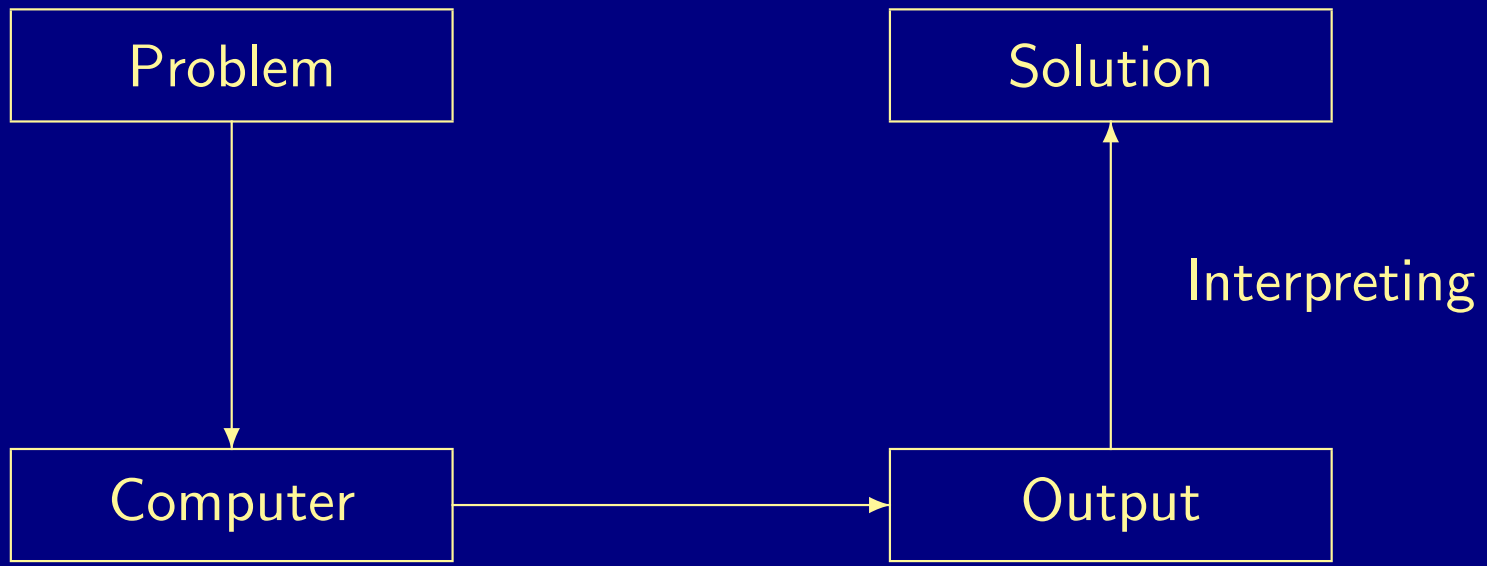
Traditional programming

"What is the problem?" versus *"How to solve the problem?"*



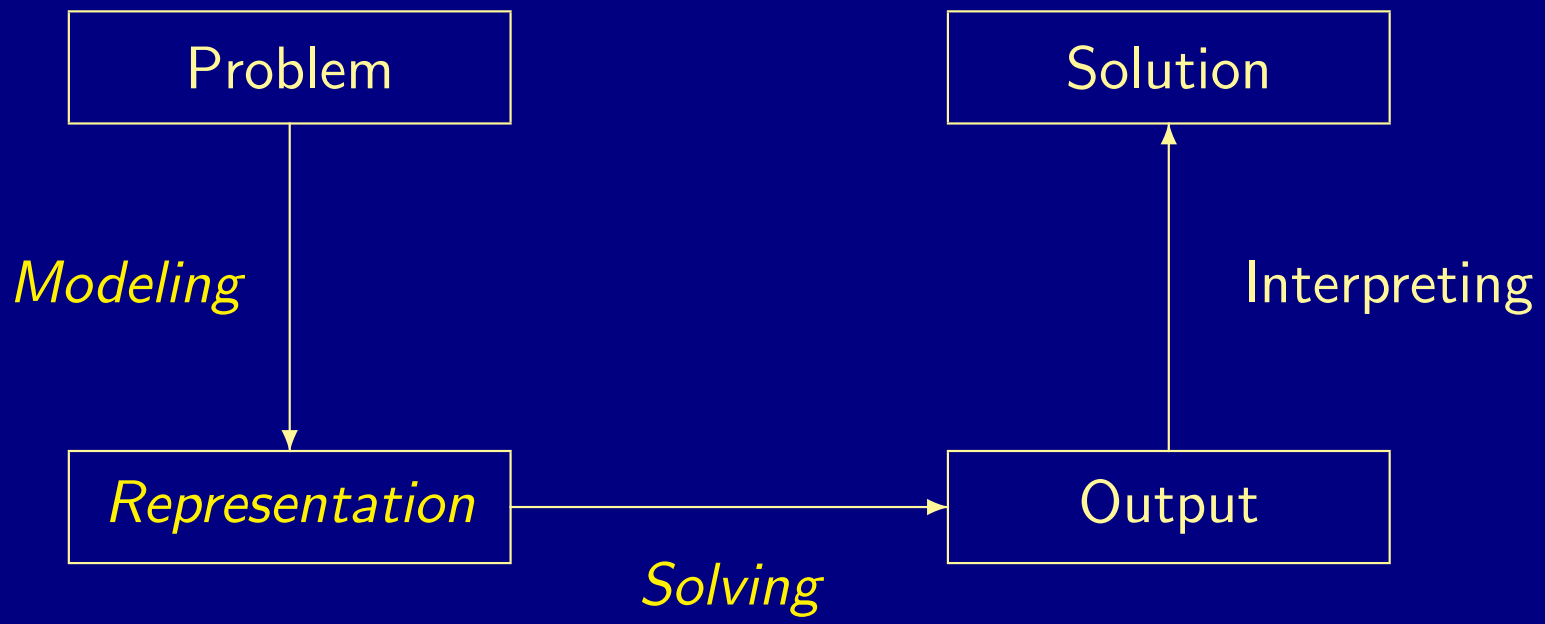
Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



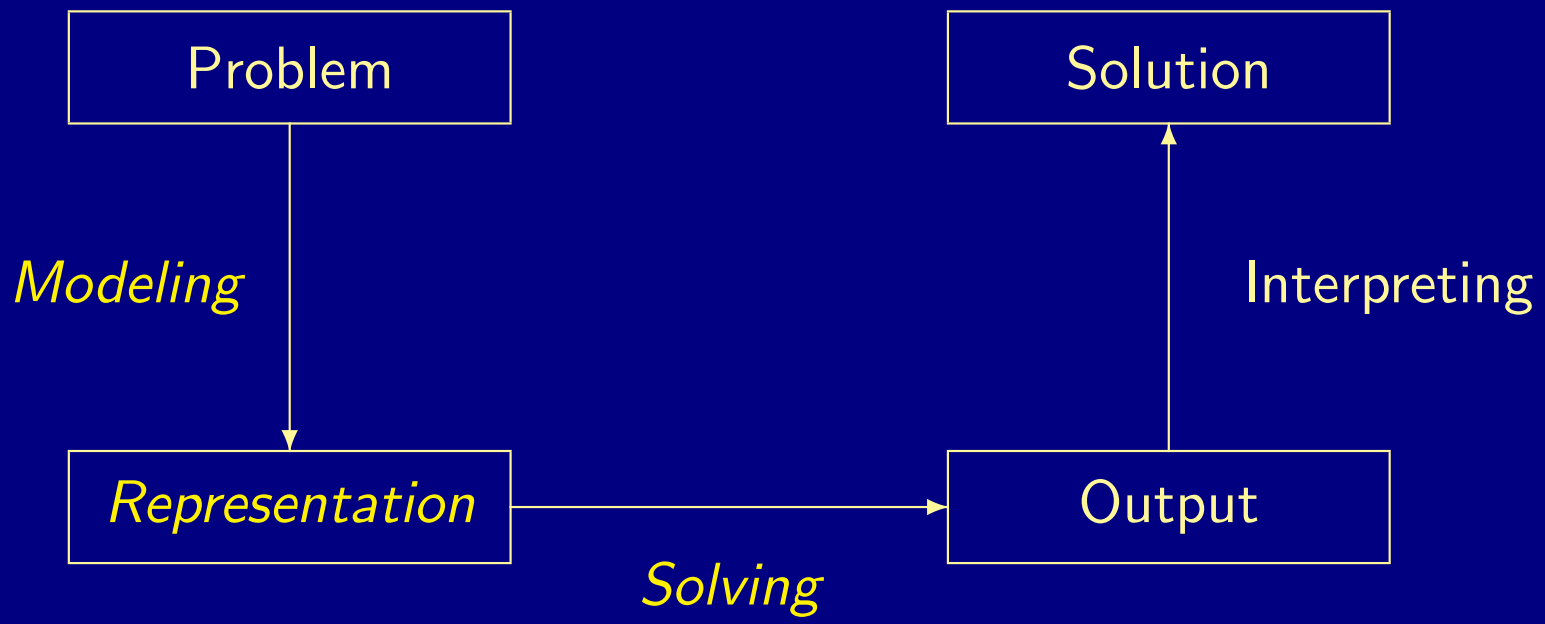
Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



Outline

Answer Set Programming

in a Nutshell

ASP is an approach to declarative problem solving, combining
- a rich yet simple modeling language
- with high-performance solving capacities

ASP has its roots in
- (deductive) databases
- logic programming (with negation)
- (logic-based) knowledge representation and (nonmonotonic) reasoning
- constraint solving (in particular, SATisfiability testing)

ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way

ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP, CASC, MISC, PB, and SAT competitions

ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas

Answer Set Programming

in a Hazelnutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning

Answer Set Programming

in a Hazelnutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning

ASP = DB+LP+KR+SAT

Outline

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Model Generation based Problem Solving

Representation	Solution	
constraint satisfaction problem	assignment	
propositional horn theories	smallest model	
propositional theories	models	**SAT**
propositional theories	minimal models	
propositional theories	stable models	
propositional programs	minimal models	
propositional programs	supported models	
propositional programs	stable models	
first-order theories	models	
first-order theories	minimal models	
first-order theories	stable models	
first-order theories	Herbrand models	
auto-epistemic theories	expansions	
default theories	extensions	
⋮	⋮	

LP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Prolog queries

```
?- above(a,c).
true.

?- above(c,a).
no.
```

LP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Prolog queries

```
?- above(a,c).
true.

?- above(c,a).
no.
```

LP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Prolog queries

```
?- above(a,c).
true.

?- above(c,a).
no.
```

LP-style playing with blocks

Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Prolog queries (testing entailment)

```
?- above(a,c).
true.

?- above(c,a).
no.
```

LP-style playing with blocks

Shuffled Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
```

Prolog queries

```
?- above(a,c).

Fatal Error: local stack overflow.
```

LP-style playing with blocks

Shuffled Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
```

Prolog queries

```
?- above(a,c).

Fatal Error: local stack overflow.
```

LP-style playing with blocks

Shuffled Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
```

Prolog queries (answered via fixed execution)

```
?- above(a,c).

Fatal Error: local stack overflow.
```

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model

$$
\left\{
\begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array}
\right\}
$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model

$$
\left\{
\begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array}
\right\}
$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model (among 426!)

$$
\left\{
\begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array}
\right\}
$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model (among 426!)

$$
\left\{
\begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array}
\right\}
$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model (among 426!)

$$
\left\{
\begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array}
\right\}
$$

Outline

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation

KR's shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation

➥ Answer Set Programming (ASP)

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *at large*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *commonly*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *in practice*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *in practice*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
first-order programs	stable Herbrand models

ASP-style playing with blocks

Logic program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Stable Herbrand model

$\{ \texttt{on(a,b)}, \texttt{on(b,c)}, \texttt{above(b,c)}, \texttt{above(a,b)}, \texttt{above(a,c)} \}$

ASP-style playing with blocks

Logic program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Stable Herbrand model

$\{ \texttt{on(a,b)}, \texttt{on(b,c)}, \texttt{above(b,c)}, \texttt{above(a,b)}, \texttt{above(a,c)} \}$

ASP-style playing with blocks

Logic program

```
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

Stable Herbrand model (and no others)

$\{\ \mathtt{on(a,b)},\ \mathtt{on(b,c)},\ \mathtt{above(b,c)},\ \mathtt{above(a,b)},\ \mathtt{above(a,c)}\ \}$

ASP-style playing with blocks

Logic program

```
on(a,b).
on(b,c).

above(X,Y) :- above(Z,Y), on(X,Z).
above(X,Y) :- on(X,Y).
```

Stable Herbrand model (and no others)

$\{ \texttt{on(a,b)}, \texttt{on(b,c)}, \texttt{above(b,c)}, \texttt{above(a,b)}, \texttt{above(a,c)} \}$

ASP-style playing with blocks

Logic program

```
on(a,b).
on(b,c).

above(X,Y) :- above(Z,Y), on(X,Z).
above(X,Y) :- on(X,Y).
```

Stable Herbrand model (and no others)

$\{ \texttt{on(a,b)}, \texttt{on(b,c)}, \texttt{above(b,c)}, \texttt{above(a,b)}, \texttt{above(a,c)} \}$

ASP versus LP

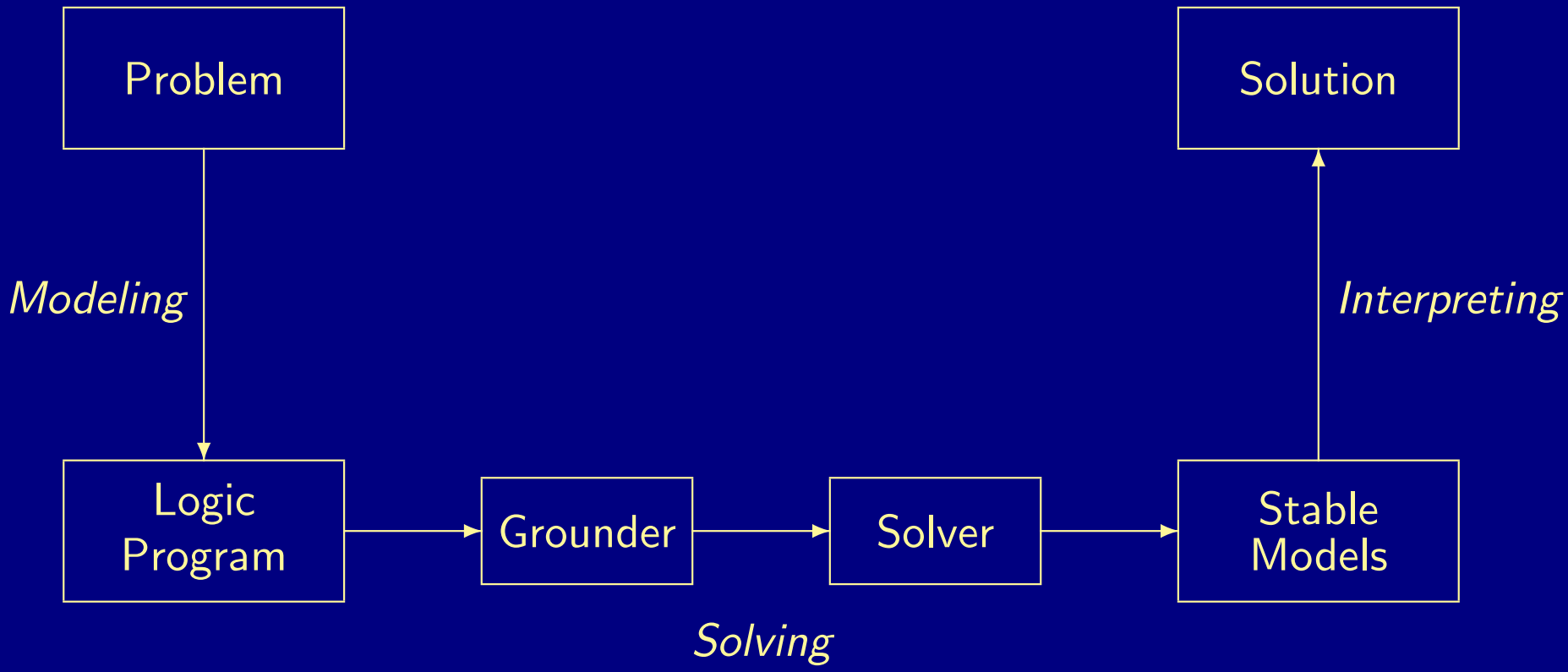
ASP	Prolog
Model generation	Query orientation
Bottom-up	Top-down
Modeling language	Programming language
Rule-based format	
Instantiation Flat terms	Unification Nested terms
(Turing +) $NP(^{NP})$	Turing

ASP versus SAT

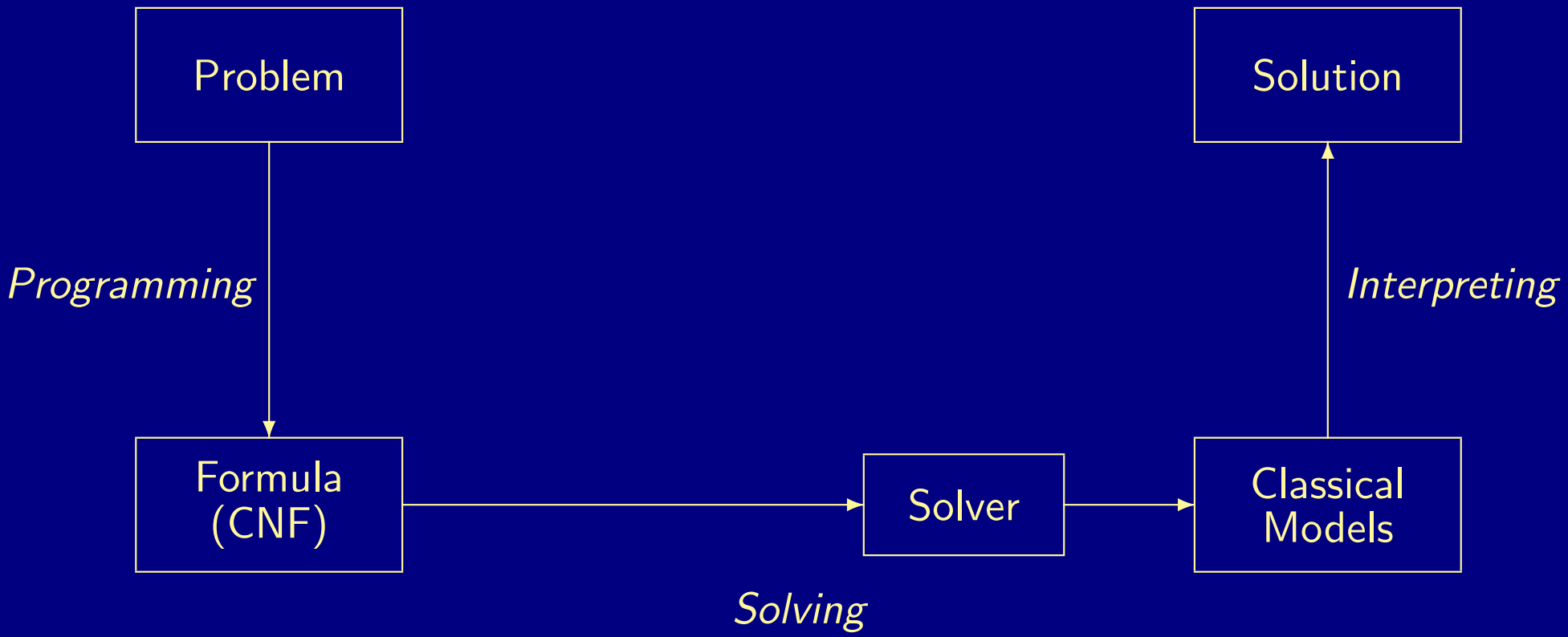
ASP	SAT
Model generation	
Bottom-up	
Constructive Logic	Classical Logic
Closed (and open) world reasoning	Open world reasoning
Modeling language	—
Complex reasoning modes	Satisfiability testing
Satisfiability	Satisfiability
Enumeration/Projection	—
Optimization	—
Intersection/Union	—
(Turing +) $NP(^{NP})$	NP

Outline

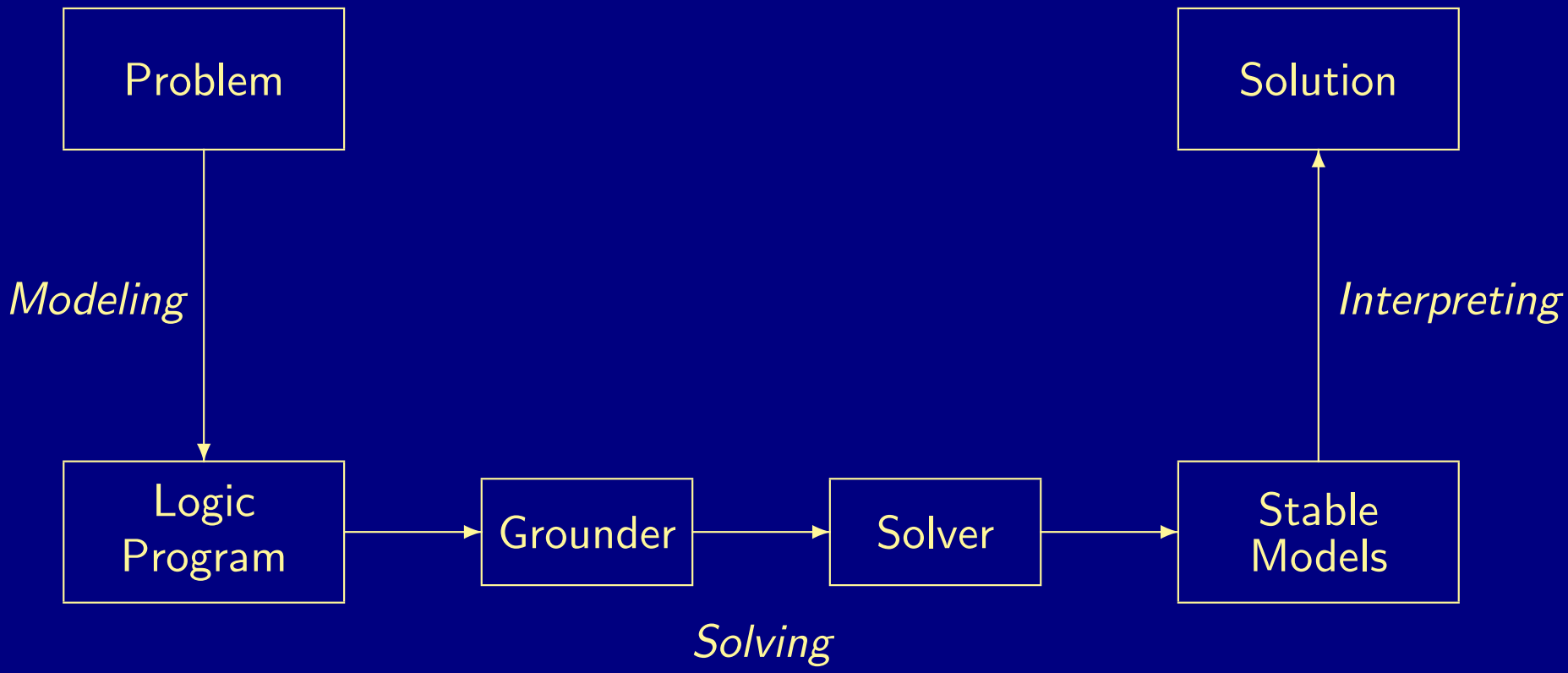
ASP solving


Problem
Solution
Modeling
Interpreting
Logic Program
Grounder
Solver
Stable Models
Solving

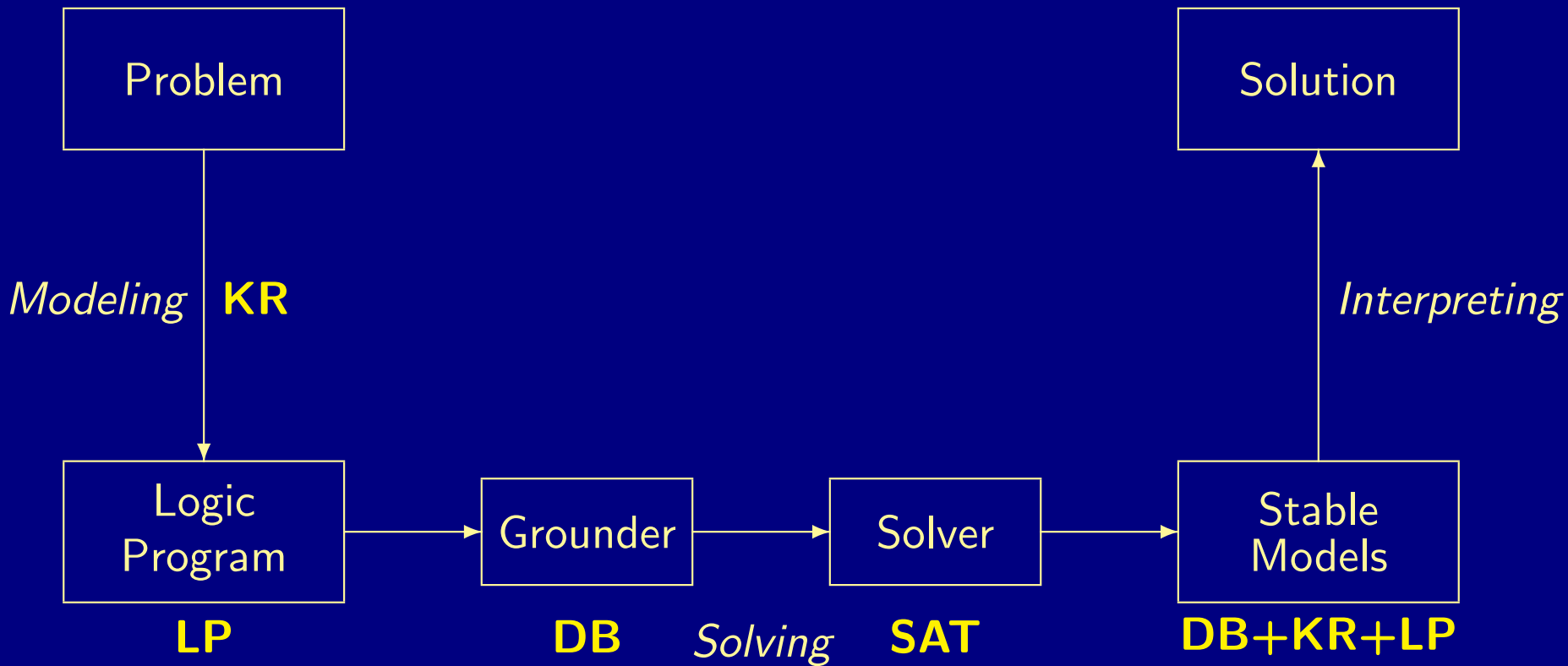
SAT solving


Problem
Solution
Programming
Interpreting
Formula (CNF)
Solver
Classical Models
Solving

Rooting ASP solving



Rooting ASP solving



Outline

Two sides of a coin

- ASP as High-level Language
  - Express problem instance(s) as sets of facts
  - Encode problem (class) as a set of rules
  - Read off solutions from stable models of facts and rules
- ASP as Low-level Language
  - Compile a problem into a logic program
  - Solve the original problem by solving its compilation

What is ASP good for?

- Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like
  - Automated Planning
  - Code Optimization
  - Composition of Renaissance Music
  - Database Integration
  - Decision Support for NASA shuttle controllers
  - Model Checking
  - Product Configuration
  - Robotics
  - System Biology
  - System Synthesis
  - (industrial) Team-building
  - and many many more

What is ASP good for?

- Combinatorial search problems in the realm of P, NP, and NP^{NP} (some with substantial amount of data), like
  - Automated Planning
  - Code Optimization
  - Composition of Renaissance Music
  - Database Integration
  - Decision Support for NASA shuttle controllers
  - Model Checking
  - Product Configuration
  - Robotics
  - System Biology
  - System Synthesis
  - (industrial) Team-building
  - and many many more

What does ASP offer?

- Integration of DB, KR, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc

What does ASP offer?

- Integration of DB, KR, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc

ASP = DB+LP+KR+SAT

Introduction: Overview

7 Syntax

8 Semantics

9 Examples

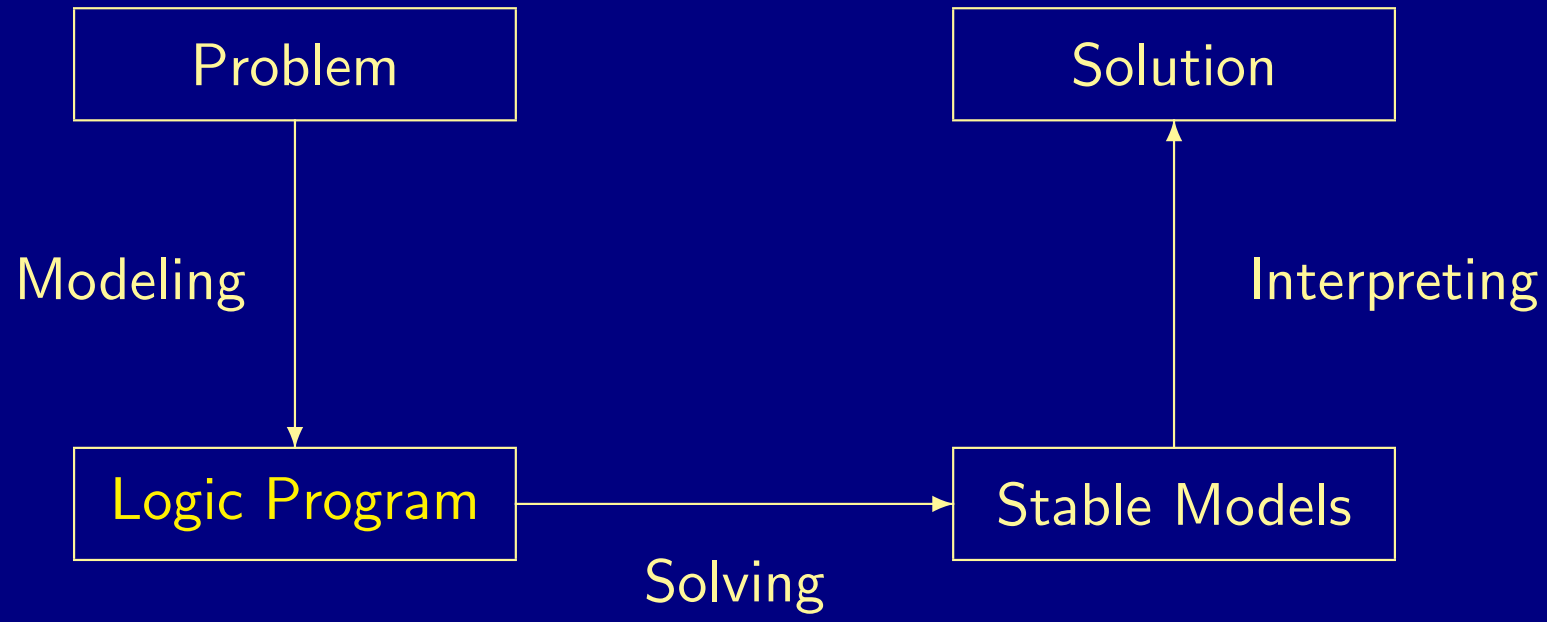
10 Variables

11 Language constructs

12 Reasoning modes

Outline

Problem solving in ASP: Syntax



Normal logic programs

- A (normal) logic program over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is an ordered pair of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

Notation

$$\begin{aligned} head(r) &= a_0 \\ body(r) &= \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\} \\ body(r)^+ &= \{a_1, \ldots, a_m\} \\ body(r)^- &= \{a_{m+1}, \ldots, a_n\} \end{aligned}$$

A program is called positive if $body(r)^- = \emptyset$ for all its rules

Normal logic programs

- A (normal) logic program over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is an ordered pair of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} head(r) &= a_0 \\ body(r) &= \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\} \\ body(r)^+ &= \{a_1, \ldots, a_m\} \\ body(r)^- &= \{a_{m+1}, \ldots, a_n\} \end{aligned}$$

- A program is called positive if $body(r)^- = \emptyset$ for all its rules

Normal logic programs

- A (normal) logic program over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is an ordered pair of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} head(r) &= a_0 \\ body(r) &= \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\} \\ body(r)^+ &= \{a_1, \ldots, a_m\} \\ body(r)^- &= \{a_{m+1}, \ldots, a_n\} \end{aligned}$$

- A program is called positive if $body(r)^- = \emptyset$ for all its rules

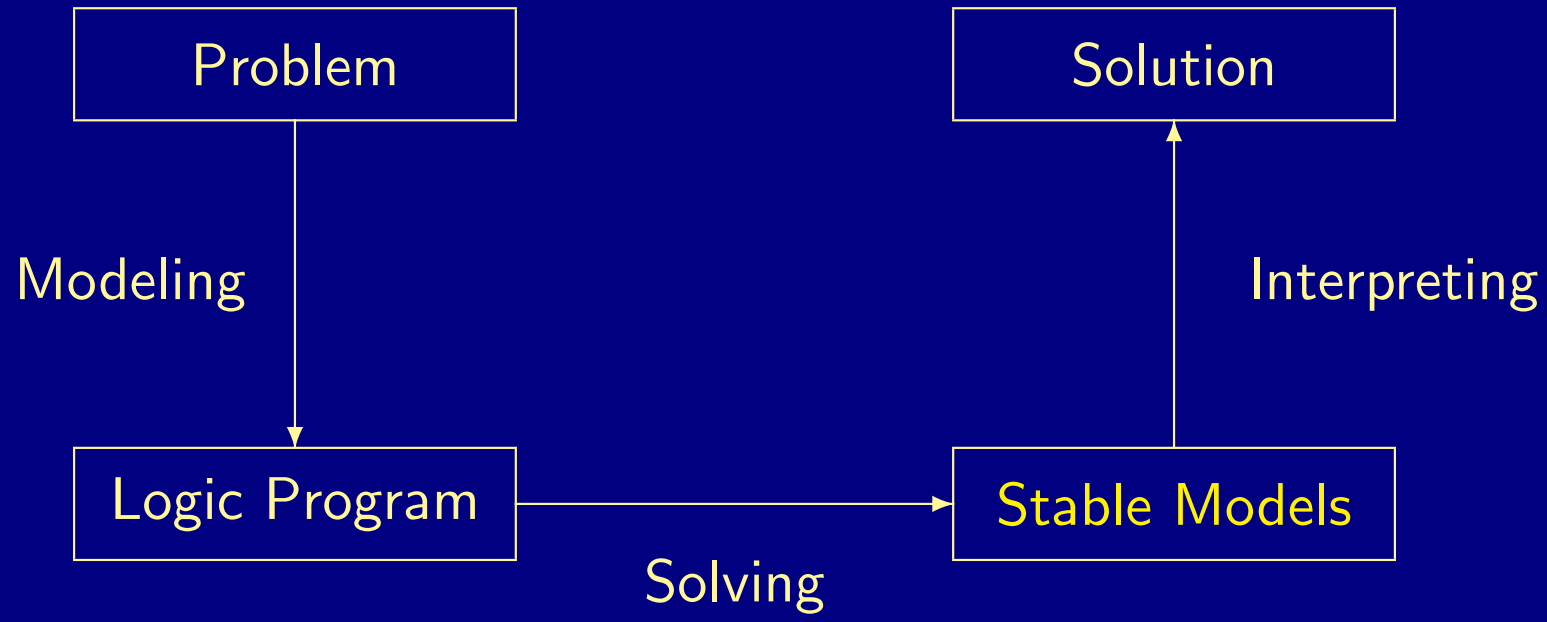
Rough notational convention

We sometimes use the following notation interchangeably
in order to stress the respective view:

	true, false	if	and	or	iff	default negation	classical negation
source code		`:-`	`,`	`|`		`not`	`-`
logic program		$\leftarrow$	,	;		$\sim$	$\neg$
formula	$\bot, \top$	$\rightarrow$	$\wedge$	$\vee$	$\leftrightarrow$	$\sim$	$\neg$

Outline

Problem solving in ASP: Semantics



Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Some "logical" remarks

- Positive rules are also referred to as definite clauses
  - Definite clauses are disjunctions with exactly one positive atom:

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- Horn clauses are clauses with at most one positive atom
  - Every definite clause is a Horn clause but not vice versa
  - Non-definite Horn clauses can be regarded as integrity constraints
  - A set of Horn clauses has a smallest model or none
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ corresponds to the smallest model of the set of definite clauses corresponding to P

Some "logical" remarks

- Positive rules are also referred to as definite clauses
  - Definite clauses are disjunctions with exactly one positive atom:

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- Horn clauses are clauses with at most one positive atom
  - Every definite clause is a Horn clause but not vice versa
  - Non-definite Horn clauses can be regarded as integrity constraints
  - A set of Horn clauses has a smallest model or none
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ corresponds to the smallest model of the set of definite clauses corresponding to P

Some "logical" remarks

- Positive rules are also referred to as definite clauses
  - Definite clauses are disjunctions with exactly one positive atom:

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- Horn clauses are clauses with at most one positive atom
  - Every definite clause is a Horn clause but not vice versa
  - Non-definite Horn clauses can be regarded as integrity constraints
  - A set of Horn clauses has a smallest model or none
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ corresponds to the smallest model of the set of definite clauses corresponding to P

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$P_\Phi \quad \begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}$$

$$\{p, q\}$$

Informally, a set X of atoms is a stable model of a logic program P

if X is a (classical) model of P and

if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad \boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$$

$$\{p, q\}$$

Informally, a set X of atoms is a stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\boxed{q \wedge (q \wedge \neg r \rightarrow p)}$

$\{p, q\}, \{q, r\},$ and $\{p, q, r\}$

Formula Φ has one stable model, often called answer set:

$\{p, q\}$

p	$\mapsto$	1
q	$\mapsto$	1
r	$\mapsto$	0

P_Φ

$q \leftarrow$

$p \leftarrow q, \sim r$

Informally, a set X of atoms is a stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad \boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$$

$$\{p, q\}$$

Informally, a set X of atoms is a stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Φ $\quad q \wedge (q \wedge \neg r \rightarrow p)$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

P_Φ

$$q \leftarrow$$
$$p \leftarrow q, \sim r$$

Informally, a set X of atoms is a stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad \boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$$

$$\{p, q\}$$

Informally, a set X of atoms is a stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad \boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}}$$

$$\{p, q\}$$

Informally, a set X of atoms is a stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Basic idea

Consider the logical formula Φ and its three (classical) models:

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula Φ has one stable model, often called answer set:

$$P_\Phi \quad \begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}$$

$$\{p, q\}$$

Informally, a set X of atoms is a stable model of a logic program P

- if X is a (classical) model of P and
- if all atoms in X are justified by some rule in P

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Formal Definition

Stable model of normal programs

- The reduct, P^X, of a program P relative to a set X of atoms is defined by

$$P^X = \{head(r) \leftarrow body(r)^+ \mid r \in P \text{ and } body(r)^- \cap X = \emptyset\}$$

- A set X of atoms is a stable model of a program P, if $Cn(P^X) = X$

- Note: $Cn(P^X)$ is the $\subseteq$–smallest (classical) model of P^X
- Note: Every atom in X is justified by an *"applying rule from P"*

Formal Definition

Stable model of normal programs

- The reduct, P^X, of a program P relative to a set X of atoms is defined by

$$P^X = \{head(r) \leftarrow body(r)^+ \mid r \in P \text{ and } body(r)^- \cap X = \emptyset\}$$

- A set X of atoms is a stable model of a program P, if $Cn(P^X) = X$

- Note: $Cn(P^X)$ is the $\subseteq$–smallest (classical) model of P^X
- Note: Every atom in X is justified by an *"applying rule from P"*

Formal Definition

Stable model of normal programs

- The reduct, P^X, of a program P relative to a set X of atoms is defined by

$$P^X = \{head(r) \leftarrow body(r)^+ \mid r \in P \text{ and } body(r)^- \cap X = \emptyset\}$$

- A set X of atoms is a stable model of a program P, if $Cn(P^X) = X$

- Note: $Cn(P^X)$ is the $\subseteq$–smallest (classical) model of P^X
- Note: Every atom in X is justified by an *"applying rule from P"*

A closer look at P^X

- In other words, given a set X of atoms from P,

  P^X is obtained from P by deleting
  1. each rule having $\sim a$ in its body with $a \in X$
     and then
  2. all negative atoms of the form $\sim a$
     in the bodies of the remaining rules

- Note: Only negative body literals are evaluated wrt X

A closer look at P^X

- In other words, given a set X of atoms from P,

  P^X is obtained from P by deleting
  1. each rule having $\sim a$ in its body with $a \in X$
     and then
  2. all negative atoms of the form $\sim a$
     in the bodies of the remaining rules

- Note: Only negative body literals are evaluated wrt X

Outline

A first example

$$P = \{p \leftarrow p,\ q \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p\}$	$p \leftarrow p$	$\emptyset$
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$

A first example

$$P = \{p \leftarrow p,\ \ q \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$$P = \{p \leftarrow p,\ \ q \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\}$	$p \leftarrow p$	$\emptyset$	✗
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✓
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✗

A first example

$$P = \{p \leftarrow p,\ \ q \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$$P = \{p \leftarrow p,\ \ q \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$$P = \{p \leftarrow p,\ \ q \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A first example

$P = \{p \leftarrow p,\ \ q \leftarrow \sim p\}$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A second example

$$P = \{p \leftarrow \sim q,\ q \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$
$\{p\}$	$p \leftarrow$	$\{p\}$
$\{q\}$	$q \leftarrow$	$\{q\}$
$\{p, q\}$		$\emptyset$

A second example

$P = \{p \leftarrow \sim q,\ q \leftarrow \sim p\}$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✗
$\{p\}$	$p \leftarrow$	$\{p\}$	✓
$\{q\}$	$q \leftarrow$	$\{q\}$	✓
$\{p, q\}$		$\emptyset$	✗

A second example

$P = \{p \leftarrow \sim q,\ q \leftarrow \sim p\}$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$P = \{p \leftarrow \sim q,\ q \leftarrow \sim p\}$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$P = \{p \leftarrow \sim q,\ q \leftarrow \sim p\}$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A second example

$P = \{p \leftarrow \sim q,\ q \leftarrow \sim p\}$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A third example

$$P = \{p \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$
$\emptyset$	$p \leftarrow$	$\{p\}$
$\{p\}$		$\emptyset$

A third example

$$P = \{p \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✗
$\{p\}$		$\emptyset$	✗

A third example

$$P = \{p \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

A third example

$$P = \{p \leftarrow \sim p\}$$

X	P^X	$Cn(P^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

Some properties

- A logic program may have zero, one, or multiple stable models!
- If X is an stable model of a logic program P, then X is a model of P (seen as a formula)
- If X and Y are stable models of a *normal* program P, then $X \not\subset Y$

Some properties

- A logic program may have zero, one, or multiple stable models!
- If X is an stable model of a logic program P, then X is a model of P (seen as a formula)
- If X and Y are stable models of a *normal* program P, then $X \not\subset Y$

Outline

Programs with Variables

Let P be a logic program

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- Ground Instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}, var(r\theta) = \emptyset\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution

- Ground Instantiation of P: $ground(P) = \bigcup_{r \in P} ground(r)$

Programs with Variables

Let P be a logic program

- Let $\mathcal{T}$ be a set of variable-free terms (also called Herbrand universe)
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$ (also called alphabet or Herbrand base)
- Ground Instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}, var(r\theta) = \emptyset\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution

- Ground Instantiation of P: $ground(P) = \bigcup_{r \in P} ground(r)$

Programs with Variables

Let P be a logic program

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- Ground Instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}, var(r\theta) = \emptyset\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution

- Ground Instantiation of P: $ground(P) = \bigcup_{r \in P} ground(r)$

Programs with Variables

Let P be a logic program

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- Ground Instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \to \mathcal{T}, var(r\theta) = \emptyset\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution

- Ground Instantiation of P: $ground(P) = \bigcup_{r \in P} ground(r)$

An example

$$P = \{\ r(a,b) \leftarrow,\ r(b,c) \leftarrow,\ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$\mathcal{T} = \{a, b, c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{l} r(a,b) \leftarrow\ , \\ r(b,c) \leftarrow\ , \\ t(a,a) \leftarrow r(a,a),\ t(b,a) \leftarrow r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b),\ t(b,b) \leftarrow r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c),\ t(b,c) \leftarrow r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

Intelligent Grounding aims at reducing the ground instantiation

An example

$$P = \{\ r(a,b) \leftarrow,\ r(b,c) \leftarrow,\ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$\mathcal{T} = \{a, b, c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{llllllll} r(a,b) & \leftarrow & , & & & & & \\ r(b,c) & \leftarrow & , & & & & & \\ t(a,a) & \leftarrow & r(a,a), & t(b,a) & \leftarrow & r(b,a), & t(c,a) & \leftarrow\ r(c,a), \\ t(a,b) & \leftarrow & r(a,b), & t(b,b) & \leftarrow & r(b,b), & t(c,b) & \leftarrow\ r(c,b), \\ t(a,c) & \leftarrow & r(a,c), & t(b,c) & \leftarrow & r(b,c), & t(c,c) & \leftarrow\ r(c,c) \end{array} \right\}$$

- Intelligent Grounding aims at reducing the ground instantiation

An example

$$P = \{\ r(a,b) \leftarrow,\ r(b,c) \leftarrow,\ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$\mathcal{T} = \{a, b, c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{l} r(a,b) \leftarrow\ , \\ r(b,c) \leftarrow\ , \\ t(a,a) \leftarrow r(a,a),\ t(b,a) \leftarrow r(b,a),\ t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b),\ t(b,b) \leftarrow r(b,b),\ t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c),\ t(b,c) \leftarrow r(b,c),\ t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

- Intelligent Grounding aims at reducing the ground instantiation

Stable models of programs with Variables

Let P be a normal logic program with variables

- A set X of (ground) atoms as a stable model of P,
  if $Cn(ground(P)^X) = X$

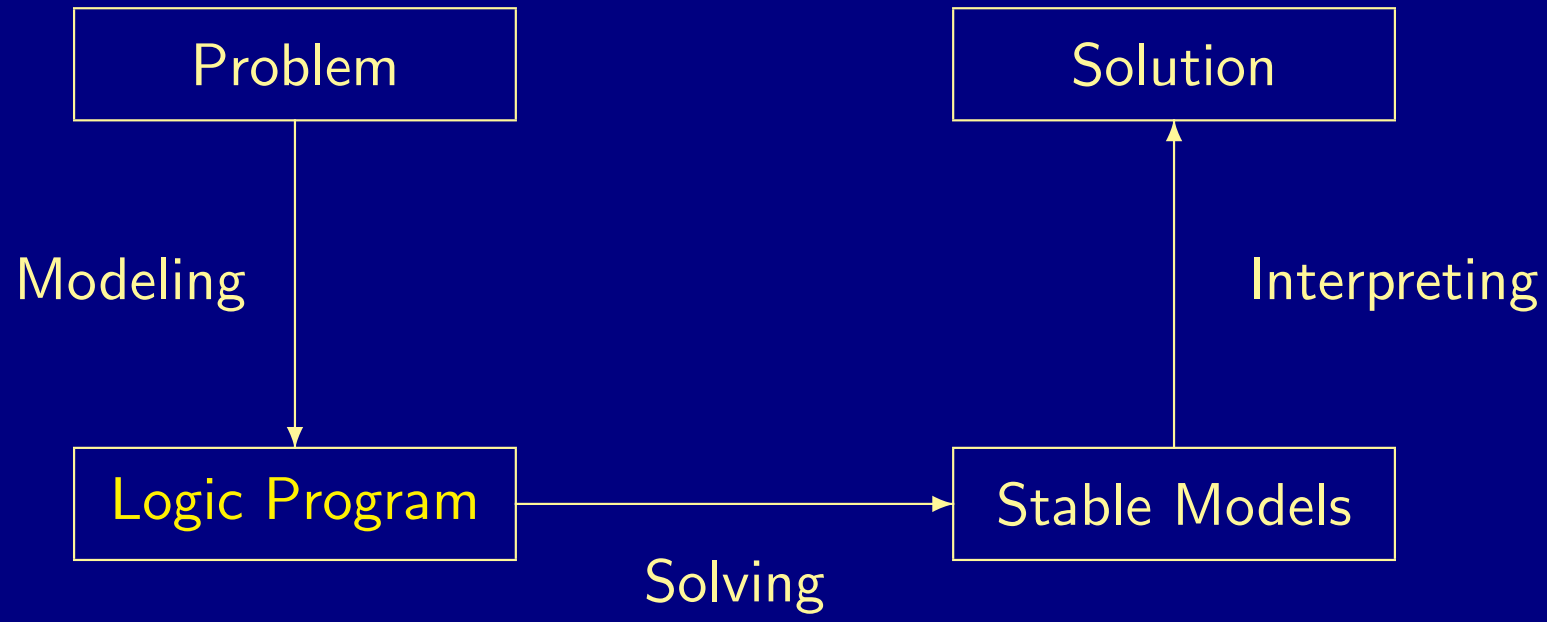
Stable models of programs with Variables

Let P be a normal logic program with variables

- A set X of (ground) atoms as a stable model of P,
  if $Cn(ground(P)^X) = X$

Outline

Problem solving in ASP: Extended Syntax



Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- **Disjunction**
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

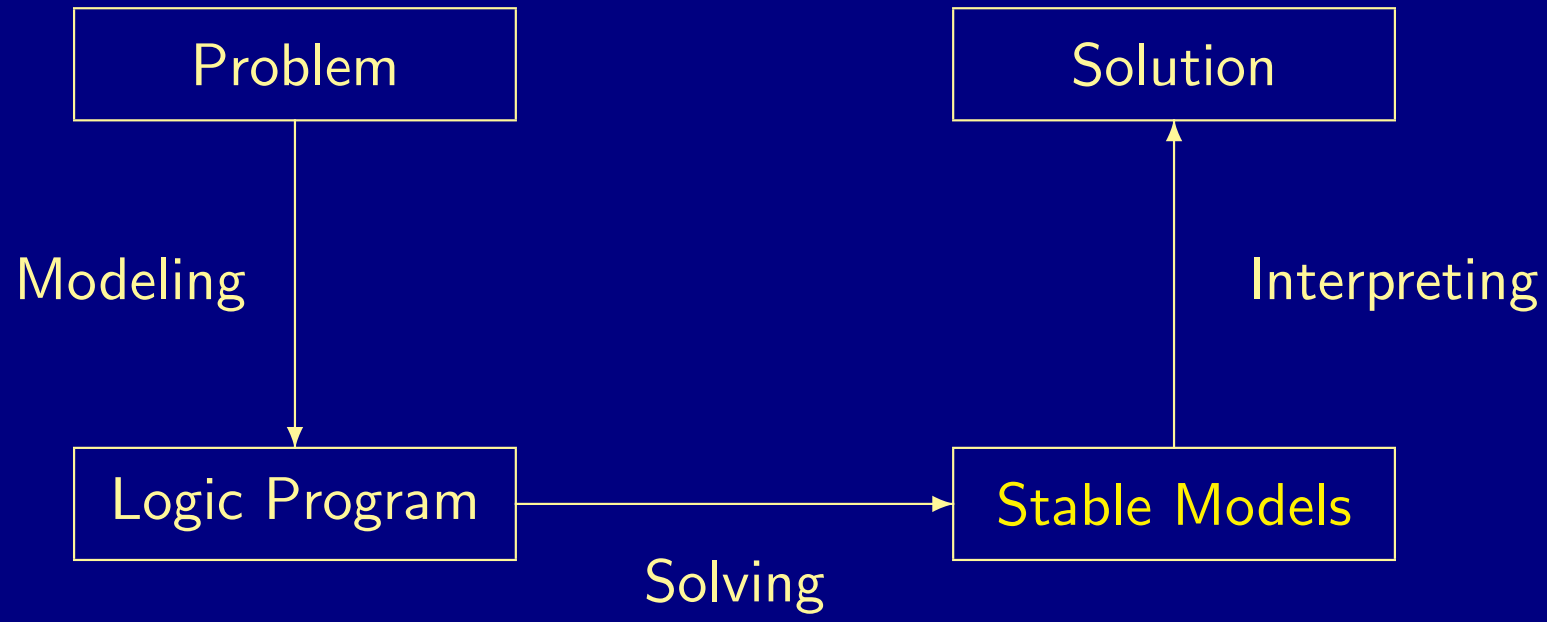
- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{$`a, b, c`$\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) | q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Outline

Problem solving in ASP: Reasoning Modes



Reasoning Modes

- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization
- and combinations of them

† without solution recording

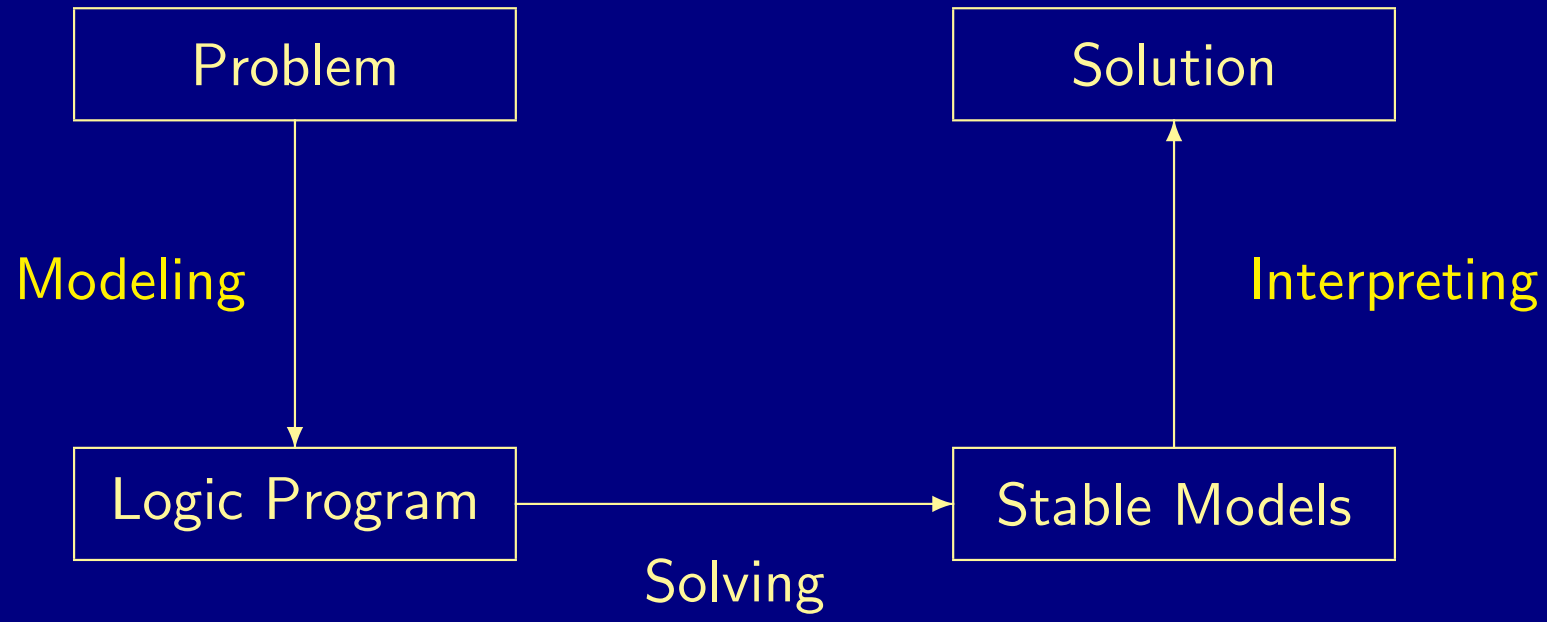
‡ without solution enumeration

Basic Modeling: Overview

13 ASP solving process
- Graph coloring

14 Methodology
- Satisfiability
- Queens
- Traveling Salesperson
- Reviewer Assignment
- Planning

Modeling and Interpreting



Modeling

- For solving a problem class **C** for a problem instance **I**, encode
  1. the problem instance **I** as a set $P_{\mathbf{I}}$ of facts and
  2. the problem class **C** as a set $P_{\mathbf{C}}$ of rules

  such that the solutions to **C** for **I** can be (polynomially) extracted from the stable models of $P_{\mathbf{I}} \cup P_{\mathbf{C}}$
- $P_{\mathbf{I}}$ is (still) called problem instance
- $P_{\mathbf{C}}$ is often called the problem encoding
- An encoding $P_{\mathbf{C}}$ is uniform, if it can be used to solve all its problem instances
  That is, $P_{\mathbf{C}}$ encodes the solutions to **C** for any set $P_{\mathbf{I}}$ of facts

Modeling

- For solving a problem class **C** for a problem instance **I**, encode
  1. the problem instance **I** as a set $P_{\mathbf{I}}$ of facts and
  2. the problem class **C** as a set $P_{\mathbf{C}}$ of rules

  such that the solutions to **C** for **I** can be (polynomially) extracted from the stable models of $P_{\mathbf{I}} \cup P_{\mathbf{C}}$
- $P_{\mathbf{I}}$ is (still) called problem instance
- $P_{\mathbf{C}}$ is often called the problem encoding
- An encoding $P_{\mathbf{C}}$ is uniform, if it can be used to solve all its problem instances
  That is, $P_{\mathbf{C}}$ encodes the solutions to **C** for any set $P_{\mathbf{I}}$ of facts

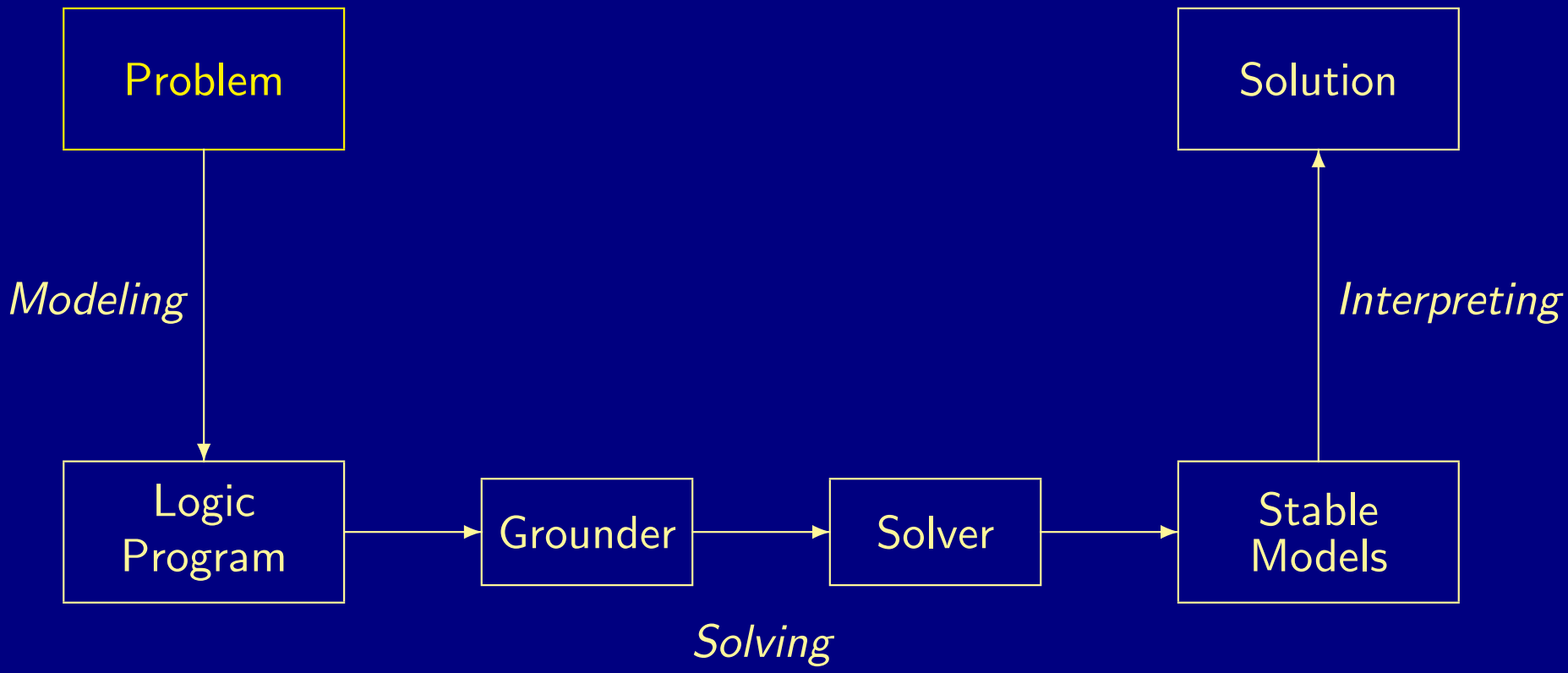
Modeling

- For solving a problem class **C** for a problem instance **I**, encode
  1. the problem instance **I** as a set $P_{\mathbf{I}}$ of facts and
  2. the problem class **C** as a set $P_{\mathbf{C}}$ of rules

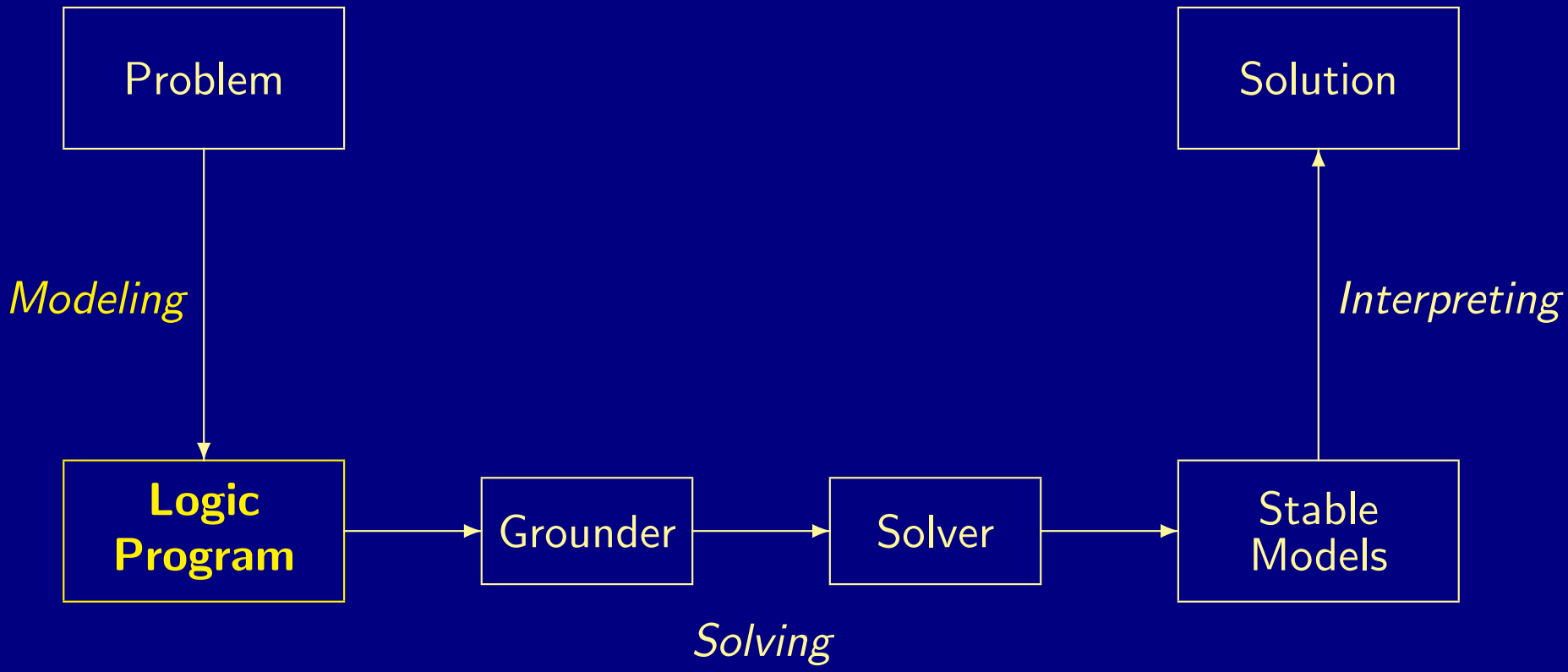
  such that the solutions to **C** for **I** can be (polynomially) extracted from the stable models of $P_{\mathbf{I}} \cup P_{\mathbf{C}}$
- $P_{\mathbf{I}}$ is (still) called problem instance
- $P_{\mathbf{C}}$ is often called the problem encoding
- An encoding $P_{\mathbf{C}}$ is uniform, if it can be used to solve all its problem instances
  That is, $P_{\mathbf{C}}$ encodes the solutions to **C** for any set $P_{\mathbf{I}}$ of facts

Outline

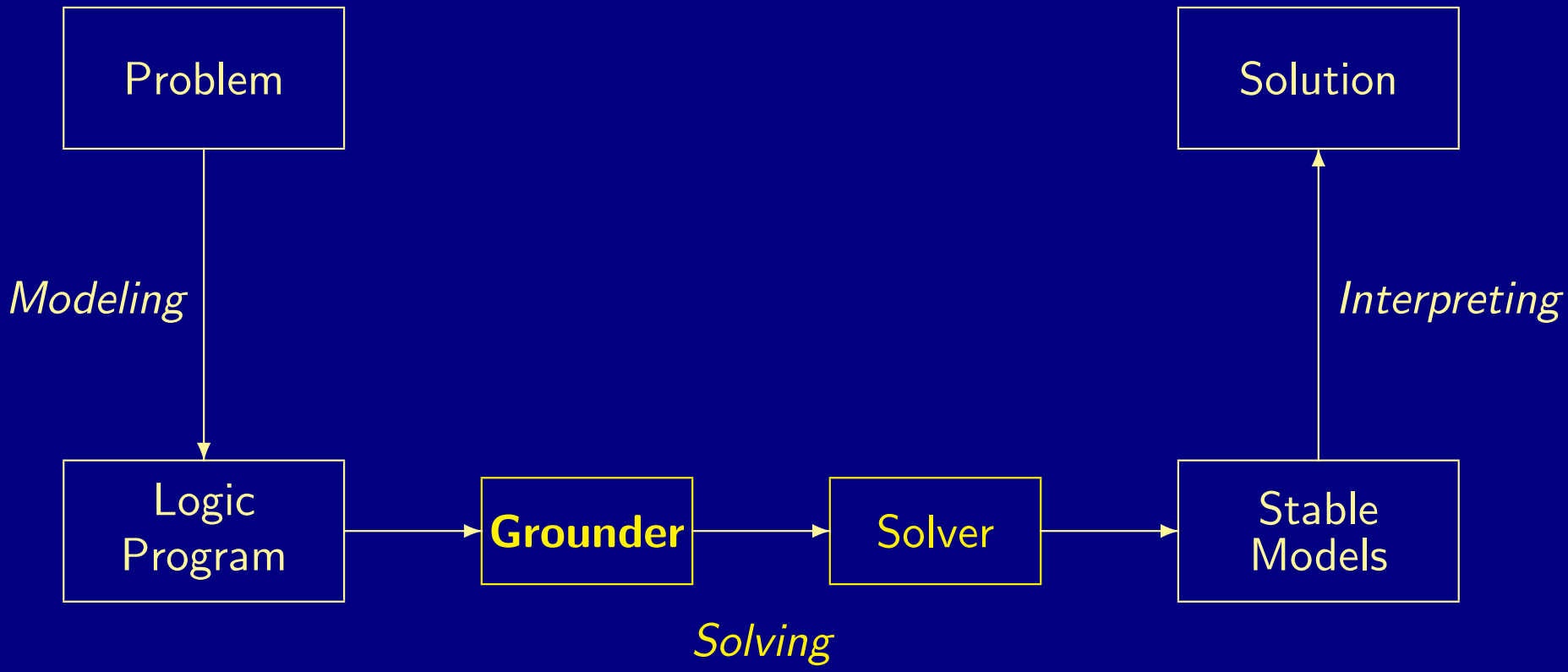
ASP solving process



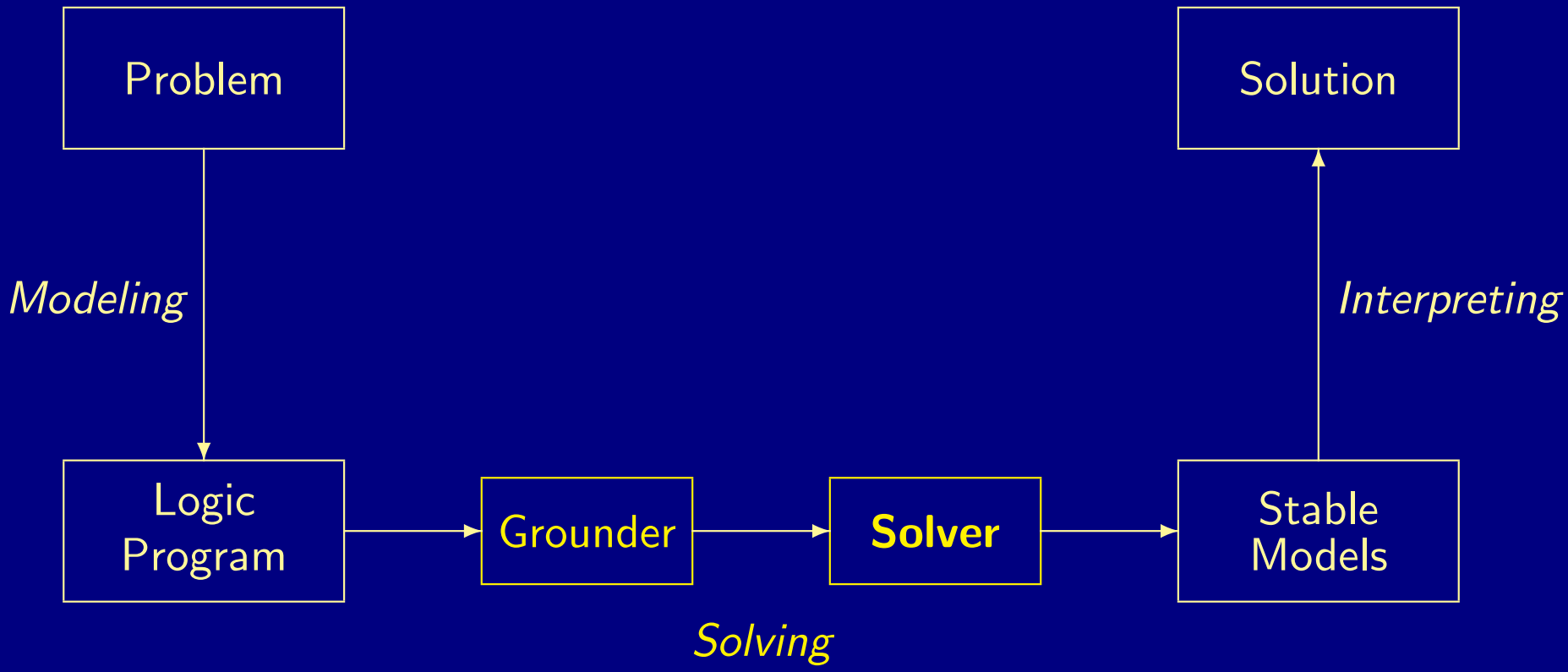
ASP solving process



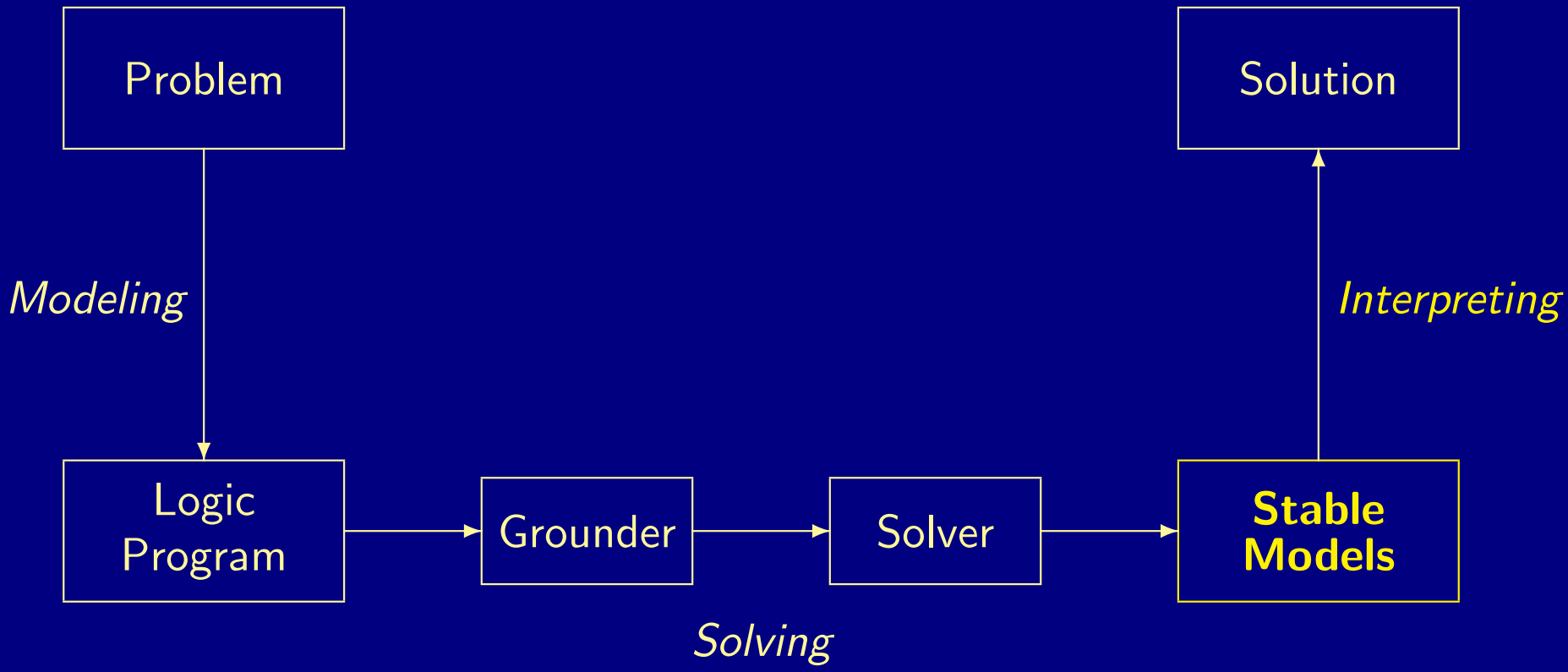
ASP solving process



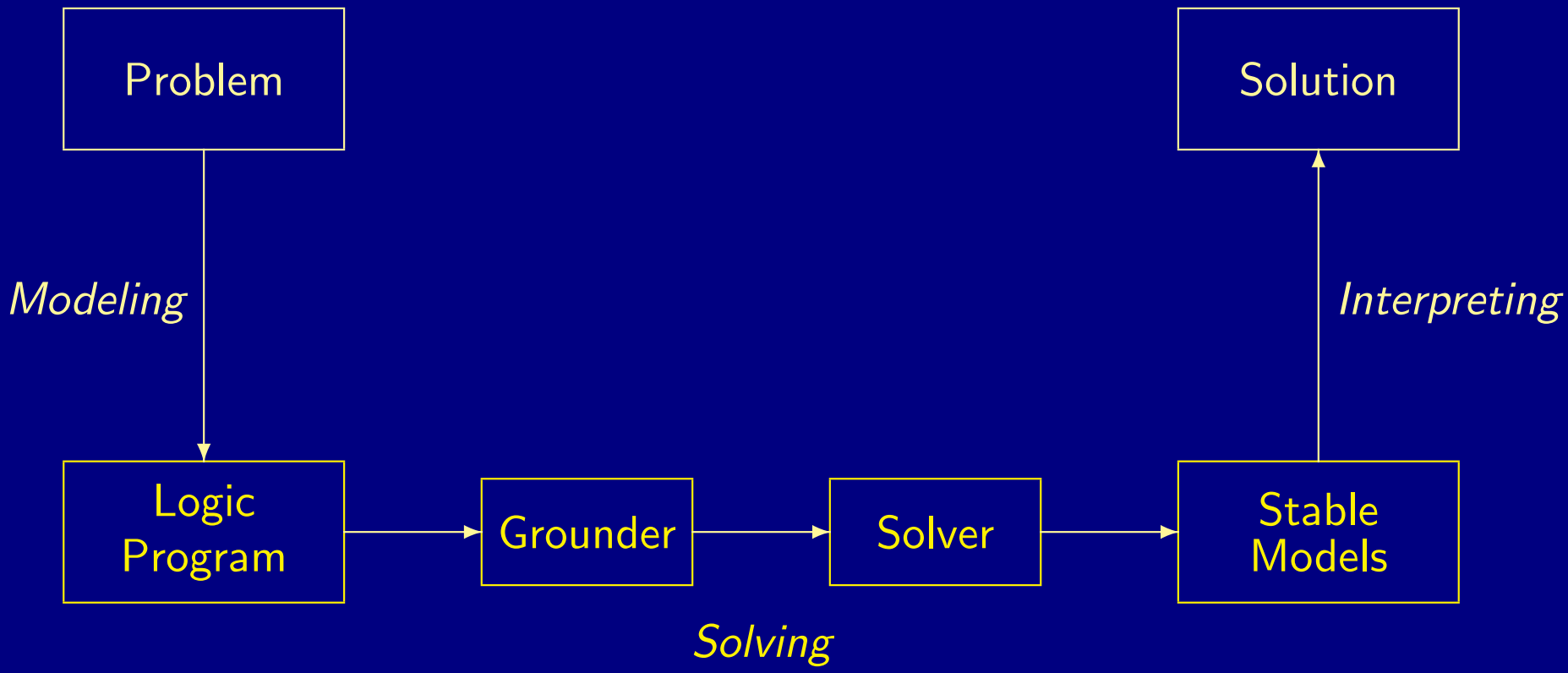
ASP solving process



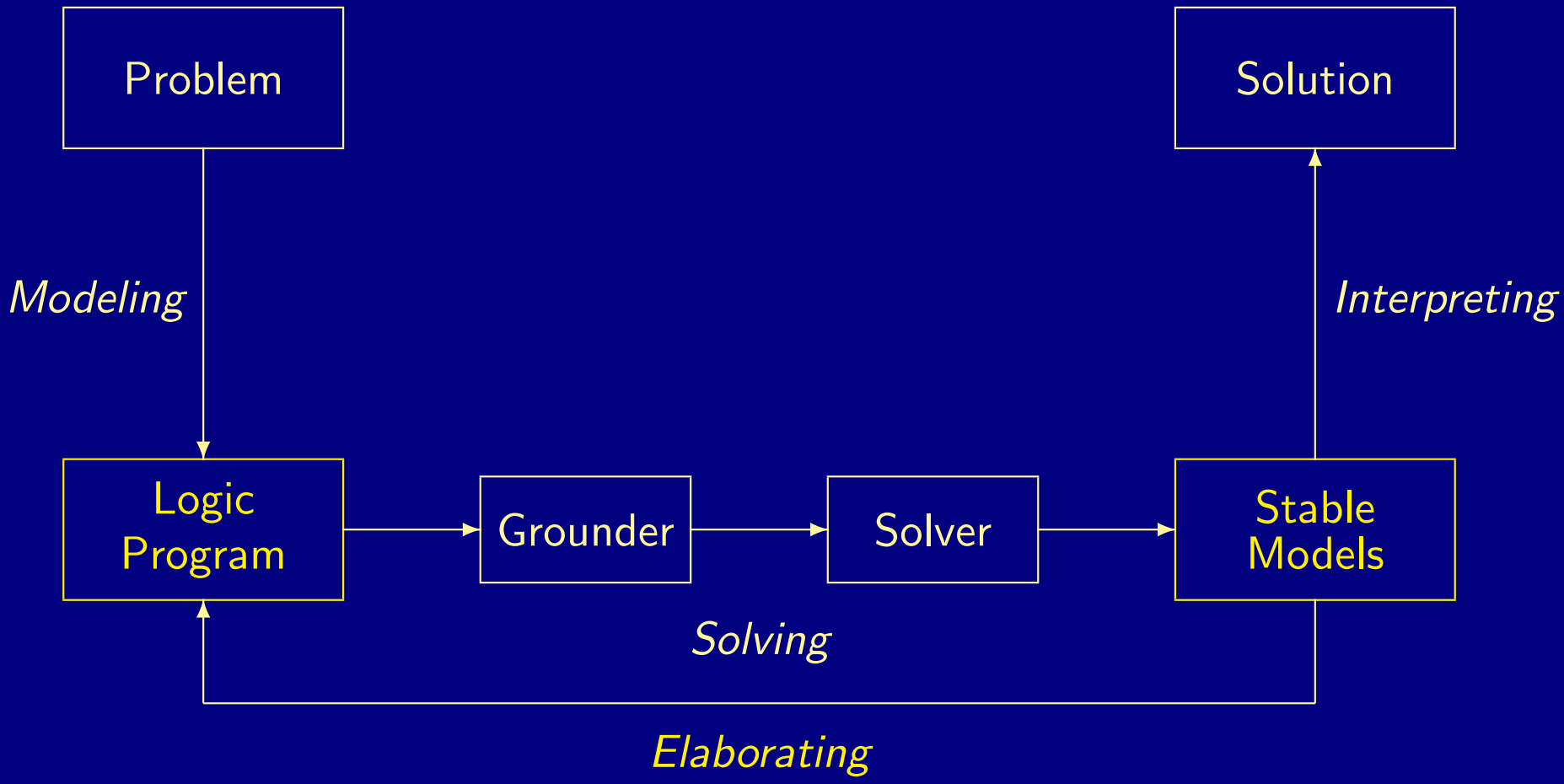
ASP solving process



ASP solving process

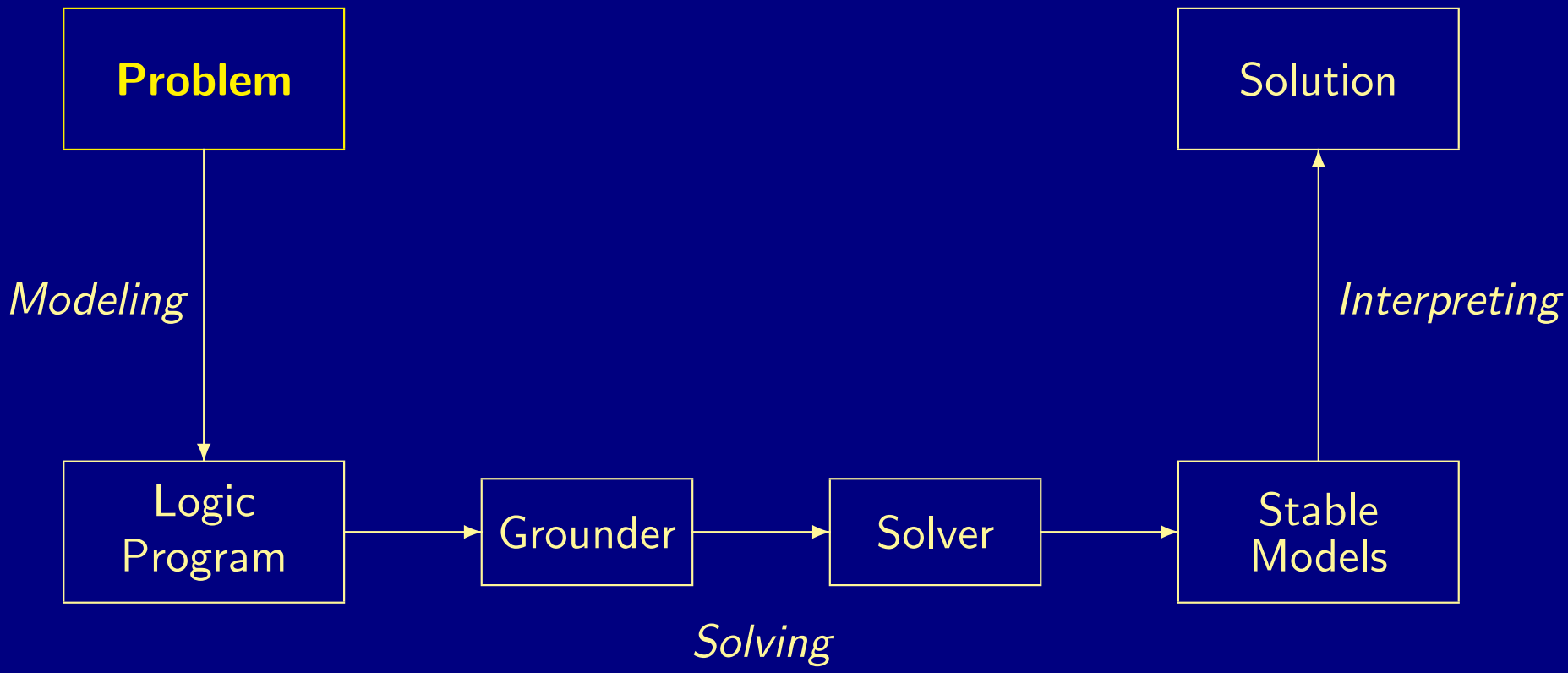


ASP solving process



Outline

ASP solving process



Graph coloring

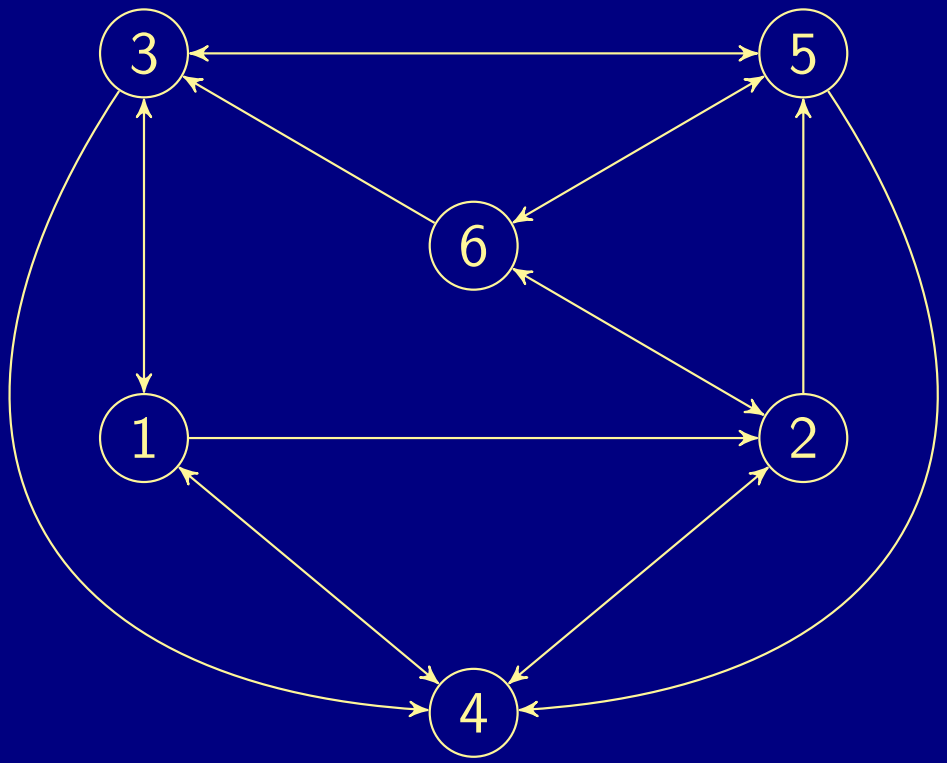
- Problem instance A graph consisting of nodes and edges

Graph coloring

- Problem instance A graph consisting of nodes and edges

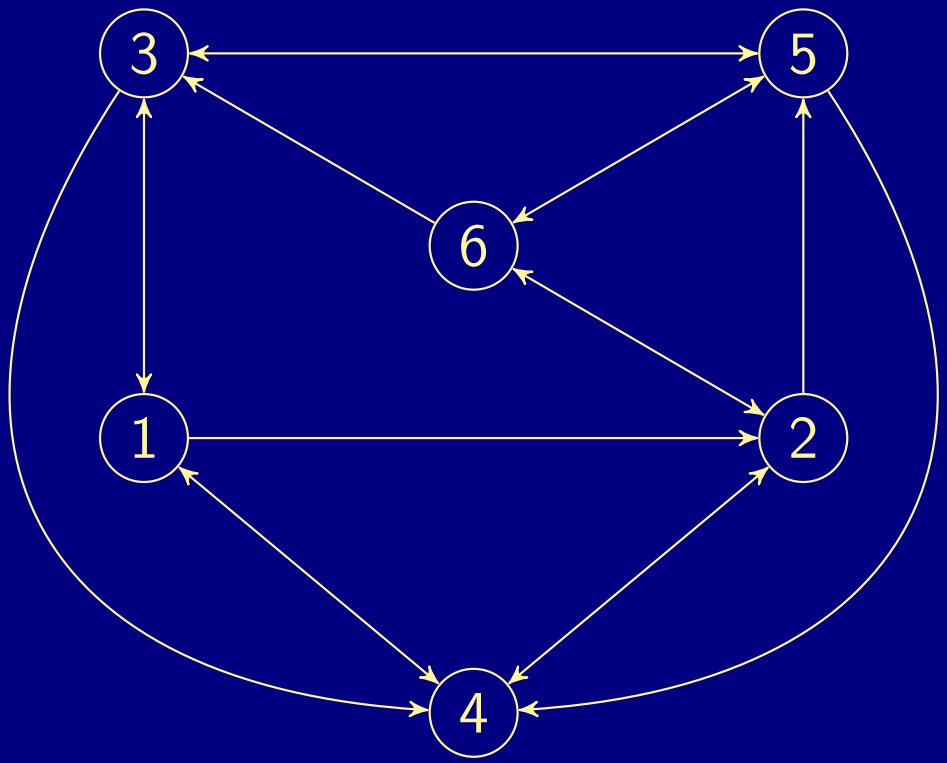
Graph coloring

- Problem instance A graph consisting of nodes and edges



Graph coloring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`



Graph coloring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `color/1`

Graph coloring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `color/1`
- Problem class Assign each node one color such that no two nodes connected by an edge have the same color

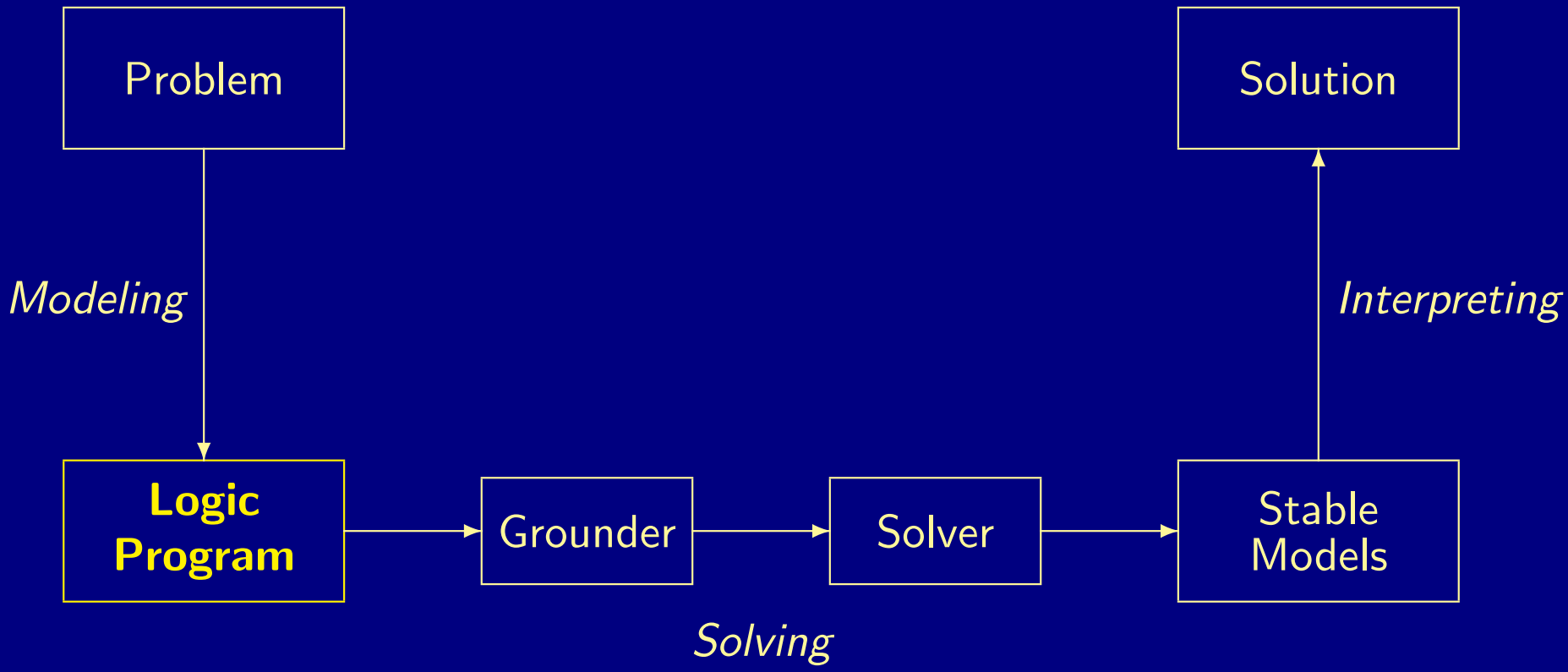
Graph coloring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `color/1`
- Problem class Assign each node one color such that no two nodes connected by an edge have the same color

  In other words,

  1. Each node has a unique color
  2. Two connected nodes must not have the same color

ASP solving process



Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Graph coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

color.lp

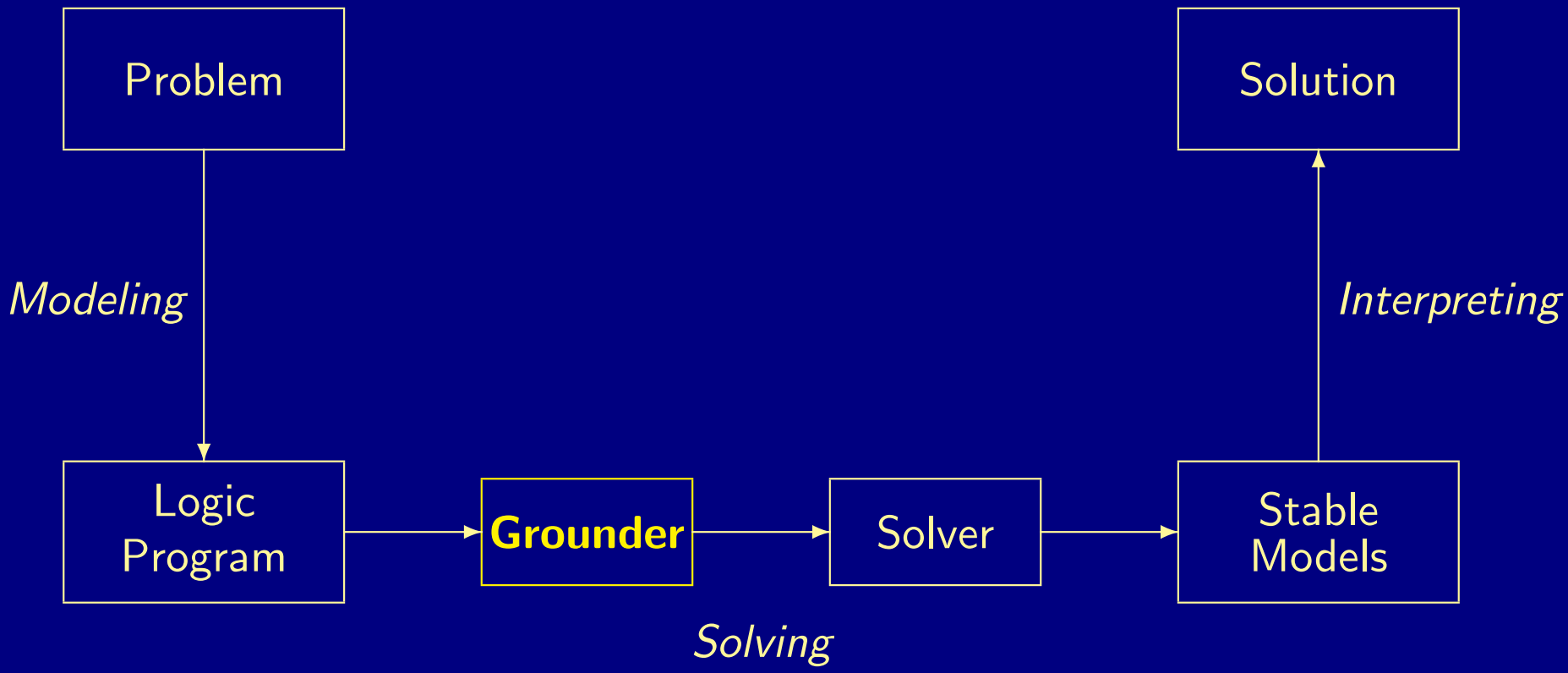
```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

ASP solving process



Graph coloring: Grounding

```
$ gringo --text color.lp
```

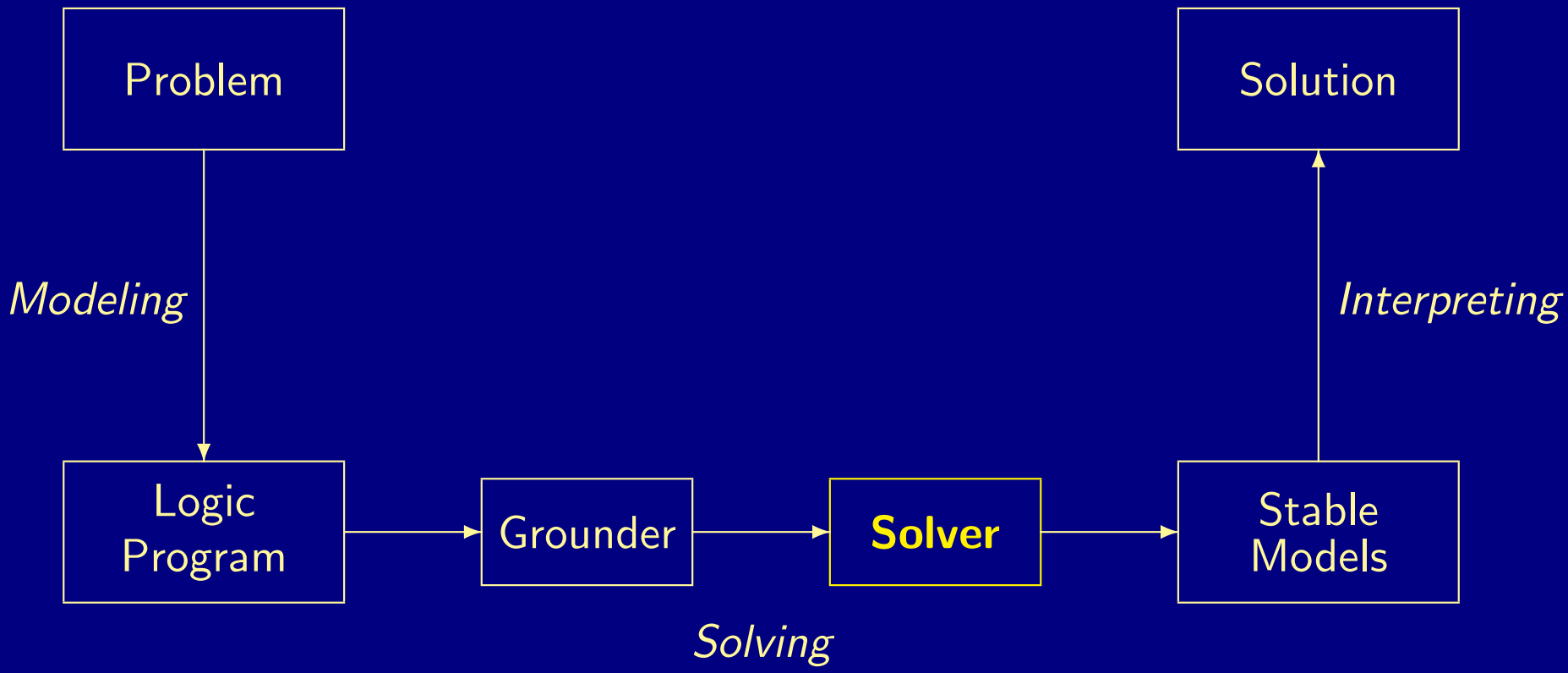
```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(1,3).  edge(1,4).  edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).  edge(4,1).  edge(4,2).  edge(5,3).
edge(5,4).  edge(5,6).  edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

 :- color(1,r), color(2,r).  :- color(2,g), color(5,g).  ...  :- color(6,r), color(2,r).
 :- color(1,b), color(2,b).  :- color(2,r), color(6,r).       :- color(6,b), color(2,b).
 :- color(1,g), color(2,g).  :- color(2,b), color(6,b).       :- color(6,g), color(2,g).
 :- color(1,r), color(3,r).  :- color(2,g), color(6,g).       :- color(6,r), color(3,r).
 :- color(1,b), color(3,b).  :- color(3,r), color(1,r).       :- color(6,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).       :- color(6,g), color(3,g).
 :- color(1,r), color(4,r).  :- color(3,g), color(1,g).       :- color(6,r), color(5,r).
 :- color(1,b), color(4,b).  :- color(3,r), color(4,r).       :- color(6,b), color(5,b).
 :- color(1,g), color(4,g).  :- color(3,b), color(4,b).       :- color(6,g), color(5,g).
 :- color(2,r), color(4,r).  :- color(3,g), color(4,g).
 :- color(2,b), color(4,b).  :- color(3,r), color(5,r).
 :- color(2,g), color(4,g).  :- color(3,b), color(5,b).
```

Graph coloring: Grounding

```
$ gringo --text color.lp
```

```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(1,3).  edge(1,4).  edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).  edge(4,1).  edge(4,2).  edge(5,3).
edge(5,4).  edge(5,6).  edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

 :- color(1,r), color(2,r).  :- color(2,g), color(5,g).  ...  :- color(6,r), color(2,r).
 :- color(1,b), color(2,b).  :- color(2,r), color(6,r).       :- color(6,b), color(2,b).
 :- color(1,g), color(2,g).  :- color(2,b), color(6,b).       :- color(6,g), color(2,g).
 :- color(1,r), color(3,r).  :- color(2,g), color(6,g).       :- color(6,r), color(3,r).
 :- color(1,b), color(3,b).  :- color(3,r), color(1,r).       :- color(6,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).       :- color(6,g), color(3,g).
 :- color(1,r), color(4,r).  :- color(3,g), color(1,g).       :- color(6,r), color(5,r).
 :- color(1,b), color(4,b).  :- color(3,r), color(4,r).       :- color(6,b), color(5,b).
 :- color(1,g), color(4,g).  :- color(3,b), color(4,b).       :- color(6,g), color(5,g).
 :- color(2,r), color(4,r).  :- color(3,g), color(4,g).
 :- color(2,b), color(4,b).  :- color(3,r), color(5,r).
 :- color(2,g), color(4,g).  :- color(3,b), color(5,b).
```

ASP solving process



Graph coloring: Solving

```
$ gringo color.lp | clasp 0
```

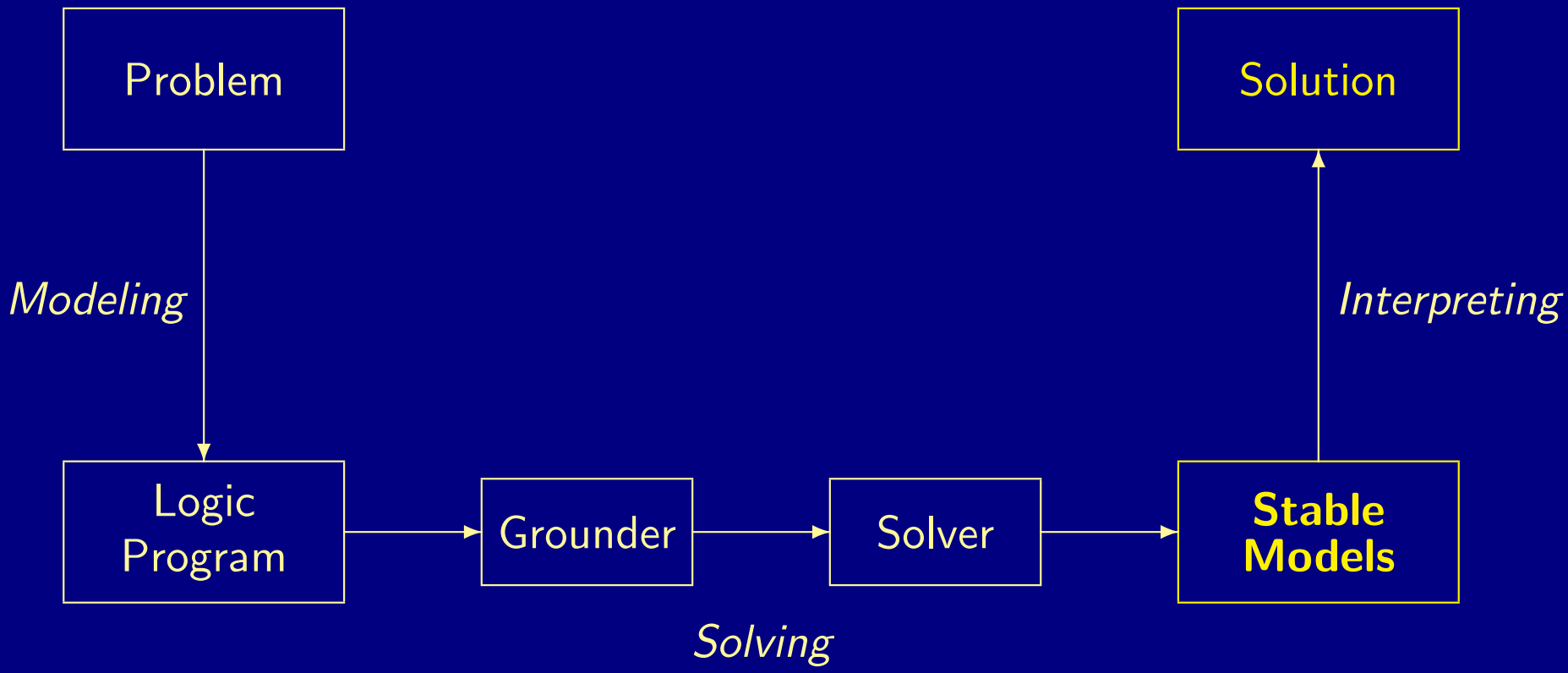
```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,b) color(4,g) color(3,r) color(2,r) color(1,b)
Answer: 4
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,b) color(4,r) color(3,g) color(2,g) color(1,b)
Answer: 5
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,r) color(4,g) color(3,b) color(2,b) color(1,r)
Answer: 6
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
SATISFIABLE

Models       : 6
Time         : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Graph coloring: Solving

```
$ gringo color.lp | clasp 0
```

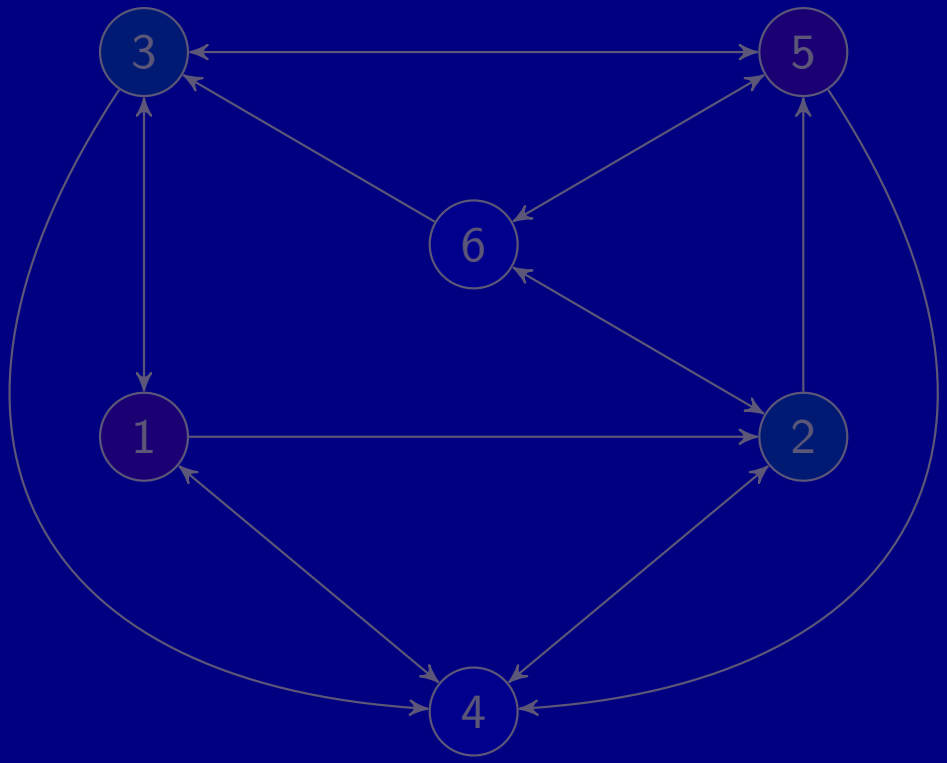
```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,b) color(4,g) color(3,r) color(2,r) color(1,b)
Answer: 4
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,b) color(4,r) color(3,g) color(2,g) color(1,b)
Answer: 5
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,r) color(4,g) color(3,b) color(2,b) color(1,r)
Answer: 6
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
SATISFIABLE

Models       : 6
Time         : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

ASP solving process



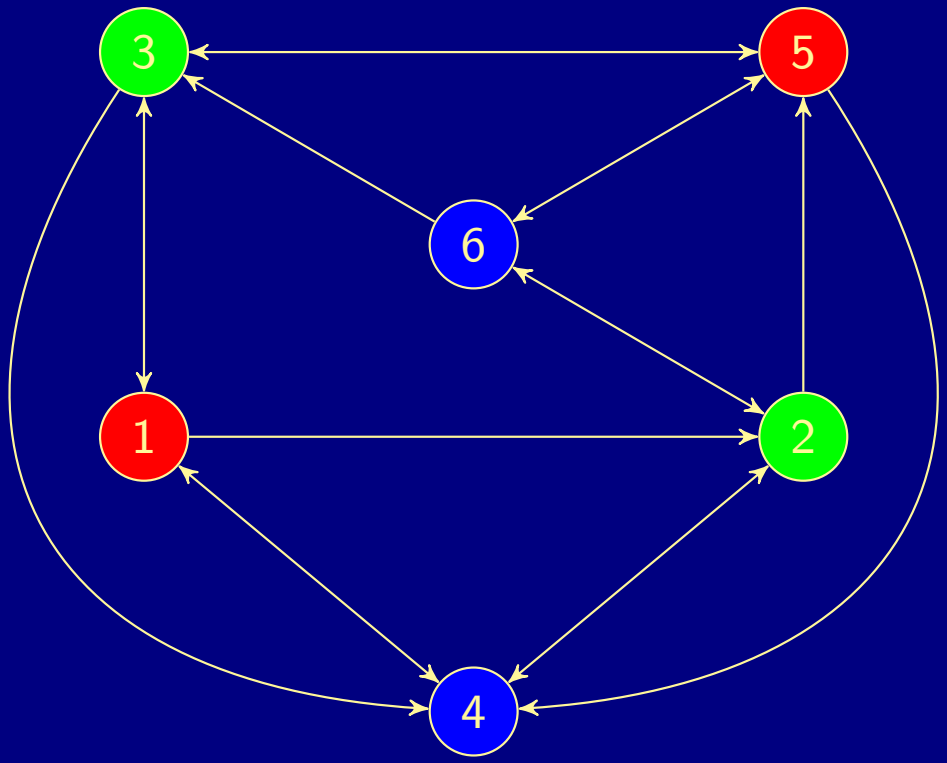
A coloring

```
Answer: 6
edge(1,2) ... col(r) ... node(1) ...                                    \
color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
```



A coloring

```
Answer: 6
edge(1,2) ... col(r) ... node(1) ...                                    \
color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
```



Outline

Basic methodology

Methodology

Generate and Test (or: Guess and Check)

- **Generator** Generate potential stable model candidates (typically through non-deterministic constructs)
- **Tester** Eliminate invalid candidates (typically through integrity constraints)

Nutshell

Logic program = Data + Generator + Tester (+ Optimizer)

Basic methodology

Methodology

Generate and Test (or: Guess and Check)

Generator	Generate potential stable model candidates (typically through non-deterministic constructs)
Tester	Eliminate invalid candidates (typically through integrity constraints)

Nutshell

Logic program = Data + Generator + Tester (+ Optimizer)

Outline

Satisfiability testing

- Problem Instance: A propositional formula ϕ in CNF
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true
- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	**Tester**	**Stable models**
$\{a, b\} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow a, \sim b$	$X_2 = \{\}$

Satisfiability testing

- Problem Instance: A propositional formula ϕ in CNF
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true

- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	**Tester**	**Stable models**
$\{a, b\} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow a, \sim b$	$X_2 = \{\}$

Satisfiability testing

- Problem Instance: A propositional formula ϕ in CNF
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true

- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	**Tester**	**Stable models**
$\{a, b\} \quad \leftarrow$	$\leftarrow \quad \sim a, b$	$X_1 \quad = \quad \{a, b\}$
	$\leftarrow \quad a, \sim b$	$X_2 \quad = \quad \{\}$

Satisfiability testing

- Problem Instance: A propositional formula ϕ in CNF
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true

- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	**Tester**	**Stable models**
$\{a, b\} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow a, \sim b$	$X_2 = \{\}$

Satisfiability testing

- Problem Instance: A propositional formula ϕ in CNF
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true

- Example: Consider formula

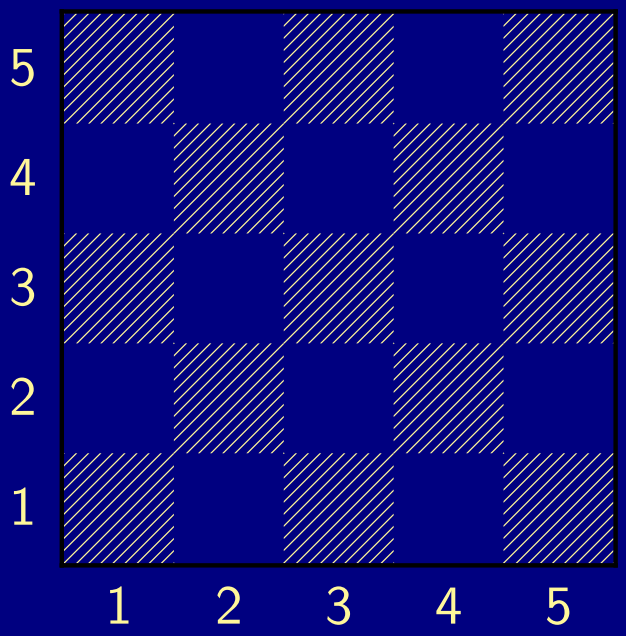
$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

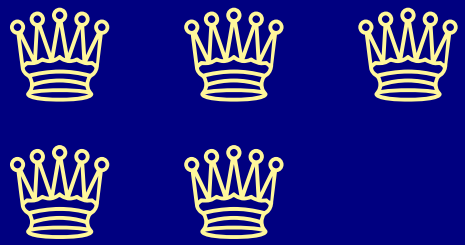
Generator	**Tester**	**Stable models**
$\{a, b\} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow a, \sim b$	$X_2 = \{\}$

Outline

The n-Queens Problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another



Defining the Field

queens.lp

```
row(1..n).
col(1..n).
```

- Create file `queens.lp`
- Define the field
  - n rows
  - n columns

Defining the Field

Running ...

```
$ clingo queens.lp --const n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
SATISFIABLE

Models       : 1
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

Placing some Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
```

- Guess a solution candidate

  by placing some queens on the board

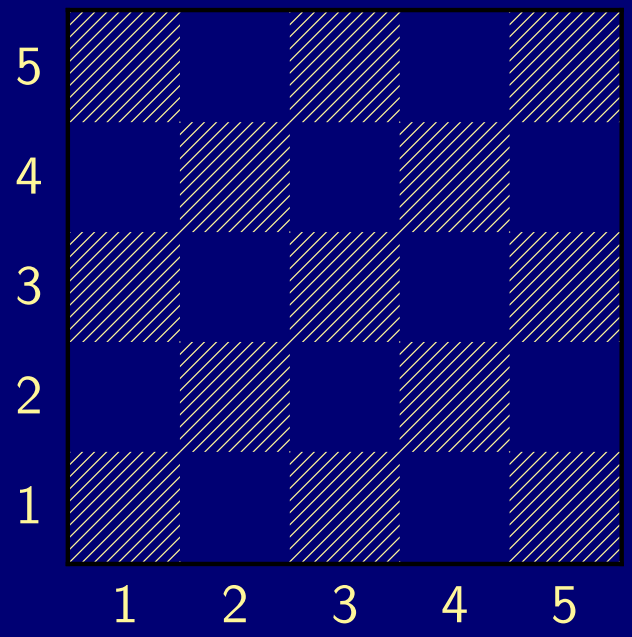
Placing some Queens

Running ...

```
$ clingo queens.lp --const n=5 3
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(1,1)
Answer: 3
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(2,1)
SATISFIABLE

Models       : 3+
```

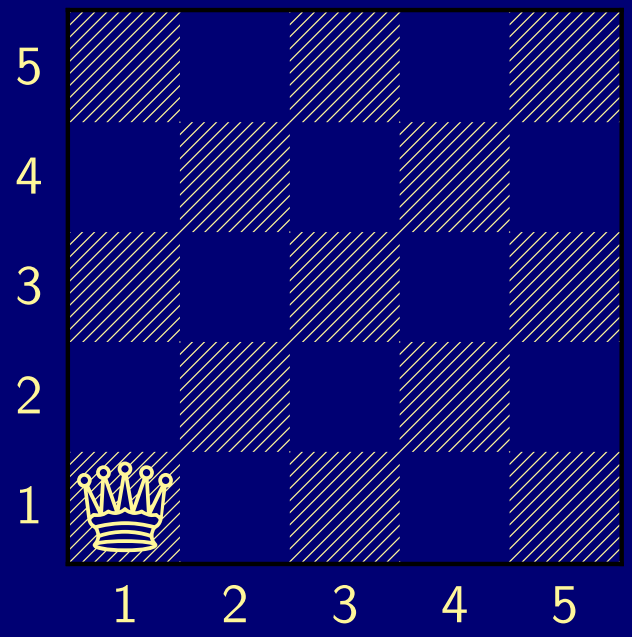
Placing some Queens: Answer 1

Answer 1



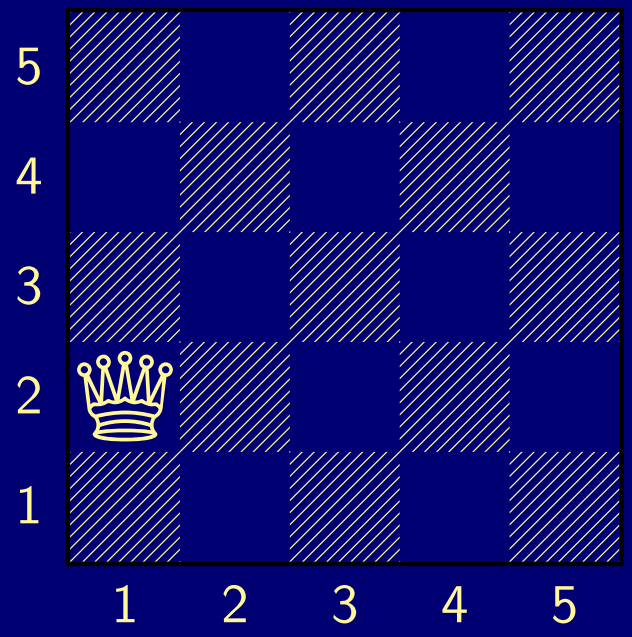
Placing some Queens: Answer 2

Answer 2



Placing some Queens: Answer 3

Answer 3



Placing n Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
```

- Place exactly n queens on the board

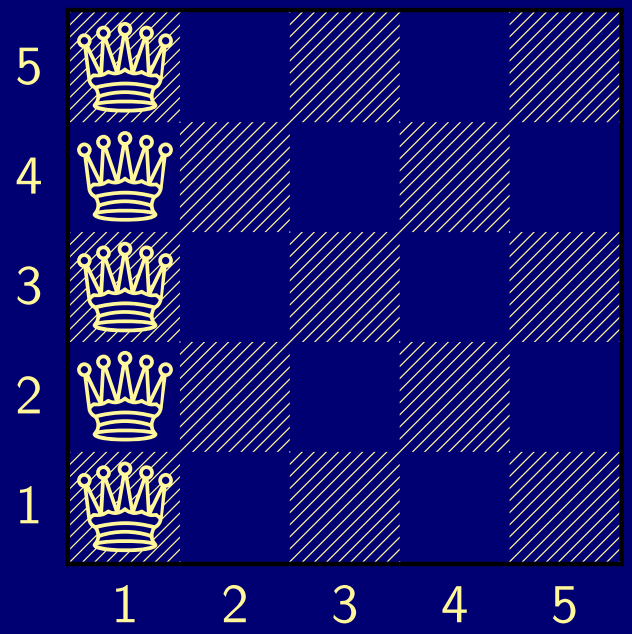
Placing n Queens

Running ...

```
$ clingo queens.lp --const n=5 2
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,1) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(1,2) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
...
```

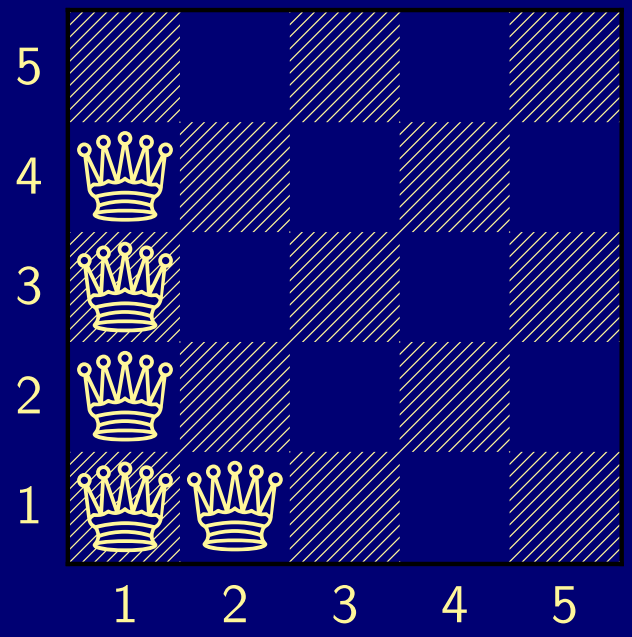
Placing n Queens: Answer 1

Answer 1



Placing n Queens: Answer 2

Answer 2



Horizontal and vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

Horizontal and vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

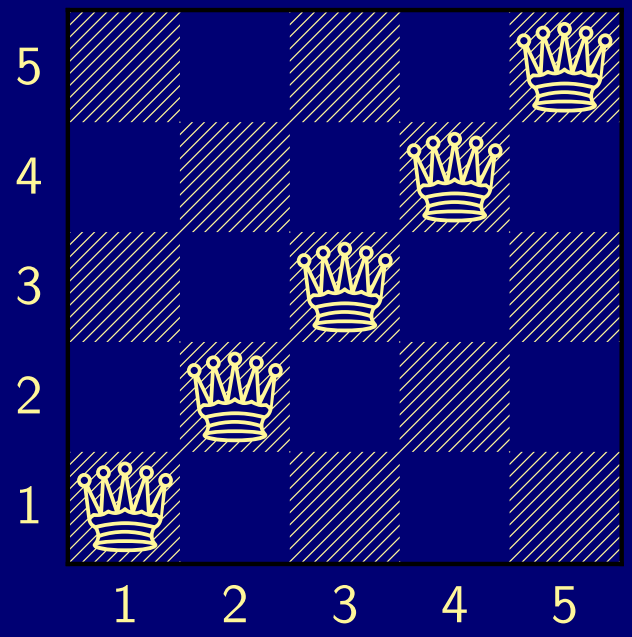
Horizontal and vertical Attack

Running ...

```
$ clingo queens.lp --const n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,5) queen(4,4) queen(3,3) \
queen(2,2) queen(1,1)
...
```

Horizontal and vertical Attack: Answer 1

Answer 1



Diagonal Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I-J == II-JJ.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I+J == II+JJ.
```

- Forbid diagonal attacks

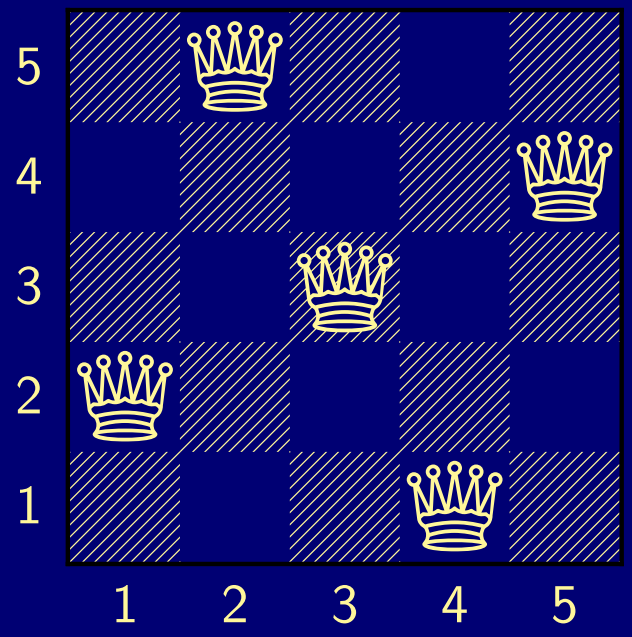
Diagonal Attack

Running ...

```
$ clingo queens.lp --const n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(4,5) queen(1,4) queen(3,3) \
queen(5,2) queen(2,1)
SATISFIABLE

Models       : 1+
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

Diagonal Attack: Answer 1

Answer 1



Optimizing

queens-opt.lp

```
1 { queen(I,1..n) } 1 :- I = 1..n.
1 { queen(1..n,J) } 1 :- J = 1..n.
 :- { queen(D-J,J) } 2, D =   2..2*n.
 :- { queen(D+J,J) } 2, D = 1-n..n-1.
```

- Encoding can be optimized
- Much faster to solve

Outline

Traveling Salesperson

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Traveling Salesperson

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Traveling Salesperson

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) = C : cost(X,Y,C) }.
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) = C : cost(X,Y,C) }.
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) = C : cost(X,Y,C) }.
```

Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { cycle(X,Y) = C : cost(X,Y,C) }.
```

Outline

Reviewer Assignment
by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment
by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment
by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Outline

Simplistic STRIPS Planning

```
time(1..k).    lasttime(T) :- time(T), not time(T+1).

fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Simplistic STRIPS Planning

```
time(1..k).    lasttime(T) :- time(T), not time(T+1).

fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Simplistic STRIPS Planning

```
time(1..k).    lasttime(T) :- time(T), not time(T+1).

fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Simplistic STRIPS Planning

```
time(1..k).    lasttime(T) :- time(T), not time(T+1).

fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Language Extensions: Overview

15 Motivation

16 Integrity constraint

17 Choice rule

18 Cardinality rule

19 Weight rule

20 Conditional literal

21 Optimization statement

22 smodels format

Outline

Language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Outline

Integrity constraint

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

where $1 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$

- Example `:- edge(3,7), color(3,red), color(7,red).`
- Embedding The above integrity constraint can be turned into the normal rule

$$x \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n, \sim x$$

where x is a new symbol, that is, $x \notin \mathcal{A}$.

- Another example $P = \{a \leftarrow not\ b,\ b \leftarrow not\ a\}$ versus $P' = P \cup \{\leftarrow a\}$ and $P'' = P \cup \{\leftarrow not\ a\}$

Integrity constraint

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

where $1 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$

- Example `:- edge(3,7), color(3,red), color(7,red).`
- Embedding The above integrity constraint can be turned into the normal rule

$$x \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n, \sim x$$

where x is a new symbol, that is, $x \notin \mathcal{A}$.

- Another example $P = \{a \leftarrow not\ b,\ b \leftarrow not\ a\}$ versus $P' = P \cup \{\leftarrow a\}$ and $P'' = P \cup \{\leftarrow not\ a\}$

Integrity constraint

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

  where $1 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$
- Example `:- edge(3,7), color(3,red), color(7,red).`
- Embedding The above integrity constraint can be turned into the normal rule

$$x \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n, \sim x$$

  where x is a new symbol, that is, $x \notin \mathcal{A}$.
- Another example $P = \{a \leftarrow not\ b,\ b \leftarrow not\ a\}$
  versus $P' = P \cup \{\leftarrow a\}$ and $P'' = P \cup \{\leftarrow not\ a\}$

Outline

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$

- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{ \{a\} \leftarrow b,\ b \leftarrow \}$ has two stable models: $\{b\}$ and $\{a, b\}$

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$
- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{ \{a\} \leftarrow b,\ b \leftarrow \}$ has two stable models: $\{b\}$ and $\{a, b\}$

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$
- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{ \{a\} \leftarrow b,\ b \leftarrow \}$ has two stable models: $\{b\}$ and $\{a, b\}$

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$
- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{ \{a\} \leftarrow b,\ b \leftarrow \}$ has two stable models: $\{b\}$ and $\{a, b\}$

Embedding in normal rules

- A choice rule of form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

can be translated into $2m + 1$ normal rules

$$\begin{array}{lcl} a' & \leftarrow & a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o \\ a_1 & \leftarrow & a', \sim \overline{a_1} \quad \ldots \quad a_m \leftarrow a', \sim \overline{a_m} \\ \overline{a_1} & \leftarrow & \sim a_1 \quad \ldots \quad \overline{a_m} \leftarrow \sim a_m \end{array}$$

by introducing new atoms $a', \overline{a_1}, \ldots, \overline{a_m}$.

Embedding in normal rules

- A choice rule of form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

can be translated into $2m + 1$ normal rules

$$\begin{array}{lcl} a' & \leftarrow & a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o \\ a_1 & \leftarrow & a', \sim \overline{a_1} \quad \ldots \quad a_m \leftarrow a', \sim \overline{a_m} \\ \overline{a_1} & \leftarrow & \sim a_1 \quad \ldots \quad \overline{a_m} \leftarrow \sim a_m \end{array}$$

by introducing new atoms $a', \overline{a_1}, \ldots, \overline{a_m}$.

Embedding in normal rules

- A choice rule of form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o$$

can be translated into $2m + 1$ normal rules

$$\begin{array}{lcl} a' & \leftarrow & a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o \\ a_1 & \leftarrow & a', \sim \overline{a_1} \quad \ldots \quad a_m \leftarrow a', \sim \overline{a_m} \\ \overline{a_1} & \leftarrow & \sim a_1 \quad \ldots \quad \overline{a_m} \leftarrow \sim a_m \end{array}$$

by introducing new atoms $a', \overline{a_1}, \ldots, \overline{a_m}$.

Outline

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$;
l is a non-negative integer.

- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Note l acts as a lower bound on the body
- Example `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- Another Example $P = \{ \ a \leftarrow 1\{b, c\}, \ b \leftarrow \}$ has stable model $\{a, b\}$

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

  where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$;
  l is a non-negative integer.

- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Note l acts as a lower bound on the body
- Example `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- Another Example $P = \{ \ a \leftarrow 1\{b, c\}, \ b \leftarrow \}$ has stable model $\{a, b\}$

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

  where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$;
  l is a non-negative integer.
- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Note l acts as a lower bound on the body
- Example `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- Another Example $P = \{ \ a \leftarrow 1\{b, c\}, \ b \leftarrow \}$ has stable model $\{a, b\}$

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$;
l is a non-negative integer.

- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Note l acts as a lower bound on the body
- Example `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- Another Example $P = \{ \ a \leftarrow 1\{b, c\}, \ b \leftarrow \}$ has stable model $\{a, b\}$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

An example

- Program $\{\ a \leftarrow,\ c \leftarrow 1\ \{a, b\}\ \}$ has the stable model $\{a, c\}$
- Translating the cardinality rule yields the rules

$$
\begin{array}{lll}
a & \leftarrow & \\
c & \leftarrow & ctr(1,1) \\
ctr(1,2) & \leftarrow & ctr(2,1), a \\
ctr(1,1) & \leftarrow & ctr(2,1) \\
ctr(2,2) & \leftarrow & ctr(3,1), b \\
ctr(2,1) & \leftarrow & ctr(3,1) \\
ctr(1,1) & \leftarrow & ctr(2,0), a \\
ctr(1,0) & \leftarrow & ctr(2,0) \\
ctr(2,1) & \leftarrow & ctr(3,0), b \\
ctr(2,0) & \leftarrow & ctr(3,0) \\
ctr(3,0) & \leftarrow &
\end{array}
$$

having stable model $\{a, ctr(3,0), ctr(2,0), ctr(1,0), ctr(1,1), c\}$

An example

- Program $\{\ a \leftarrow,\ c \leftarrow 1\ \{a, b\}\ \}$ has the stable model $\{a, c\}$
- Translating the cardinality rule yields the rules

$$
\begin{array}{lll}
a & \leftarrow & \\
c & \leftarrow & ctr(1,1) \\
ctr(1,2) & \leftarrow & ctr(2,1), a \\
ctr(1,1) & \leftarrow & ctr(2,1) \\
ctr(2,2) & \leftarrow & ctr(3,1), b \\
ctr(2,1) & \leftarrow & ctr(3,1) \\
ctr(1,1) & \leftarrow & ctr(2,0), a \\
ctr(1,0) & \leftarrow & ctr(2,0) \\
ctr(2,1) & \leftarrow & ctr(3,0), b \\
ctr(2,0) & \leftarrow & ctr(3,0) \\
ctr(3,0) & \leftarrow &
\end{array}
$$

having stable model $\{a, ctr(3,0), ctr(2,0), ctr(1,0), ctr(1,1), c\}$

. . . and vice versa

- A normal rule

$$a_0 \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n,$$

can be represented by the cardinality rule

$$a_0 \leftarrow n\ \{a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n\}$$

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \} \; u$$

where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$;
l and u are non-negative integers

stands for

$$\begin{array}{rcl} a_0 & \leftarrow & b, \sim c \\ b & \leftarrow & l \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \} \\ c & \leftarrow & u{+}1 \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \} \end{array}$$

where b and c are new symbols

- The single constraint in the body of the above cardinality rule is referred to as a cardinality constraint

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \} \ u$$

where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$;
l and u are non-negative integers

stands for

$$\begin{array}{rcl} a_0 & \leftarrow & b, \sim c \\ b & \leftarrow & l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \} \\ c & \leftarrow & u{+}1 \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \} \end{array}$$

where b and c are new symbols

- The single constraint in the body of the above cardinality rule is referred to as a cardinality constraint

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \} \ u$$

where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$;
l and u are non-negative integers

stands for

$$\begin{array}{rcl} a_0 & \leftarrow & b, \sim c \\ b & \leftarrow & l \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \} \\ c & \leftarrow & u{+}1 \ \{ \ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \ \} \end{array}$$

where b and c are new symbols

- The single constraint in the body of the above cardinality rule is referred to as a cardinality constraint

Cardinality constraints

- Syntax A cardinality constraint is of the form

$$l \{ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \} u$$

where $1 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
l and u are non-negative integers

- Informal meaning A cardinality constraint is satisfied by a stable model X, if the number of its contained literals satisfied by X is between l and u (inclusive)
- In other words, if

$$l \leq |(\{a_1, \ldots, a_m\} \cap X) \cup (\{a_{m+1}, \ldots, a_n\} \setminus X)| \leq u$$

Cardinality constraints

- Syntax A cardinality constraint is of the form

$$l \{ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \} u$$

where $1 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
l and u are non-negative integers

- Informal meaning A cardinality constraint is satisfied by a stable model X, if the number of its contained literals satisfied by X is between l and u (inclusive)
- In other words, if

$$l \leq |(\{a_1, \ldots, a_m\} \cap X) \cup (\{a_{m+1}, \ldots, a_n\} \setminus X)| \leq u$$

Cardinality constraints

- Syntax A cardinality constraint is of the form

$$l \{ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \} u$$

where $1 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
l and u are non-negative integers

- Informal meaning A cardinality constraint is satisfied by a stable model X, if the number of its contained literals satisfied by X is between l and u (inclusive)
- In other words, if

$$l \leq |(\{a_1, \ldots, a_m\} \cap X) \cup (\{a_{m+1}, \ldots, a_n\} \setminus X)| \leq u$$

Cardinality constraints as heads

- A rule of the form

$$l\ \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}\ u \leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p$$

where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$;
l and u are non-negative integers

stands for

$$\begin{aligned} b &\leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p \\ \{a_1, \ldots, a_m\} &\leftarrow b \\ c &\leftarrow l\ \{a_1, \ldots, a_m, , \sim a_{m+1}, \ldots, \sim a_n\}\ u \\ &\leftarrow b, \sim c \end{aligned}$$

where b and c are new symbols

- Example `1 { color(v42,red),color(v42,green),color(v42,blue) } 1.`

Cardinality constraints as heads

- A rule of the form

$$l\ \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}\ u \leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p$$

where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$;
l and u are non-negative integers

stands for

$$\begin{array}{rcl} b & \leftarrow & a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p \\ \{a_1, \ldots, a_m\} & \leftarrow & b \\ c & \leftarrow & l\ \{a_1, \ldots, a_m, , \sim a_{m+1}, \ldots, \sim a_n\}\ u \\ & \leftarrow & b, \sim c \end{array}$$

where b and c are new symbols

- Example `1 { color(v42,red),color(v42,green),color(v42,blue) } 1.`

Cardinality constraints as heads

- A rule of the form

$$l\ \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}\ u \leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p$$

where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$;
l and u are non-negative integers

stands for

$$\begin{array}{rcl} b & \leftarrow & a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p \\ \{a_1, \ldots, a_m\} & \leftarrow & b \\ c & \leftarrow & l\ \{a_1, \ldots, a_m, , \sim a_{m+1}, \ldots, \sim a_n\}\ u \\ & \leftarrow & b, \sim c \end{array}$$

where b and c are new symbols

- Example `1 { color(v42,red),color(v42,green),color(v42,blue) } 1.`

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

where for $1 \leq i \leq n$ each $l_i\ S_i\ u_i$

stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{lll} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, \sim b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{lll} b_i & \leftarrow & l_i\ S_i \\ c_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

where for $1 \leq i \leq n$ each $l_i\ S_i\ u_i$
stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{lll} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, \sim b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{lll} b_i & \leftarrow & l_i\ S_i \\ c_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

where for $1 \leq i \leq n$ each $l_i\ S_i\ u_i$
stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{lll} {S_0}^+ \leftarrow a & & \\ \leftarrow a, \sim b_0 & b_i \leftarrow l_i\ S_i & \\ \leftarrow a, c_0 & c_i \leftarrow u_i{+}1\ S_i & \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

where for $1 \leq i \leq n$ each $l_i\ S_i\ u_i$
stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{lll} {S_0}^+ \leftarrow a & & \\ \leftarrow a, \sim b_0 & b_i \leftarrow & l_i\ S_i \\ \leftarrow a, c_0 & c_i \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

where for $1 \leq i \leq n$ each $l_i\ S_i\ u_i$
stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{lll} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, \sim b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{lll} b_i & \leftarrow & l_i\ S_i \\ c_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

where for $1 \leq i \leq n$ each $l_i\ S_i\ u_i$
stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{lll} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, \sim b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{lll} b_i & \leftarrow & l_i\ S_i \\ c_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols

Outline

Weight rule

- Syntax A weight rule is the form

$$a_0 \leftarrow l \ \{ \ a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n \ \}$$

  where $0 \leq m \leq n$ and each a_i is an atom;
  l and w_i are integers for $0 \leq i \leq n$
- A weighted literal, $\ell_i = w_i$, associates each literal ℓ_i with a weight w_i
- Note A cardinality rule is a weight rule where $w_i = 1$ for $0 \leq i \leq n$

Weight rule

- Syntax A weight rule is the form

$$a_0 \leftarrow l \ \{ \ a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n \ \}$$

  where $0 \leq m \leq n$ and each a_i is an atom;
  l and w_i are integers for $0 \leq i \leq n$
- A weighted literal, $\ell_i = w_i$, associates each literal ℓ_i with a weight w_i
- Note A cardinality rule is a weight rule where $w_i = 1$ for $0 \leq i \leq n$

Weight constraints

- Syntax A weight constraint is of the form

$$l \; \{ \; a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n \; \} \; u$$

  where $1 \leq m \leq n$ and each a_i is an atom;
  l, u and w_i are integers for $0 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 {course(db)=6,course(ai)=6,course(project)=8,course(xml)=3} 20`

Weight constraints

- Syntax A weight constraint is of the form

$$l \left\{ a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n \right\} u$$

  where $1 \leq m \leq n$ and each a_i is an atom;
  l, u and w_i are integers for $0 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 {course(db)=6,course(ai)=6,course(project)=8,course(xml)=3} 20`

Weight constraints

- Syntax A weight constraint is of the form

$$l \; \{ \; a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n \; \} \; u$$

  where $1 \leq m \leq n$ and each a_i is an atom;
  l, u and w_i are integers for $0 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 {course(db)=6,course(ai)=6,course(project)=8,course(xml)=3} 20`

Weight constraints

- Syntax A weight constraint is of the form

$$l \left\{ a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n \right\} u$$

  where $1 \leq m \leq n$ and each a_i is an atom;
  l, u and w_i are integers for $0 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 {course(db)=6,course(ai)=6,course(project)=8,course(xml)=3} 20`

Outline

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1 : \cdots : \ell_n$$

where ℓ and ℓ_i are literals for $0 \leq i \leq n$

- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given ‘`p(1). p(2). p(3). q(2).`’

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1 : \cdots : \ell_n$$

where ℓ and ℓ_i are literals for $0 \leq i \leq n$

- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given ‘`p(1). p(2). p(3). q(2).`’

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1 : \cdots : \ell_n$$

where ℓ and ℓ_i are literals for $0 \leq i \leq n$

- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given ‘`p(1). p(2). p(3). q(2).`’

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1 : \cdots : \ell_n$$

where ℓ and ℓ_i are literals for $0 \leq i \leq n$

- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given ‘`p(1). p(2). p(3). q(2).`’

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1 : \cdots : \ell_n$$

where ℓ and ℓ_i are literals for $0 \leq i \leq n$

- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given ‘`p(1). p(2). p(3). q(2).`’

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1 : \cdots : \ell_n$$

where ℓ and ℓ_i are literals for $0 \leq i \leq n$

- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given ‘`p(1). p(2). p(3). q(2).`’

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1 : \cdots : \ell_n$$

where ℓ and ℓ_i are literals for $0 \leq i \leq n$

- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given ‘`p(1). p(2). p(3). q(2).`’

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Outline

Optimization statement

- Idea Express cost functions subject to minimization and/or maximization
- Syntax A minimize statement is of the form

$$\mathit{minimize}\{\ \ell_1 = w_1@p_1, \ldots, \ell_n = w_n@p_n\ \}.$$

where each ℓ_i is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

Priority levels, p_i, allow for representing lexicographically ordered minimization objectives

- Meaning A minimize statement is a directive that instructs the ASP solver to compute optimal stable models by minimizing a weighted sum of elements

Optimization statement

- Idea Express cost functions subject to minimization and/or maximization
- Syntax A minimize statement is of the form

$$\mathit{minimize}\{\ \ell_1 = w_1@p_1, \ldots, \ell_n = w_n@p_n\ \}.$$

where each ℓ_i is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

Priority levels, p_i, allow for representing lexicographically ordered minimization objectives

- Meaning A minimize statement is a directive that instructs the ASP solver to compute optimal stable models by minimizing a weighted sum of elements

Optimization statement

- Idea Express cost functions subject to minimization and/or maximization
- Syntax A minimize statement is of the form

$$\mathit{minimize}\{\ \ell_1 = w_1@p_1, \ldots, \ell_n = w_n@p_n\ \}.$$

where each ℓ_i is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

Priority levels, p_i, allow for representing lexicographically ordered minimization objectives

- Meaning A minimize statement is a directive that instructs the ASP solver to compute optimal stable models by minimizing a weighted sum of elements

Optimization statement

- A maximize statement of the form

$$\mathit{maximize}\{\ \ell_1 = w_1@p_1, \ldots, \ell_n = w_n@p_n\ \}$$

stands for $\mathit{minimize}\{\ \ell_1 = -w_1@p_1, \ldots, \ell_n = -w_n@p_n\ \}$

- Example When configuring a computer, we may want to maximize hard disk capacity, while minimizing price

```
#maximize[ hd(1)=250@1, hd(2)=500@1, hd(3)=750@1, hd(4)=1000@1 ].
#minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
```

The priority levels indicate that (minimizing) price is more important than (maximizing) capacity

Optimization statement

- A maximize statement of the form

  $$\mathit{maximize}\{\ \ell_1 = w_1@p_1, \ldots, \ell_n = w_n@p_n\ \}$$

  stands for $\mathit{minimize}\{\ \ell_1 = -w_1@p_1, \ldots, \ell_n = -w_n@p_n\ \}$

- Example When configuring a computer, we may want to maximize hard disk capacity, while minimizing price

```
#maximize[ hd(1)=250@1, hd(2)=500@1, hd(3)=750@1, hd(4)=1000@1 ].
#minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
```

  The priority levels indicate that (minimizing) price is more important than (maximizing) capacity

Outline

smodels format

- Logic programs in *smodels* format consist of
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - optimization statements
- Such a format is obtained by grounders *lparse* and `gringo`

Systems: Overview

23 Potassco

24 gringo

25 clasp

26 Siblings
- claspfolio
- clingcon
- iclingo
- oclingo

Outline

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, lingo, pyngo
- *Solver*: clasp, {a,h,pb,un}clasp, claspD, claspfolio, claspar, aspeed
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: asp{un}cud, coala, fimo, metasp, plasp, etc

- *Benchmark repository*: http://asparagus.cs.uni-potsdam.de

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, lingo, pyngo
- *Solver*: clasp, {a,h,pb,un}clasp, claspD, claspfolio, claspar, aspeed
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: asp{un}cud, coala, fimo, metasp, plasp, etc

- *Benchmark repository*: http://asparagus.cs.uni-potsdam.de

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: gringo, lingo, pyngo
- *Solver*: clasp, {a,h,pb,un}clasp, claspD, claspfolio, claspar, aspeed
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: asp{un}cud, coala, fimo, metasp, plasp, etc

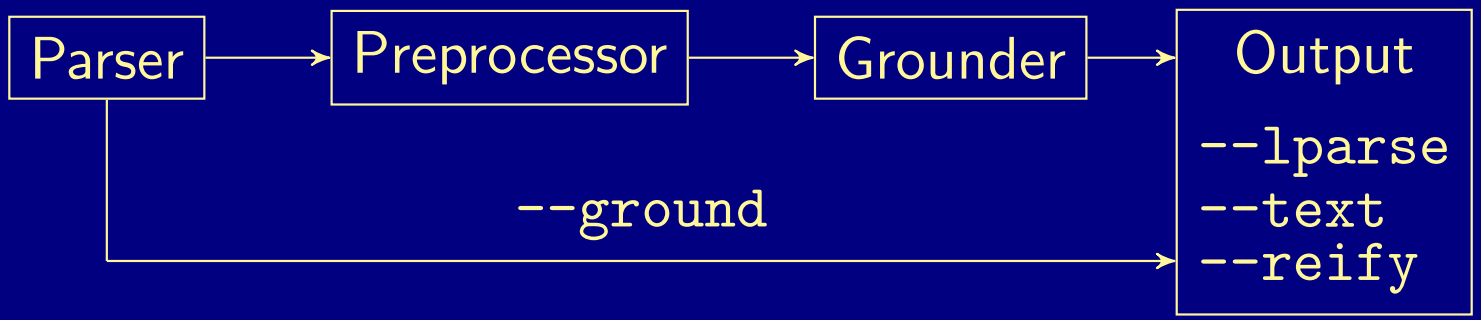
- *Benchmark repository*: http://asparagus.cs.uni-potsdam.de

Outline

gringo

- Accepts safe programs with aggregates
- Tolerates unrestricted use of function symbols
  (as long as it yields a finite ground instantiation :)
- Expressive power of a Turing machine

- Basic architecture of *gringo*:



An example

$d(a)$
$d(c)$
$d(d)$

$p(a, b)$
$p(b, c)$
$p(c, d)$
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$

$q(a)$
$q(b)$
$q(X) \leftarrow \sim r(X), d(X)$

$r(X) \leftarrow \sim q(X), d(X)$
$s(X) \leftarrow \sim r(X), p(X, Y), q(Y)$

An example

Safe ?

$d(a)$
$d(c)$
$d(d)$

$p(a, b)$
$p(b, c)$
$p(c, d)$
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$

$q(a)$
$q(b)$
$q(X) \leftarrow \sim r(X), d(X)$

$r(X) \leftarrow \sim q(X), d(X)$
$s(X) \leftarrow \sim r(X), p(X, Y), q(Y)$

An example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a,b)$	✔
$p(b,c)$	✔
$p(c,d)$	✔
$p(X,Z) \leftarrow p(X,Y), p(Y,Z)$	
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	
$r(X) \leftarrow \sim q(X), d(X)$	
$s(X) \leftarrow \sim r(X), p(X,Y), q(Y)$	

An example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a,b)$	✔
$p(b,c)$	✔
$p(c,d)$	✔
$p(X,Z) \leftarrow p(X,Y), p(Y,Z)$	
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	
$r(X) \leftarrow \sim q(X), d(X)$	
$s(X) \leftarrow \sim r(X), p(X,Y), q(Y)$	

An example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a,b)$	✔
$p(b,c)$	✔
$p(c,d)$	✔
$p(X,Z) \leftarrow p(X,Y), p(Y,Z)$	✔
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	✔
$r(X) \leftarrow \sim q(X), d(X)$	✔
$s(X) \leftarrow \sim r(X), p(X,Y), q(Y)$	✔

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{\, \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \,\}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{\, \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \,\}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{\, \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \,\}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{\, \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \,\}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

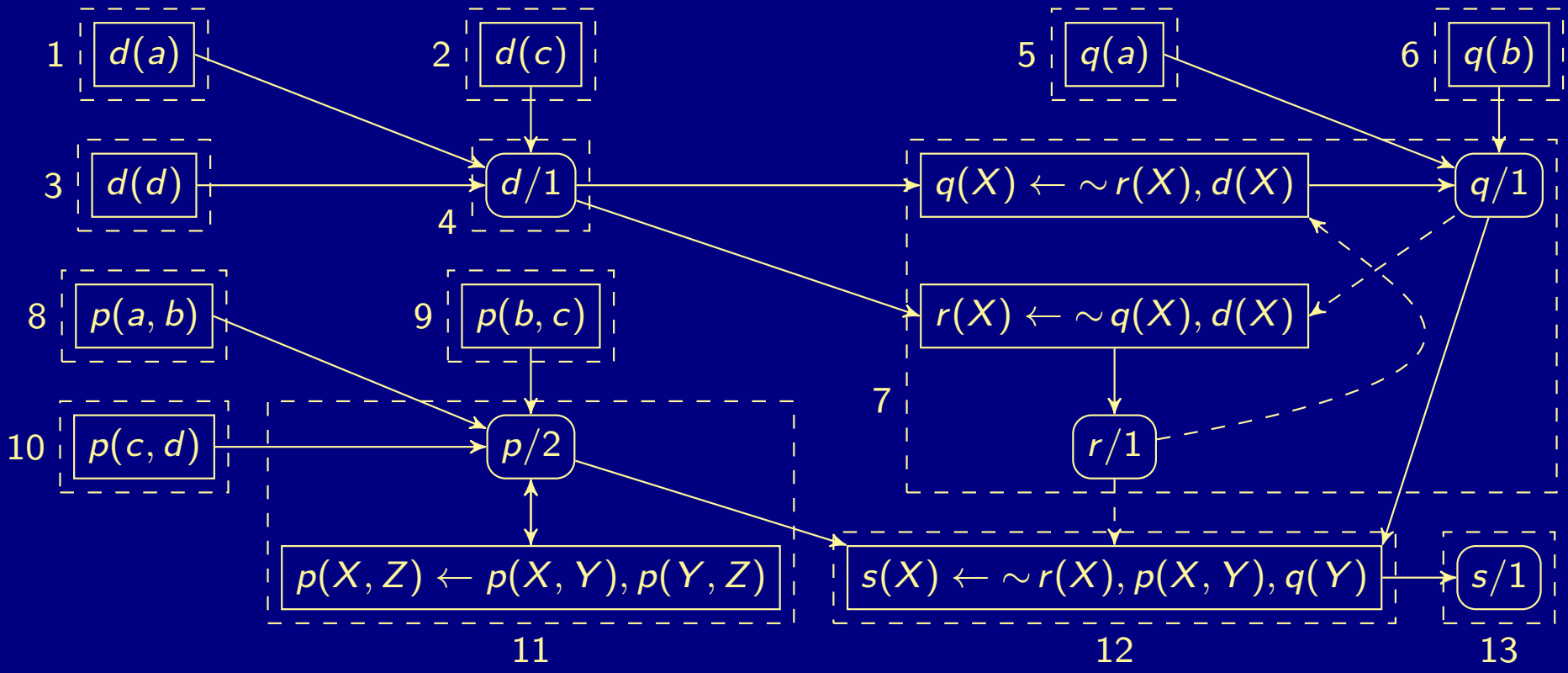
Naive instantiation

Algorithm 1: NaiveInstantiation

Input : A safe (first-order) logic program P
Output : A ground logic program P'

```
1  D := ∅
2  P' := ∅
3  repeat
4  |   D' := D
5  |   foreach r ∈ P do
6  |   |   B := body(r)^+
7  |   |   foreach θ ∈ Θ(B, D) do
8  |   |   |   D := D ∪ {head(r)θ}
9  |   |   |   P' := P' ∪ {rθ}
10 until D = D'
```

Predicate-rule dependency graph



Instantiation

SCC	$\Theta(B, D)$	D	P'
1	$\{\emptyset\}$	$d(a)$	$d(a) \leftarrow$
2	$\{\emptyset\}$	$d(c)$	$d(c) \leftarrow$
3	$\{\emptyset\}$	$d(d)$	$d(d) \leftarrow$
5	$\{\emptyset\}$	$q(a)$	$q(a) \leftarrow$
6	$\{\emptyset\}$	$q(b)$	$q(b) \leftarrow$
7	$\{\{X \mapsto a\},$		$q(a) \leftarrow \sim r(a), d(a)$
	$\{X \mapsto c\},$	$q(c)$	$q(c) \leftarrow \sim r(c), d(c)$
	$\{X \mapsto d\},$	$q(d)$	$q(d) \leftarrow \sim r(d), d(d)$
	$\{X \mapsto a\},$		$r(a) \leftarrow \sim q(a), d(a)$
	$\{X \mapsto c\},$	$r(c)$	$r(c) \leftarrow \sim q(c), d(c)$
	$\{X \mapsto d\}\}$	$r(d)$	$r(d) \leftarrow \sim q(d), d(d)$

Instantiation

SCC	$\Theta(B, D)$	D	P'
8	$\{\emptyset\}$	$p(a,b)$	$p(a,b) \leftarrow$
9	$\{\emptyset\}$	$p(b,c)$	$p(b,c) \leftarrow$
10	$\{\emptyset\}$	$p(c,d)$	$p(c,d) \leftarrow$
11	$\{\{X \mapsto a, Y \mapsto b, Z \mapsto c\},$ $\{X \mapsto b, Y \mapsto c, Z \mapsto d\}\}$	$p(a,c)$ $p(b,d)$	$p(a,c) \leftarrow p(a,b), p(b,c)$ $p(b,d) \leftarrow p(b,c), p(c,d)$
	$\{\{X \mapsto a, Y \mapsto c, Z \mapsto d\},$ $\{X \mapsto a, Y \mapsto b, Z \mapsto d\}\}$	$p(a,d)$	$p(a,d) \leftarrow p(a,c), p(c,d)$ $p(a,d) \leftarrow p(a,b), p(b,d)$
12	$\{\{X \mapsto a, Y \mapsto b\},$ $\{X \mapsto a, Y \mapsto c\},$ $\{X \mapsto a, Y \mapsto d\},$ $\{X \mapsto b, Y \mapsto c\},$ $\{X \mapsto b, Y \mapsto d\},$ $\{X \mapsto c, Y \mapsto d\}\}$	$s(a)$ $s(b)$ $s(c)$	$s(a) \leftarrow \sim r(a), p(a,b), q(b)$ $s(a) \leftarrow \sim r(a), p(a,c), q(c)$ $s(a) \leftarrow \sim r(a), p(a,d), q(d)$ $s(b) \leftarrow \sim r(b), p(b,c), q(c)$ $s(b) \leftarrow \sim r(b), p(b,d), q(d)$ $s(c) \leftarrow \sim r(c), p(c,d), q(d)$

Outline

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - Advanced preprocessing including, like equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - Advanced preprocessing including, like equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - Advanced preprocessing including, like equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

Reasoning modes of *clasp*

- Beyond deciding (stable) model existence, *clasp* allows for:
  - Optimization
  - Enumeration (without solution recording)
  - Projective enumeration (without solution recording)
  - Intersection and Union (linear solution computation)
  - and combinations thereof
- *clasp* allows for
  - ASP solving (*smodels* format)
  - MaxSAT and SAT solving (extended *dimacs* format)
  - PB solving (*opb* and *wbo* format)

Reasoning modes of *clasp*

- Beyond deciding (stable) model existence, *clasp* allows for:
  - Optimization
  - Enumeration (without solution recording)
  - Projective enumeration (without solution recording)
  - Intersection and Union (linear solution computation)
  - and combinations thereof
- *clasp* allows for
  - ASP solving (*smodels* format)
  - MaxSAT and SAT solving (extended *dimacs* format)
  - PB solving (*opb* and *wbo* format)

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Sequential CDCL-style solving

```
loop
    propagate                          // deterministically assign literals
    if no conflict then
        if all variables assigned then return solution
        else decide                    // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            analyze                    // analyze conflict and add conflict constraint
            backjump                   // unassign literals until conflict constraint is unit
```

Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

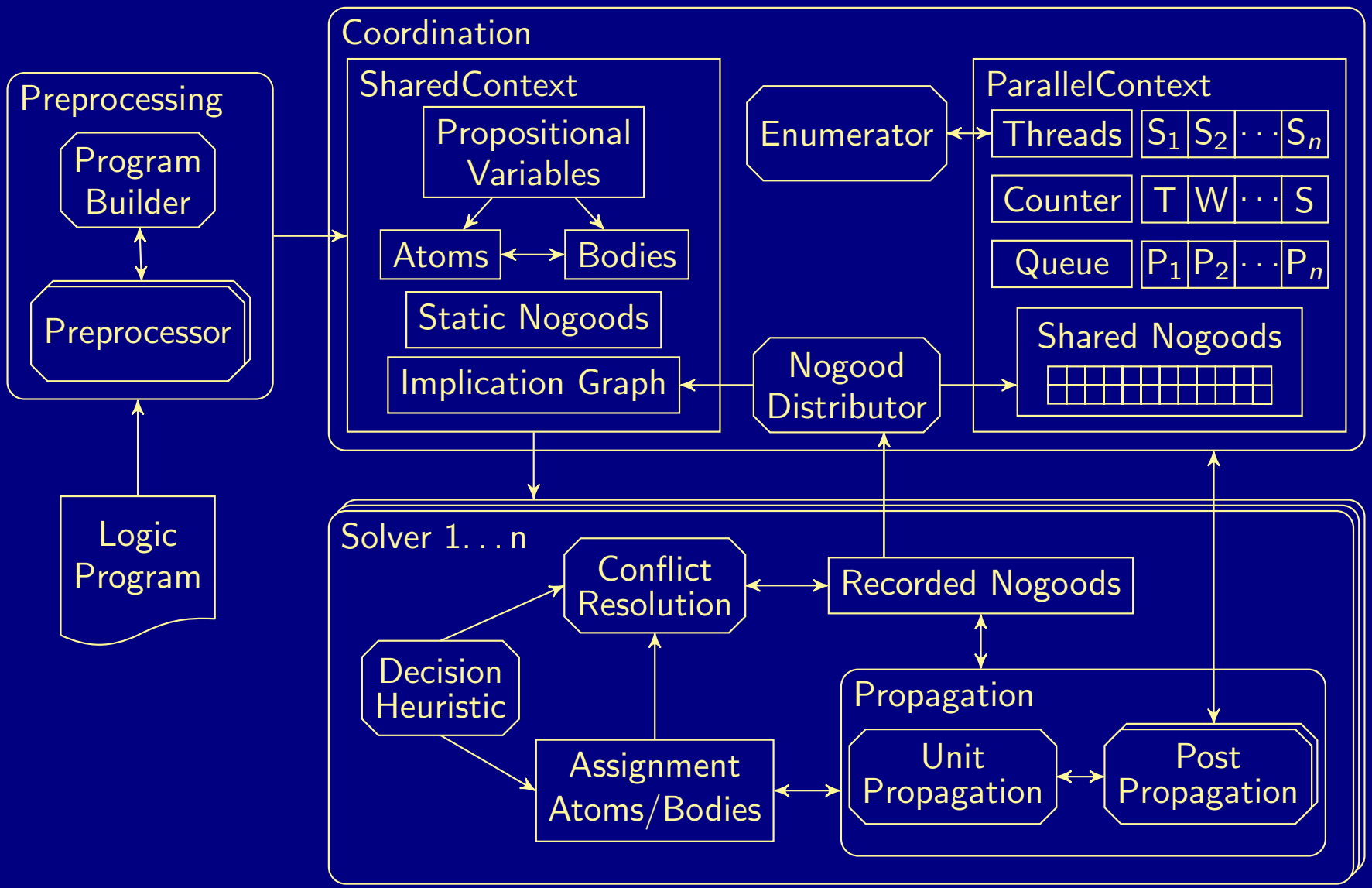
Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

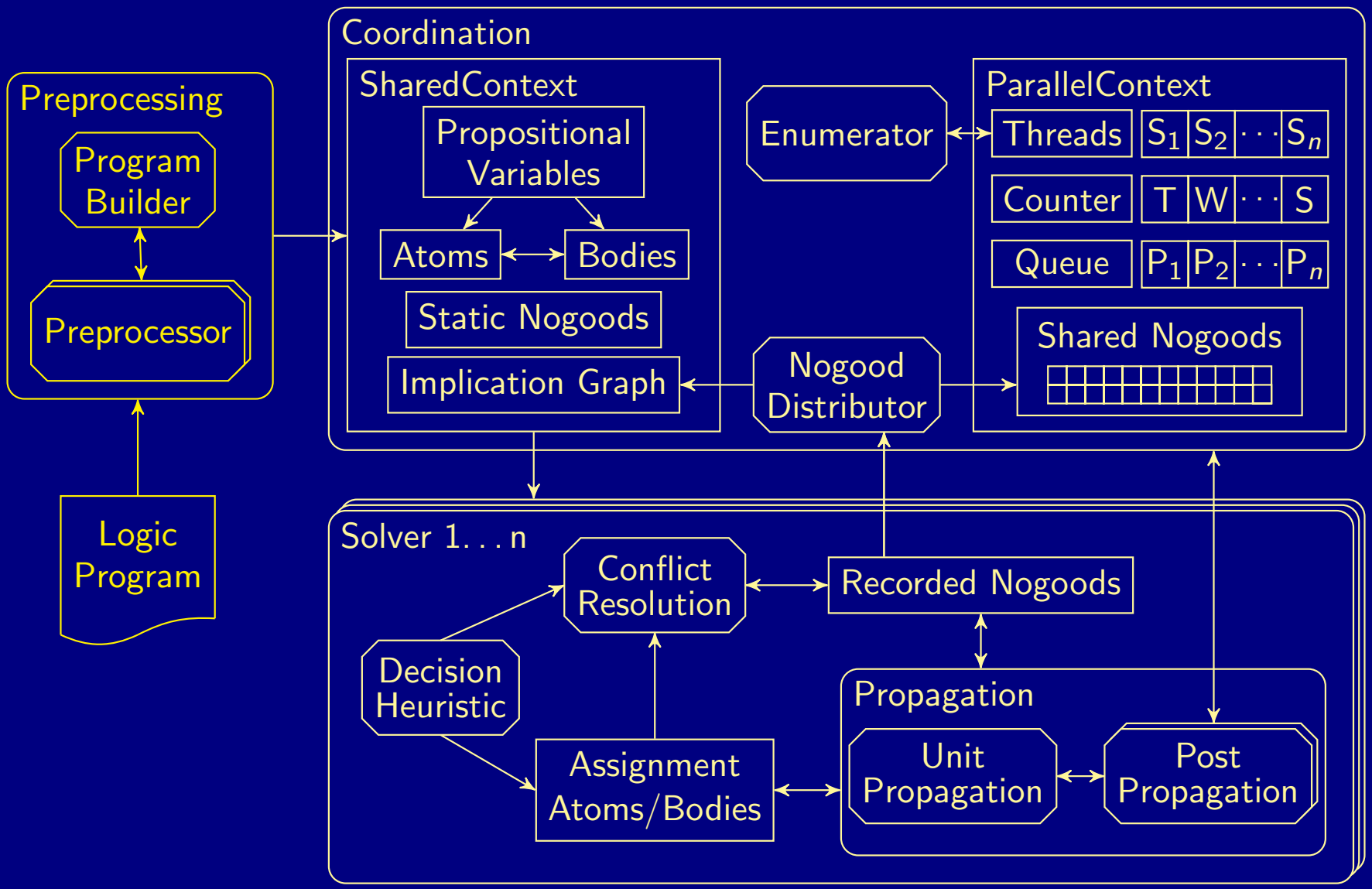
Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

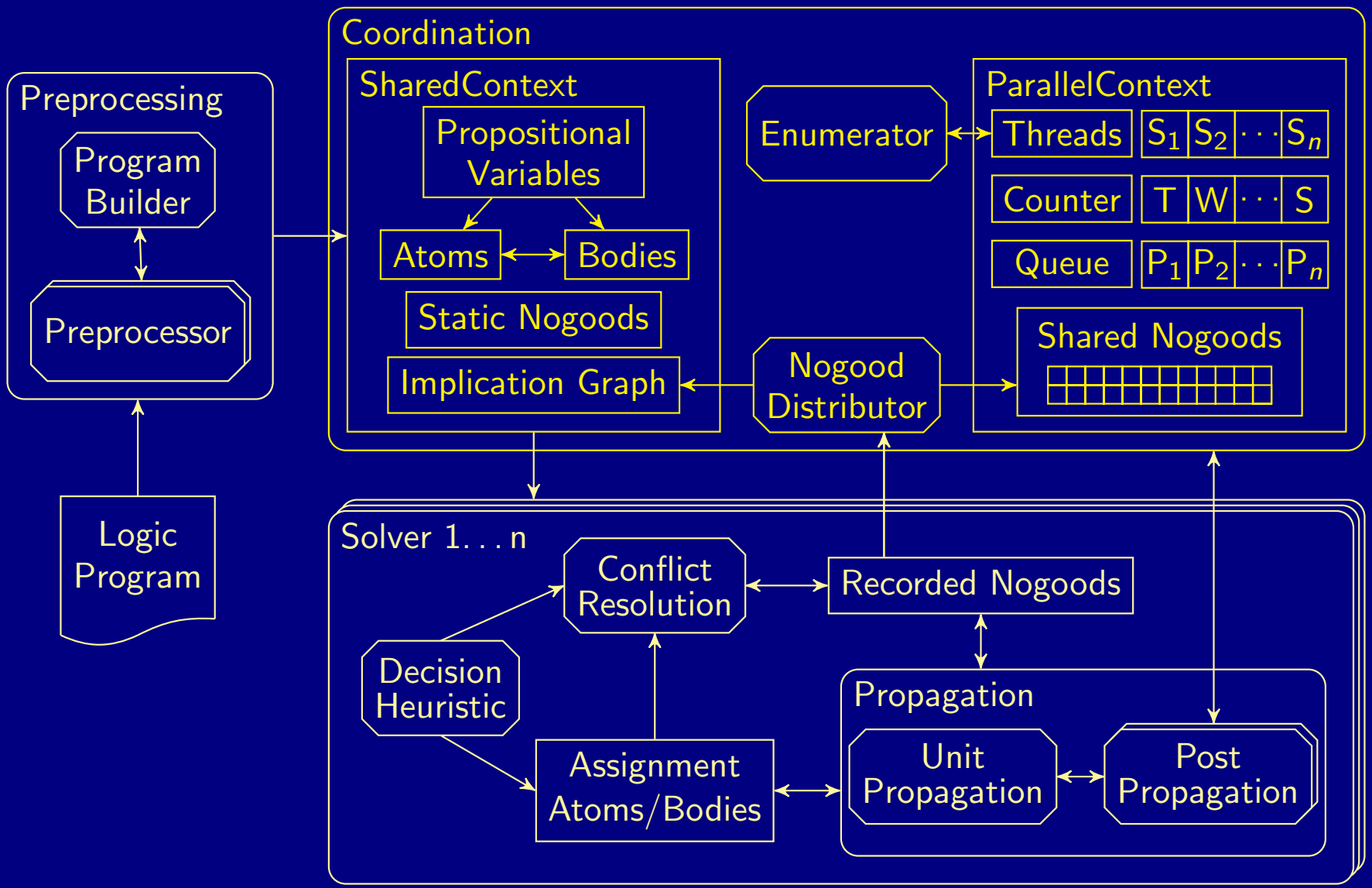
Multi-threaded architecture of *clasp*



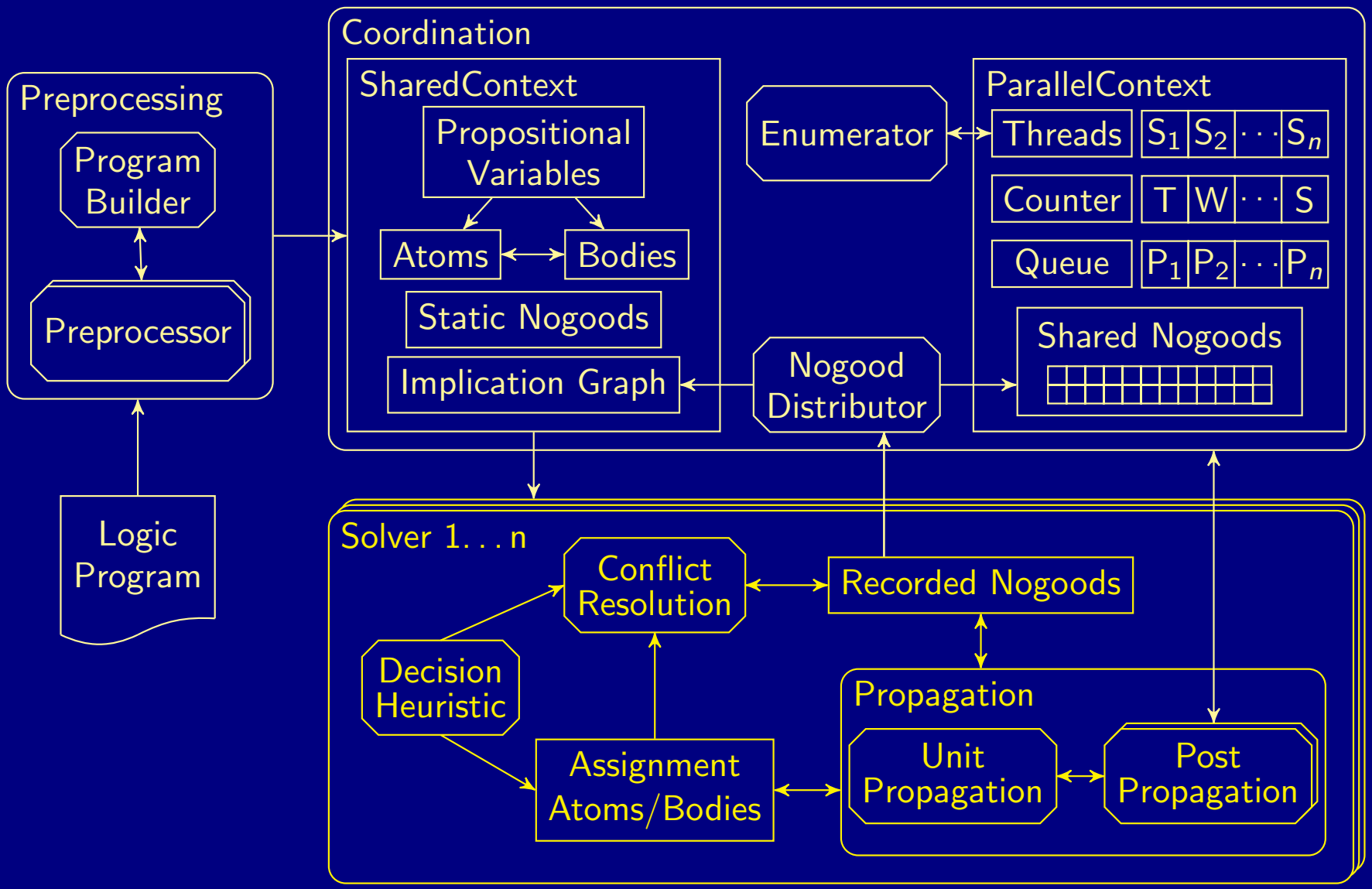
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*

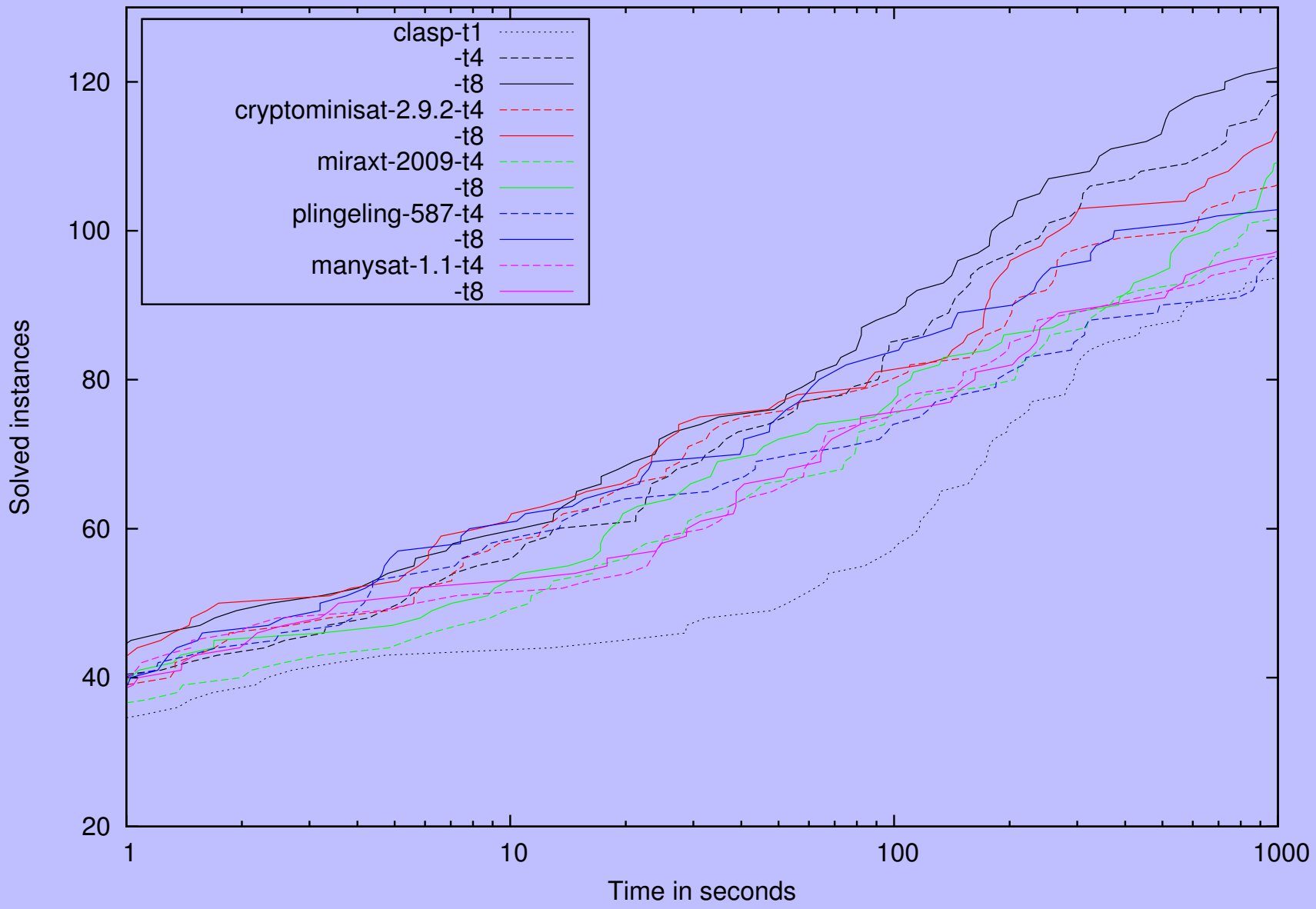


clasp in context

- Compare *clasp* (2.0.5) to the multi-threaded SAT solvers
  - *cryptominisat* (2.9.2)
  - *manysat* (1.1)
  - *miraxt* (2009)
  - *plingeling* (587f)

  all run with four and eight threads in their default settings
- 160/300 benchmarks from crafted category at SAT'11
  - all solvable by *ppfolio* in 1000 seconds
  - crafted SAT benchmarks are closest to ASP benchmarks

clasp in context



Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Using *clasp*

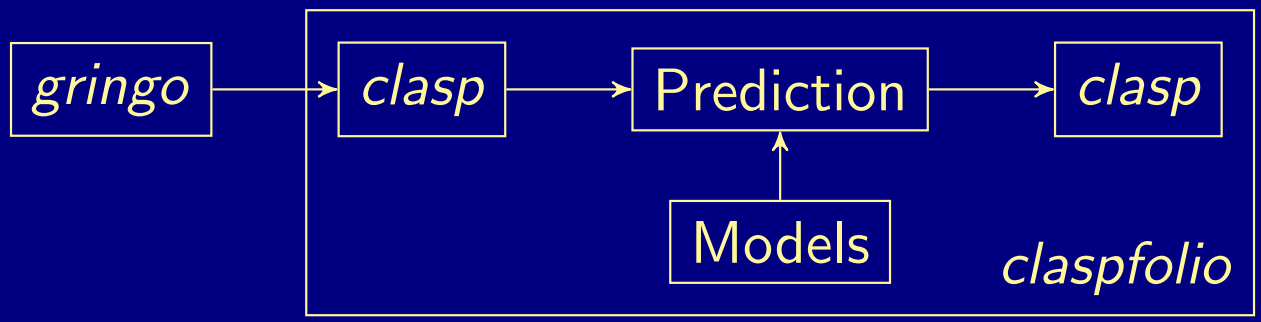
```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Outline

Outline

claspfolio

- Automatic selection of *clasp* configuration among 25 configuration via (learned) classifiers
- Basic architecture of *claspfolio*:



Solving with *clasp* (as usual)

```
$ clasp queens500 --quiet

clasp version 2.0.2
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 11.445s (Solving: 10.58s 1st Model: 10.55s Unsat: 0.00s)
CPU Time     : 11.410s
```

Solving with *clasp* (as usual)

```
$ clasp queens500 --quiet

clasp version 2.0.2
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 11.445s (Solving: 10.58s 1st Model: 10.55s Unsat: 0.00s)
CPU Time     : 11.410s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars, ...
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars, ...
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars, ...
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt          \
      --modelpath=./models/                            \
      queens500 --quiet --autoverbose=1                \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time     : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt          \
      --modelpath=./models/                            \
      queens500 --quiet --autoverbose=1                \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time     : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt          \
      --modelpath=./models/                            \
      queens500 --quiet --autoverbose=1                \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time     : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt          \
      --modelpath=./models/                            \
      queens500 --quiet --autoverbose=1                \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time     : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

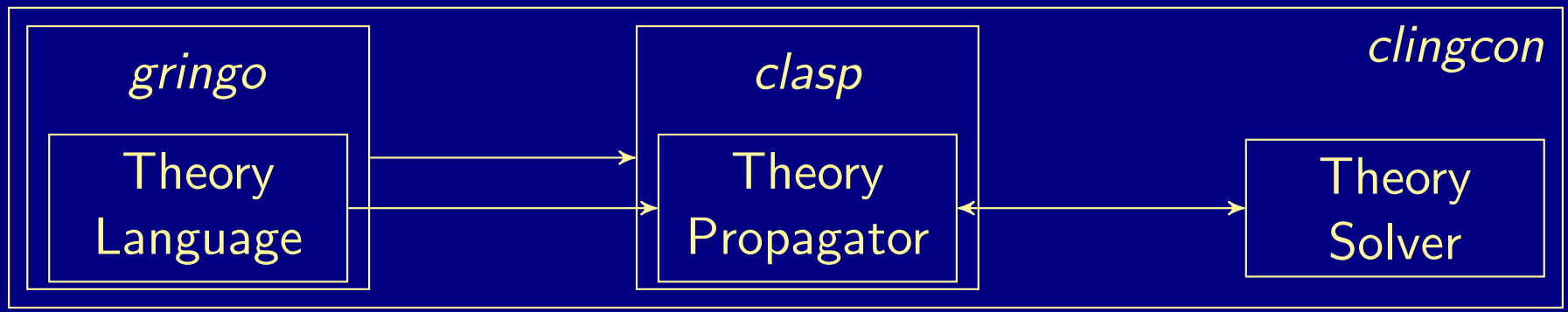
```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time     : 4.760s
```

Outline

clingcon

- Hybrid grounding and solving
- Solving in hybrid domains, like Bio-Informatics

- Basic architecture of *clingcon*:



Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, not (1 $<= amount(B,T)).
amount(B,T) $<= 30           :- pour(B,T), T < t.
amount(B,T) $==  0           :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
amount(B,T) $<= 30           :- pour(B,T), T < t.
amount(B,T) $==  0           :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
amount(B,T) $==  0              :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
 :- not pour(B,T), bucket(B), time(T), T < t, amount(B,T) $!=  0.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
 :- not pour(B,T), bucket(B), time(T), T < t, amount(B,T) $!=  0.

 :- bucket(B), time(T), T < t, volume(B,T+1) $!= volume(B,T)$+amount(B,T).

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text
```

```
time(0). ... time(4).
bucket(a).
bucket(b).

1 { pour(b,0), pour(a,0) } 1.

 :- pour(a,0), 1 $> amount(a,0).
 :- pour(b,0), 1 $> amount(b,0).

 :- pour(a,0), amount(a,0) $> 30.
 :- pour(b,0), amount(b,0) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).

down(a,0) :- volume(a,0) $< volume(a,0).
down(a,0) :- volume(b,0) $< volume(a,0).
down(b,0) :- volume(a,0) $< volume(b,0).
down(b,0) :- volume(b,0) $< volume(b,0).

up(a,0) :- not down(a,0).
up(b,0) :- not down(b,0).

 :- up(a,4).
```

```
$domain(0..500).
 :- volume(a,0) $!= 0.
 :- volume(b,0) $!= 100.

... 1 { pour(b,3), pour(a,3) } 1.

...  :- pour(a,3), 1 $> amount(a,3).
...  :- pour(b,3), 1 $> amount(b,3).

...  :- pour(a,3), amount(a,3) $> 30.
...  :- pour(b,3), amount(b,3) $> 30.

...  :- not pour(a,3), amount(a,3) $!= 0.
...  :- not pour(b,3), amount(b,3) $!= 0.

...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

... down(a,4) :- volume(a,4) $< volume(a,4).
... down(a,4) :- volume(b,4) $< volume(a,4).
... down(b,4) :- volume(a,4) $< volume(b,4).
... down(b,4) :- volume(b,4) $< volume(b,4).

... up(a,4) :- not down(a,4).
... up(b,4) :- not down(b,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text
```

```
time(0). ... time(4).
bucket(a).
bucket(b).

1 { pour(b,0), pour(a,0) } 1.

 :- pour(a,0), 1 $> amount(a,0).
 :- pour(b,0), 1 $> amount(b,0).

 :- pour(a,0), amount(a,0) $> 30.
 :- pour(b,0), amount(b,0) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).

down(a,0) :- volume(a,0) $< volume(a,0).
down(a,0) :- volume(b,0) $< volume(a,0).
down(b,0) :- volume(a,0) $< volume(b,0).
down(b,0) :- volume(b,0) $< volume(b,0).

up(a,0) :- not down(a,0).
up(b,0) :- not down(b,0).

 :- up(a,4).
```

```
$domain(0..500).
 :- volume(a,0) $!= 0.
 :- volume(b,0) $!= 100.

... 1 { pour(b,3), pour(a,3) } 1.

...  :- pour(a,3), 1 $> amount(a,3).
...  :- pour(b,3), 1 $> amount(b,3).

...  :- pour(a,3), amount(a,3) $> 30.
...  :- pour(b,3), amount(b,3) $> 30.

...  :- not pour(a,3), amount(a,3) $!= 0.
...  :- not pour(b,3), amount(b,3) $!= 0.

...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

... down(a,4) :- volume(a,4) $< volume(a,4).
... down(a,4) :- volume(b,4) $< volume(a,4).
... down(b,4) :- volume(a,4) $< volume(b,4).
... down(b,4) :- volume(b,4) $< volume(b,4).

... up(a,4) :- not down(a,4).
... up(b,4) :- not down(b,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text
```

```
time(0). ... time(4).
bucket(a).
bucket(b).

1 { pour(b,0), pour(a,0) } 1.

 :- pour(a,0), 1 $> amount(a,0).
 :- pour(b,0), 1 $> amount(b,0).

 :- pour(a,0), amount(a,0) $> 30.
 :- pour(b,0), amount(b,0) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).

down(a,0) :- volume(a,0) $< volume(a,0).
down(a,0) :- volume(b,0) $< volume(a,0).
down(b,0) :- volume(a,0) $< volume(b,0).
down(b,0) :- volume(b,0) $< volume(b,0).

up(a,0) :- not down(a,0).
up(b,0) :- not down(b,0).

 :- up(a,4).
```

```
$domain(0..500).
 :- volume(a,0) $!= 0.
 :- volume(b,0) $!= 100.

... 1 { pour(b,3), pour(a,3) } 1.

...  :- pour(a,3), 1 $> amount(a,3).
...  :- pour(b,3), 1 $> amount(b,3).

...  :- pour(a,3), amount(a,3) $> 30.
...  :- pour(b,3), amount(b,3) $> 30.

...  :- not pour(a,3), amount(a,3) $!= 0.
...  :- not pour(b,3), amount(b,3) $!= 0.

...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

... down(a,4) :- volume(a,4) $< volume(a,4).
... down(a,4) :- volume(b,4) $< volume(a,4).
... down(b,4) :- volume(a,4) $< volume(b,4).
... down(b,4) :- volume(b,4) $< volume(b,4).

... up(a,4) :- not down(a,4).
... up(b,4) :- not down(b,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text
```

```
time(0). ... time(4).
bucket(a).
bucket(b).

1 { pour(b,0), pour(a,0) } 1.

 :- pour(a,0), 1 $> amount(a,0).
 :- pour(b,0), 1 $> amount(b,0).

 :- pour(a,0), amount(a,0) $> 30.
 :- pour(b,0), amount(b,0) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).

down(a,0) :- volume(a,0) $< volume(a,0).
down(a,0) :- volume(b,0) $< volume(a,0).
down(b,0) :- volume(a,0) $< volume(b,0).
down(b,0) :- volume(b,0) $< volume(b,0).

up(a,0) :- not down(a,0).
up(b,0) :- not down(b,0).

 :- up(a,4).
```

```
$domain(0..500).
 :- volume(a,0) $!= 0.
 :- volume(b,0) $!= 100.

... 1 { pour(b,3), pour(a,3) } 1.

...  :- pour(a,3), 1 $> amount(a,3).
...  :- pour(b,3), 1 $> amount(b,3).

...  :- pour(a,3), amount(a,3) $> 30.
...  :- pour(b,3), amount(b,3) $> 30.

...  :- not pour(a,3), amount(a,3) $!= 0.
...  :- not pour(b,3), amount(b,3) $!= 0.

...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

... down(a,4) :- volume(a,4) $< volume(a,4).
... down(a,4) :- volume(b,4) $< volume(a,4).
... down(b,4) :- volume(a,4) $< volume(b,4).
... down(b,4) :- volume(b,4) $< volume(b,4).

... up(a,4) :- not down(a,4).
... up(b,4) :- not down(b,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text
```

```
time(0). ... time(4).
bucket(a).
bucket(b).

1 { pour(b,0), pour(a,0) } 1.

 :- pour(a,0), 1 $> amount(a,0).
 :- pour(b,0), 1 $> amount(b,0).

 :- pour(a,0), amount(a,0) $> 30.
 :- pour(b,0), amount(b,0) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).

down(a,0) :- volume(a,0) $< volume(a,0).
down(a,0) :- volume(b,0) $< volume(a,0).
down(b,0) :- volume(a,0) $< volume(b,0).
down(b,0) :- volume(b,0) $< volume(b,0).

up(a,0) :- not down(a,0).
up(b,0) :- not down(b,0).

 :- up(a,4).
```

```
$domain(0..500).
 :- volume(a,0) $!= 0.
 :- volume(b,0) $!= 100.

... 1 { pour(b,3), pour(a,3) } 1.

...  :- pour(a,3), 1 $> amount(a,3).
...  :- pour(b,3), 1 $> amount(b,3).

...  :- pour(a,3), amount(a,3) $> 30.
...  :- pour(b,3), amount(b,3) $> 30.

...  :- not pour(a,3), amount(a,3) $!= 0.
...  :- not pour(b,3), amount(b,3) $!= 0.

...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

... down(a,4) :- volume(a,4) $< volume(a,4).
... down(a,4) :- volume(b,4) $< volume(a,4).
... down(b,4) :- volume(a,4) $< volume(b,4).
... down(b,4) :- volume(b,4) $< volume(b,4).

... up(a,4) :- not down(a,4).
... up(b,4) :- not down(b,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Boolean variables

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Non-Boolean variables

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11          amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30          amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30          amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30          amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0           volume(b,0)=100        volume(a,0) $< volume(b,0)
volume(a,1)=11          volume(b,1)=100        volume(a,1) $< volume(b,1)
volume(a,2)=41          volume(b,2)=100        volume(a,2) $< volume(b,2)
volume(a,3)=71          volume(b,3)=100        volume(a,3) $< volume(b,3)
volume(a,4)=101         volume(b,4)=100        volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1+
Time         : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11          amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30          amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30          amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30          amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0           volume(b,0)=100        volume(a,0) $< volume(b,0)
volume(a,1)=11          volume(b,1)=100        volume(a,1) $< volume(b,1)
volume(a,2)=41          volume(b,2)=100        volume(a,2) $< volume(b,2)
volume(a,3)=71          volume(b,3)=100        volume(a,3) $< volume(b,3)
volume(a,4)=101         volume(b,4)=100        volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1+
Time         : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11          amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30          amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30          amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30          amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0           volume(b,0)=100        volume(a,0) $< volume(b,0)
volume(a,1)=11          volume(b,1)=100        volume(a,1) $< volume(b,1)
volume(a,2)=41          volume(b,2)=100        volume(a,2) $< volume(b,2)
volume(a,3)=71          volume(b,3)=100        volume(a,3) $< volume(b,3)
volume(a,4)=101         volume(b,4)=100        volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1+
Time         : 0.000
```

Pouring Water into Buckets on a Scale

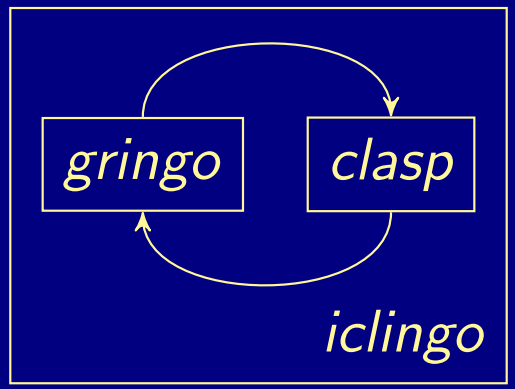
```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11          amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30          amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30          amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30          amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0           volume(b,0)=100        volume(a,0) $< volume(b,0)
volume(a,1)=11          volume(b,1)=100        volume(a,1) $< volume(b,1)
volume(a,2)=41          volume(b,2)=100        volume(a,2) $< volume(b,2)
volume(a,3)=71          volume(b,3)=100        volume(a,3) $< volume(b,3)
volume(a,4)=101         volume(b,4)=100        volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1+
Time         : 0.000
```

Outline

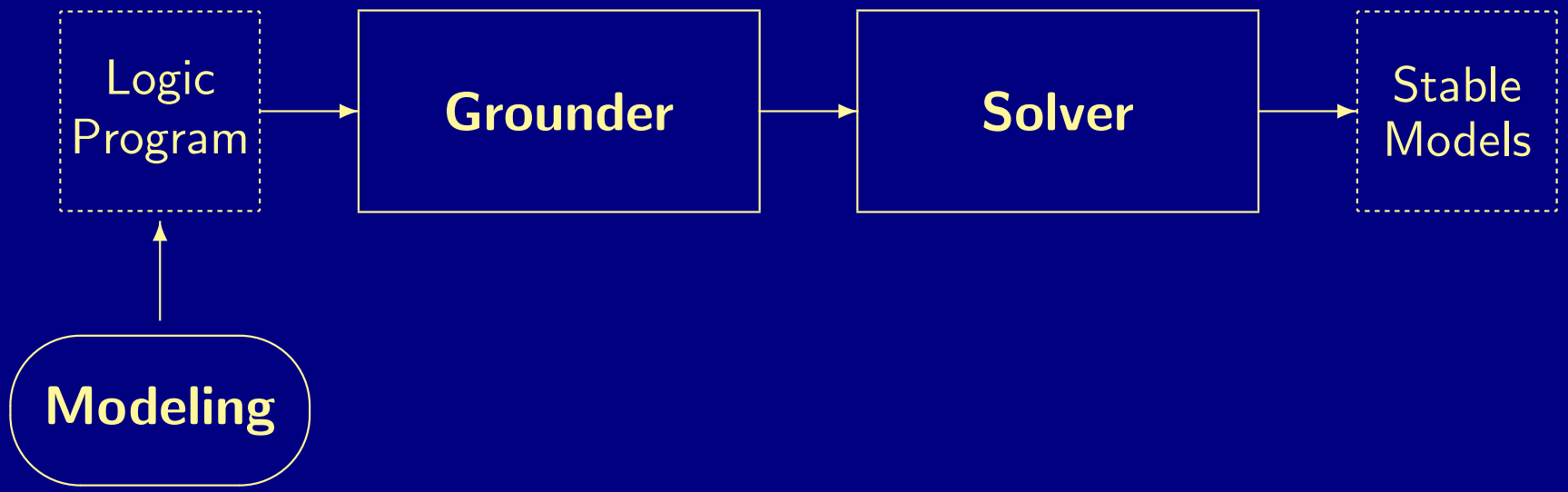
iclingo

- Incremental grounding and solving
- Offline solving in dynamic domains, like Automated Planning

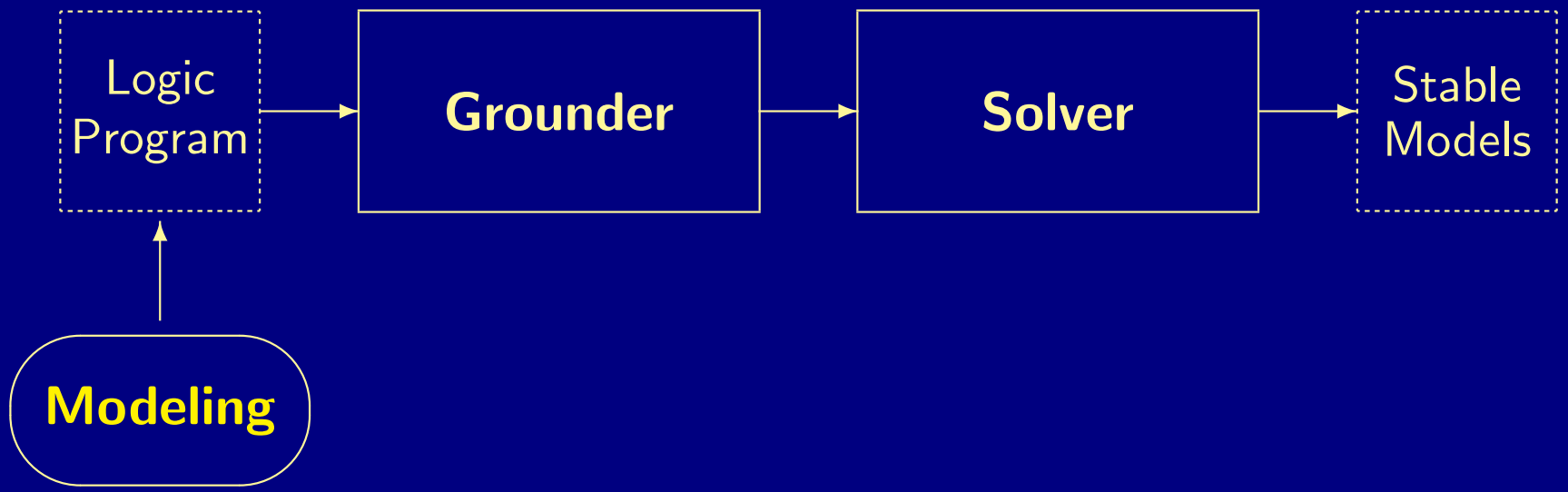
- Basic architecture of *iclingo*:



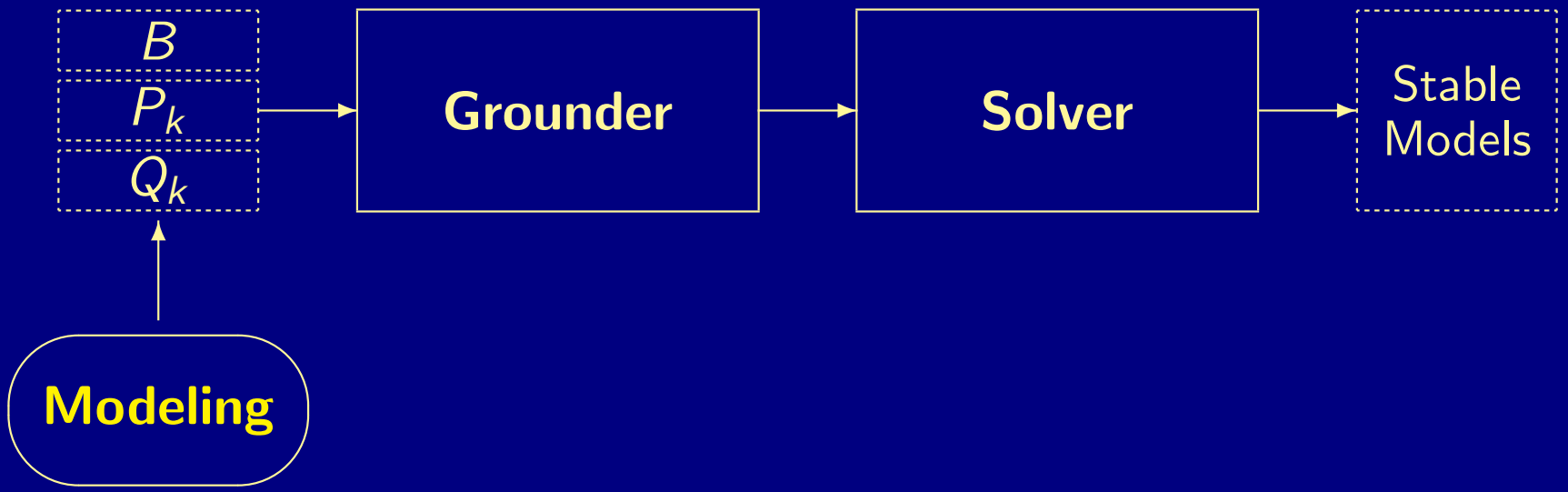
Incremental ASP Solving Process



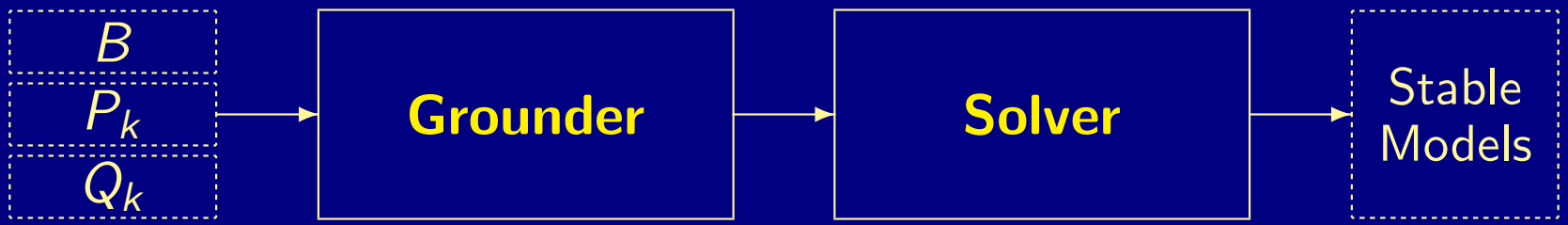
Incremental ASP Solving Process



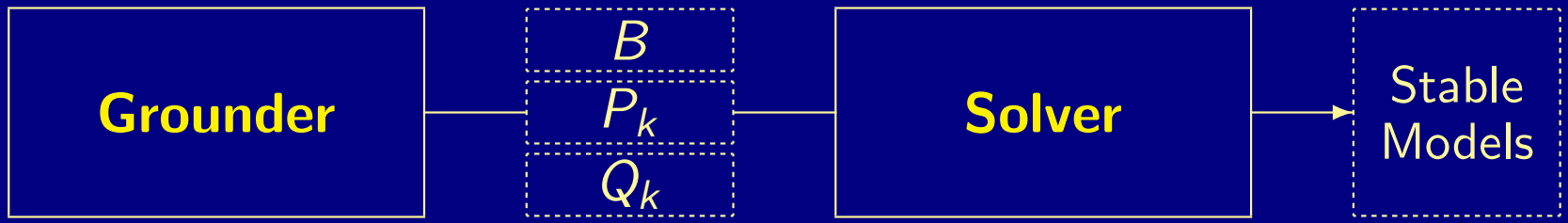
Incremental ASP Solving Process



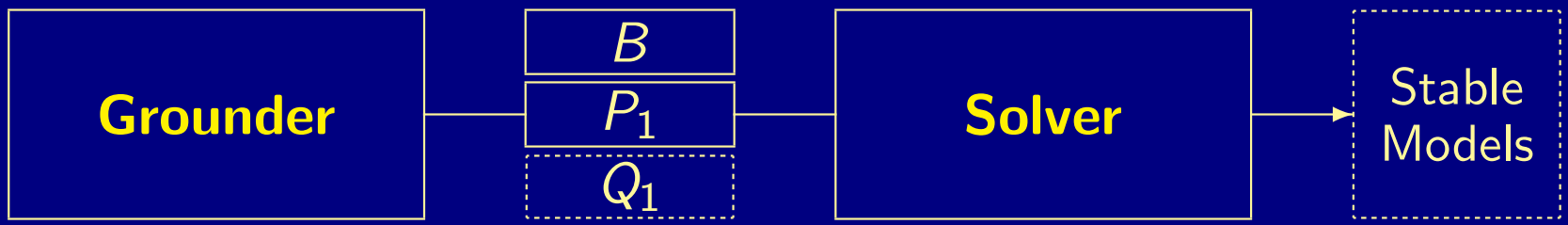
Incremental ASP Solving Process



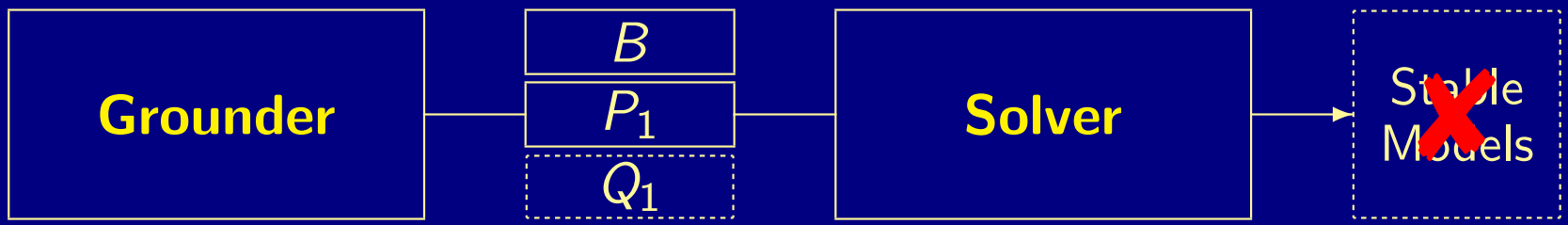
Incremental ASP Solving Process



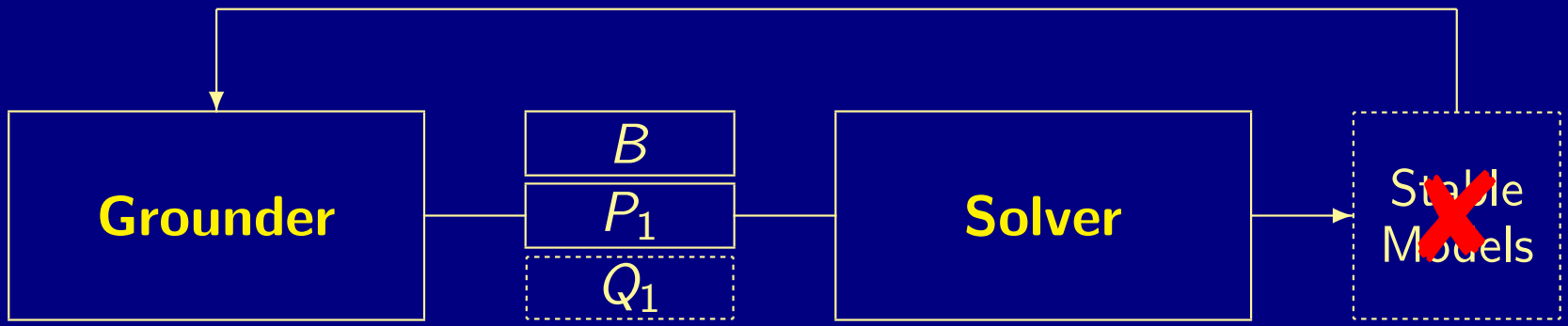
Incremental ASP Solving Process



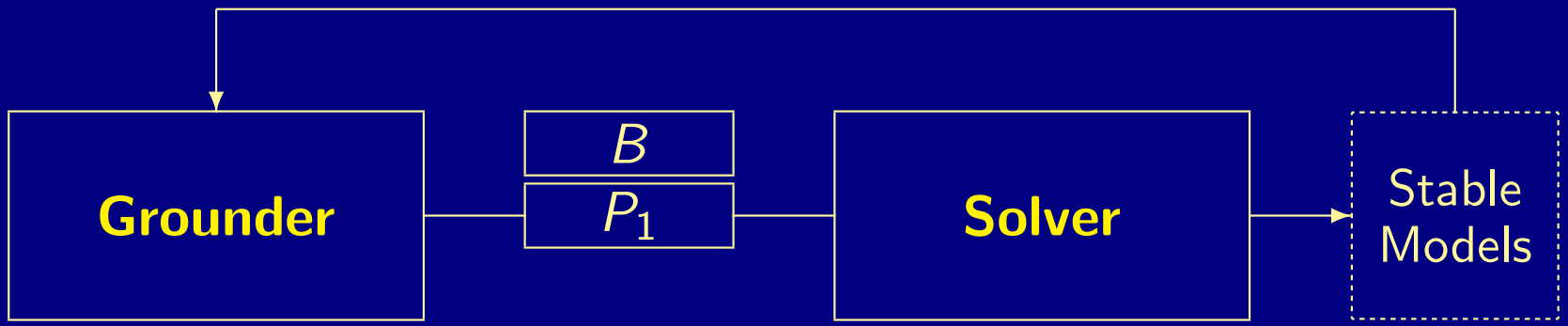
Incremental ASP Solving Process



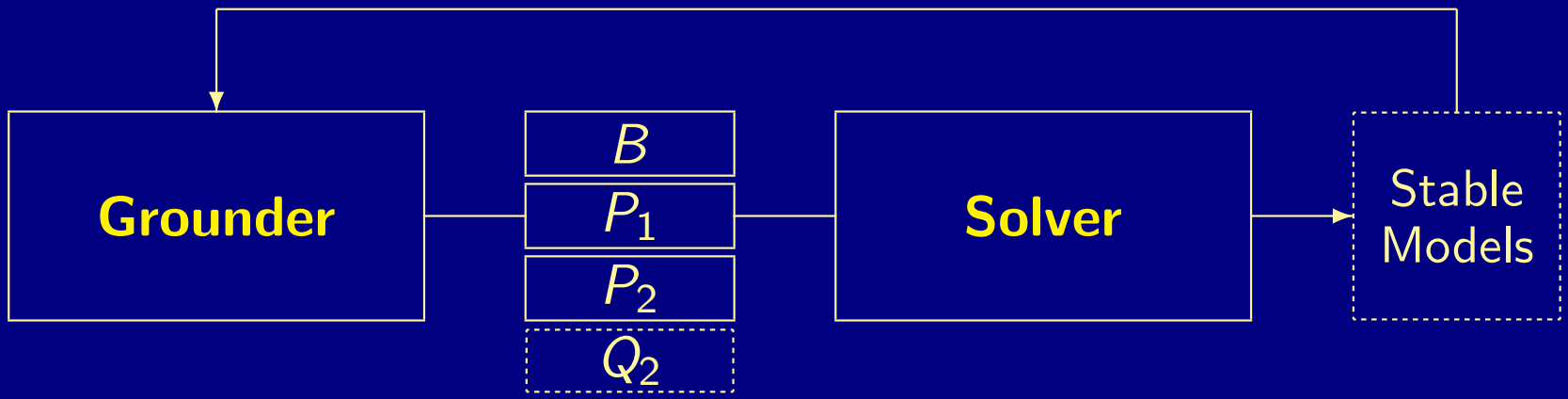
Incremental ASP Solving Process



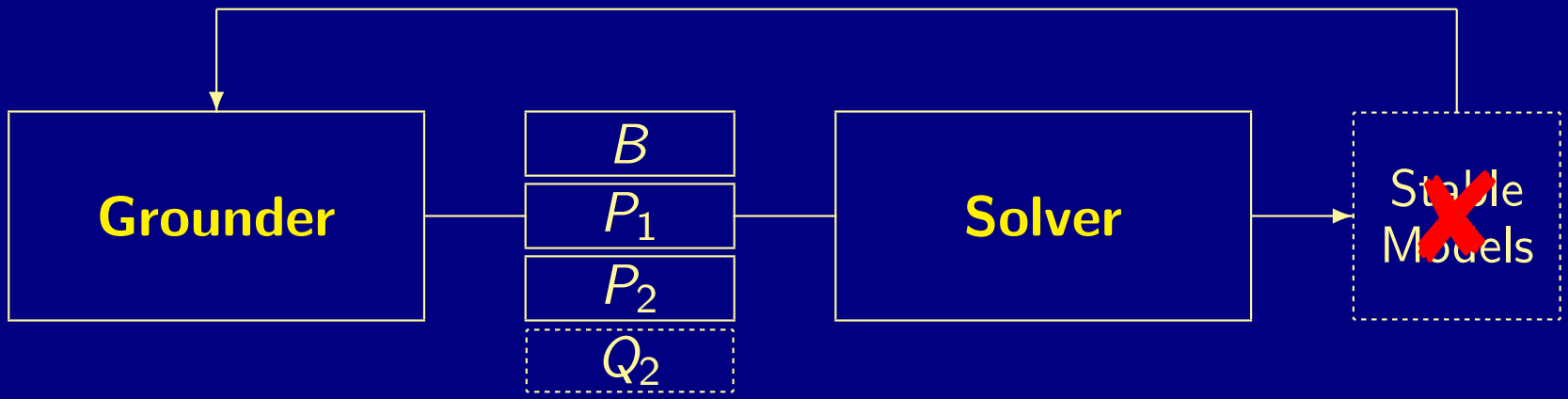
Incremental ASP Solving Process



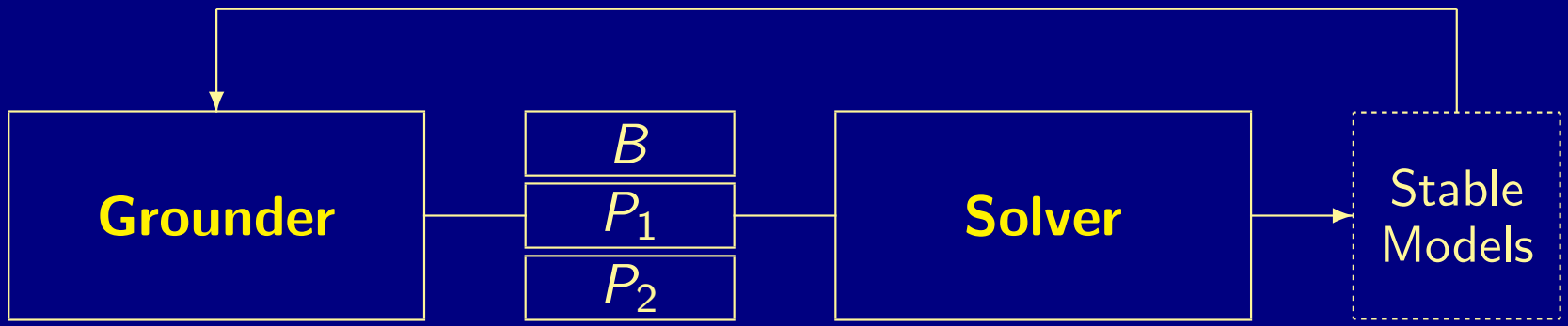
Incremental ASP Solving Process



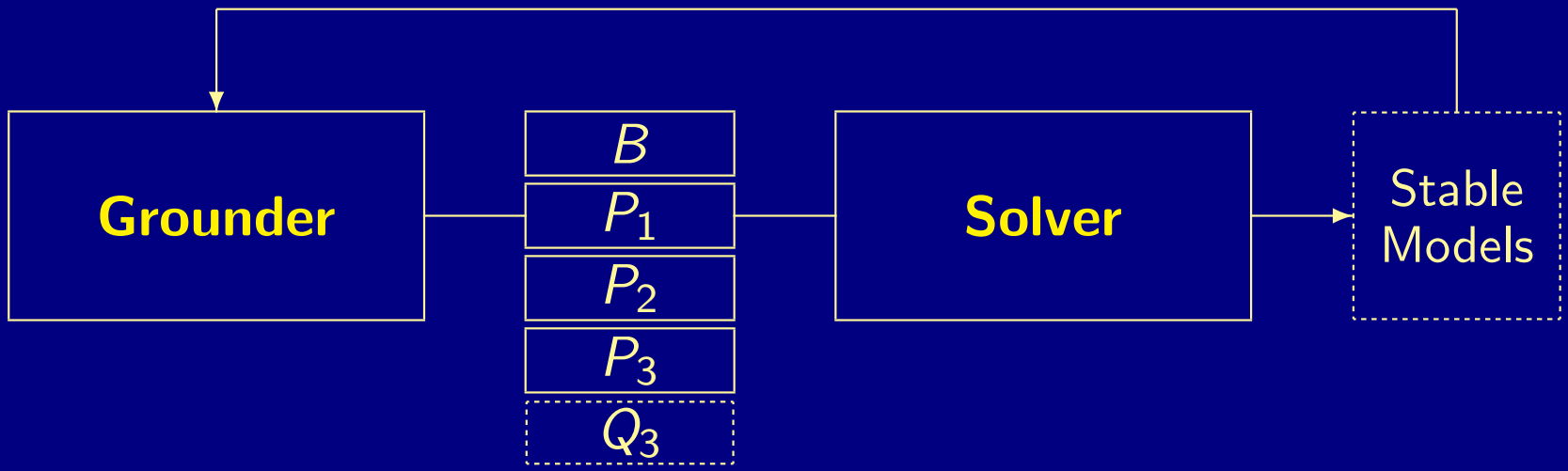
Incremental ASP Solving Process



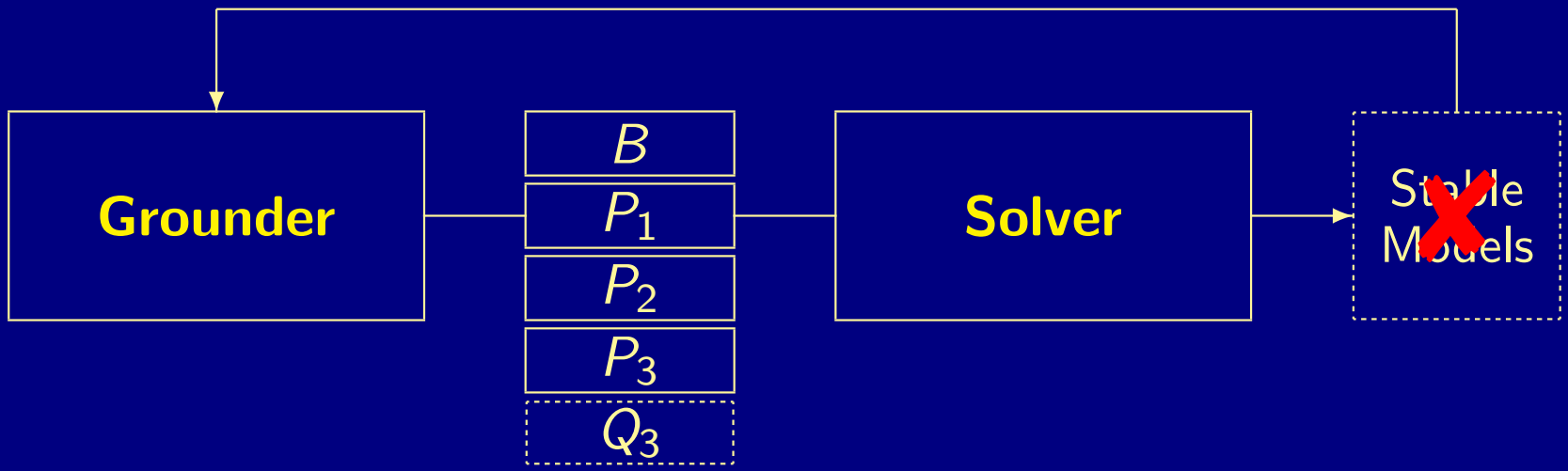
Incremental ASP Solving Process



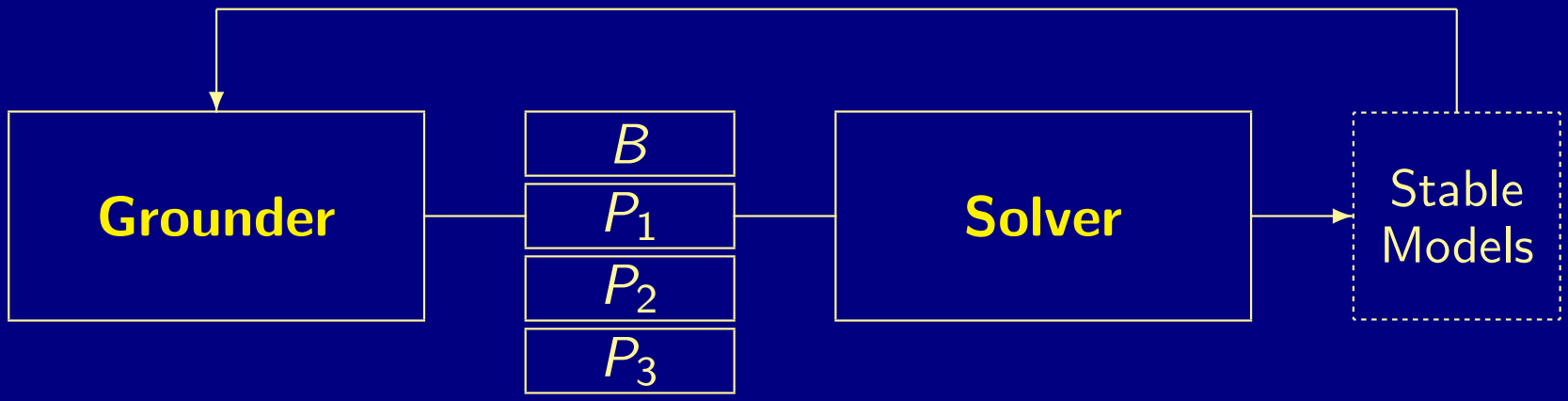
Incremental ASP Solving Process



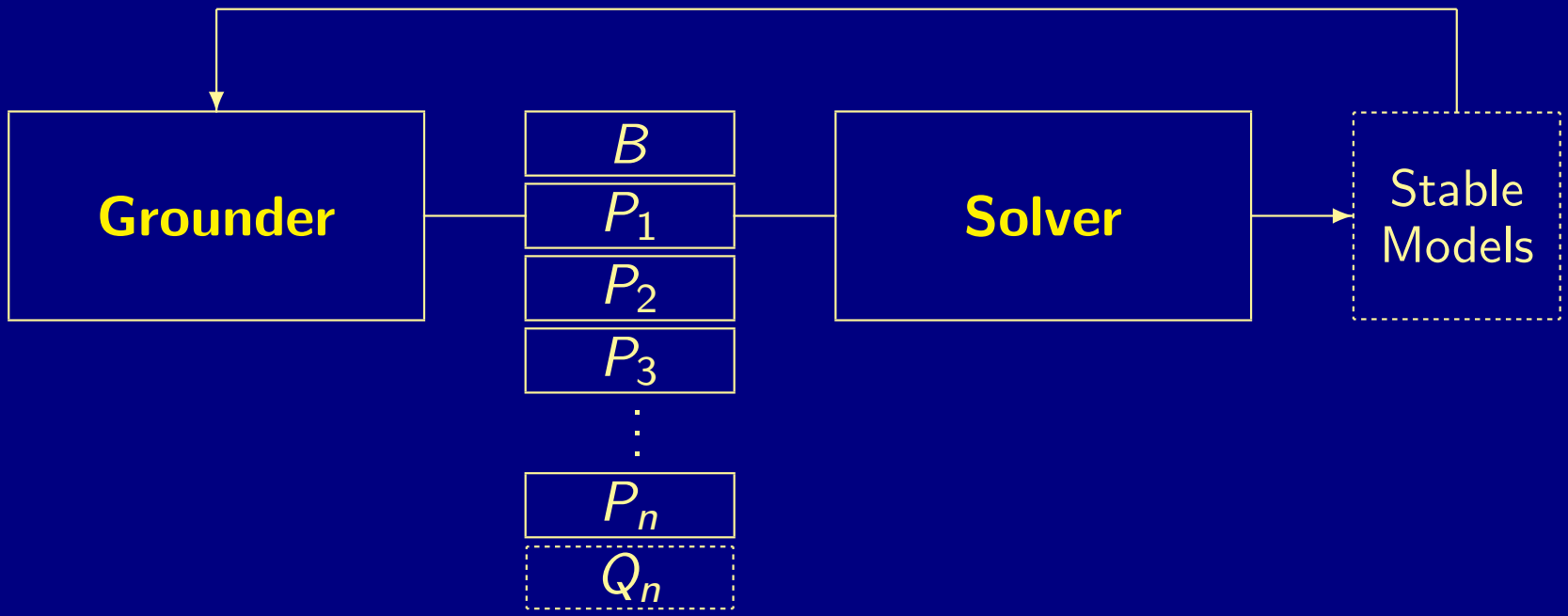
Incremental ASP Solving Process



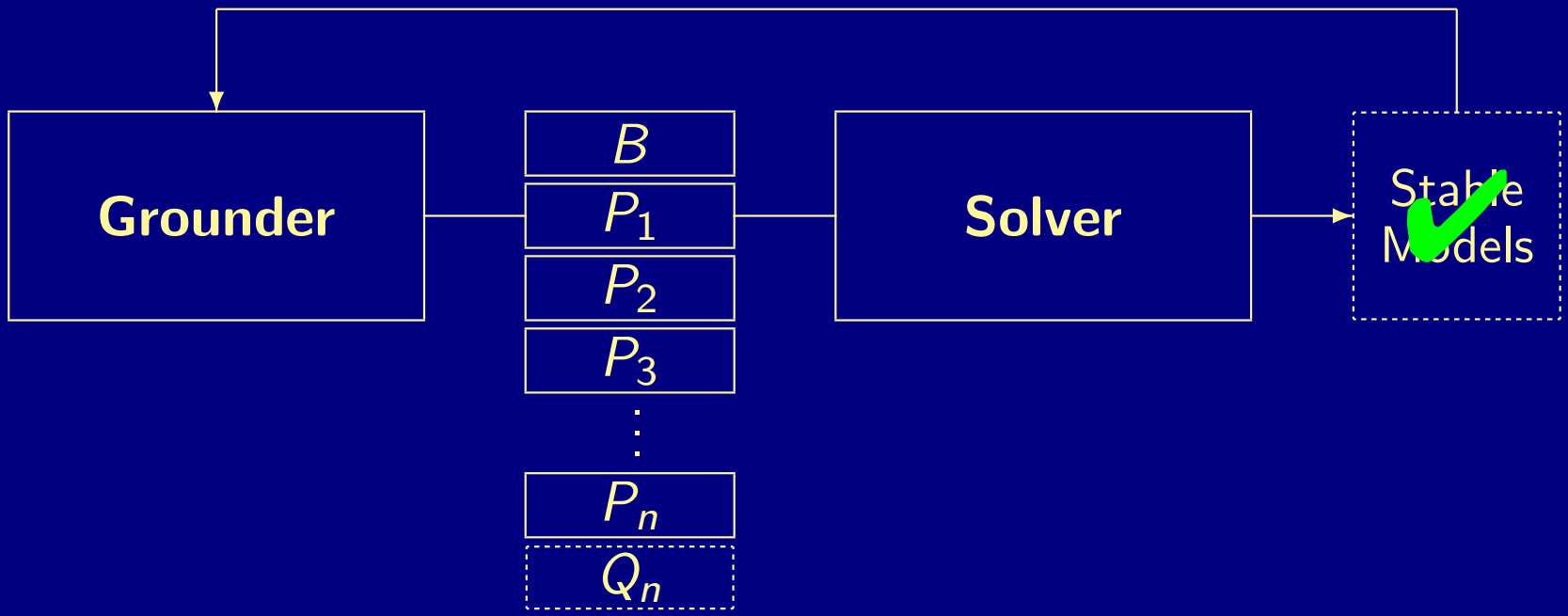
Incremental ASP Solving Process



Incremental ASP Solving Process



Incremental ASP Solving Process



Simplistic STRIPS Planning

```
#base.
fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

ocdel(F,t) :- occ(A,t), del(A,F).
holds(F,t) :- occ(A,t), add(A,F).
holds(F,t) :- holds(F,t-1), not ocdel(F,t).

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
#base.
fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

ocdel(F,t) :- occ(A,t), del(A,F).
holds(F,t) :- occ(A,t), add(A,F).
holds(F,t) :- holds(F,t-1), not ocdel(F,t).

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
#base.
fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

ocdel(F,t) :- occ(A,t), del(A,F).
holds(F,t) :- occ(A,t), add(A,F).
holds(F,t) :- holds(F,t-1), not ocdel(F,t).

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
#base.
fluent(p).     action(a).     action(b).       init(p).
fluent(q).        pre(a,p).      pre(b,q).
fluent(r).        add(a,q).      add(b,r).     query(r).
                  del(a,p).      del(b,q).

holds(P,0) :- init(P).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

ocdel(F,t) :- occ(A,t), del(A,F).
holds(F,t) :- occ(A,t), add(A,F).
holds(F,t) :- holds(F,t-1), not ocdel(F,t).

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
$ iclingo iplanning.lp

Answer: 1
occ(a,1) occ(b,2)
SATISFIABLE

Models       : 1
Total Steps  : 2
Time         : 0.000
```

Simplistic STRIPS Planning

```
$ iclingo iplanning.lp

Answer: 1
occ(a,1) occ(b,2)
SATISFIABLE

Models       : 1
Total Steps  : 2
Time         : 0.000
```

Simplistic STRIPS Planning

```
$ iclingo iplanning.lp --istats

============== step 1 ==============

Models   : 0
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 27
Choices  : 0
Conflicts: 0
============== step 2 ==============
Answer: 1
occ(a,1) occ(b,2)

Models   : 1
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 16
Choices  : 0
Conflicts: 0
============== Summary ==============
SATISFIABLE

Models      : 1
Total Steps : 2
Time        : 0.000
```

Simplistic STRIPS Planning

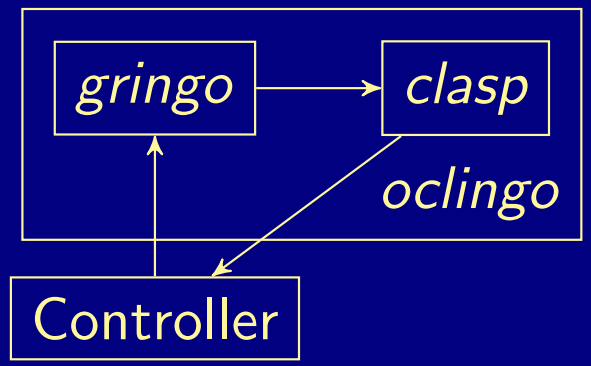
```
$ iclingo iplanning.lp --istats

============== step 1 ==============

Models   : 0
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 27
Choices  : 0
Conflicts: 0
============== step 2 ==============
Answer: 1
occ(a,1) occ(b,2)

Models   : 1
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 16
Choices  : 0
Conflicts: 0
============== Summary ==============
SATISFIABLE

Models      : 1
Total Steps : 2
Time        : 0.000
```

Outline

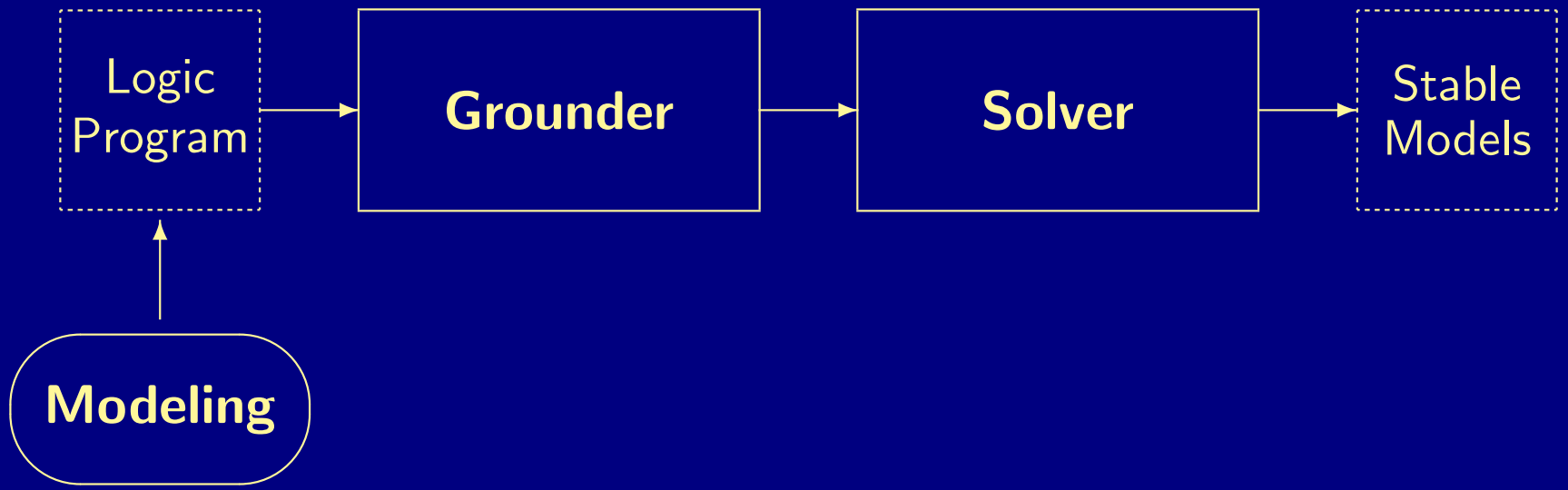
oclingo

- Reactive grounding and solving
- Online solving in dynamic domains, like Robotics

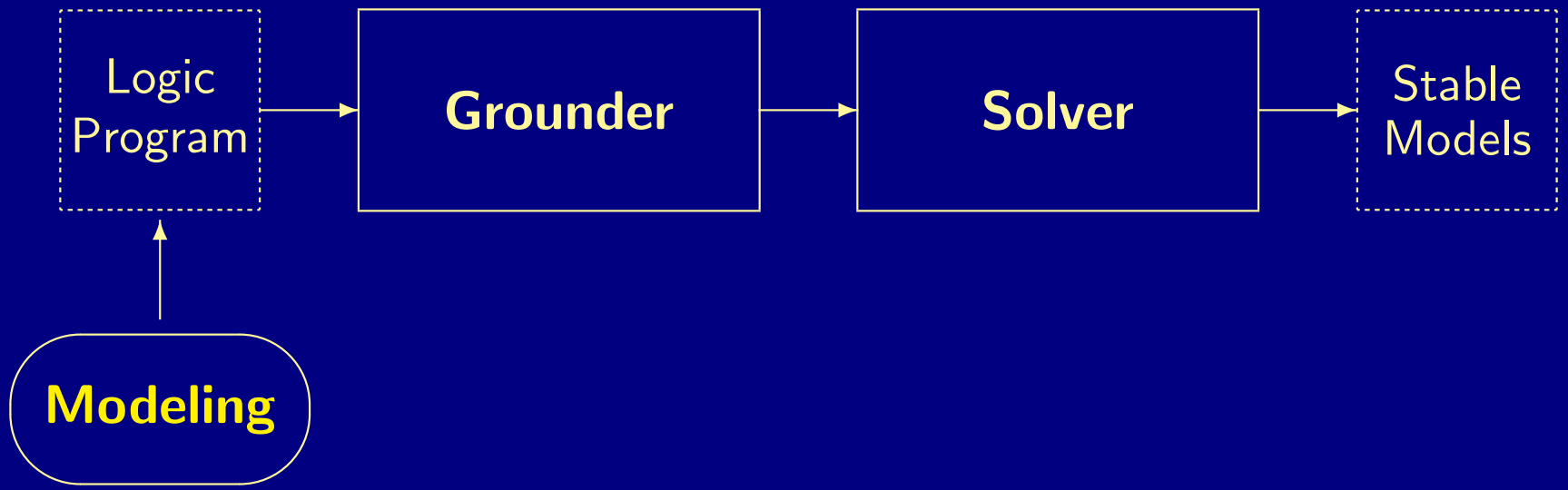
- Basic architecture of *oclingo*:



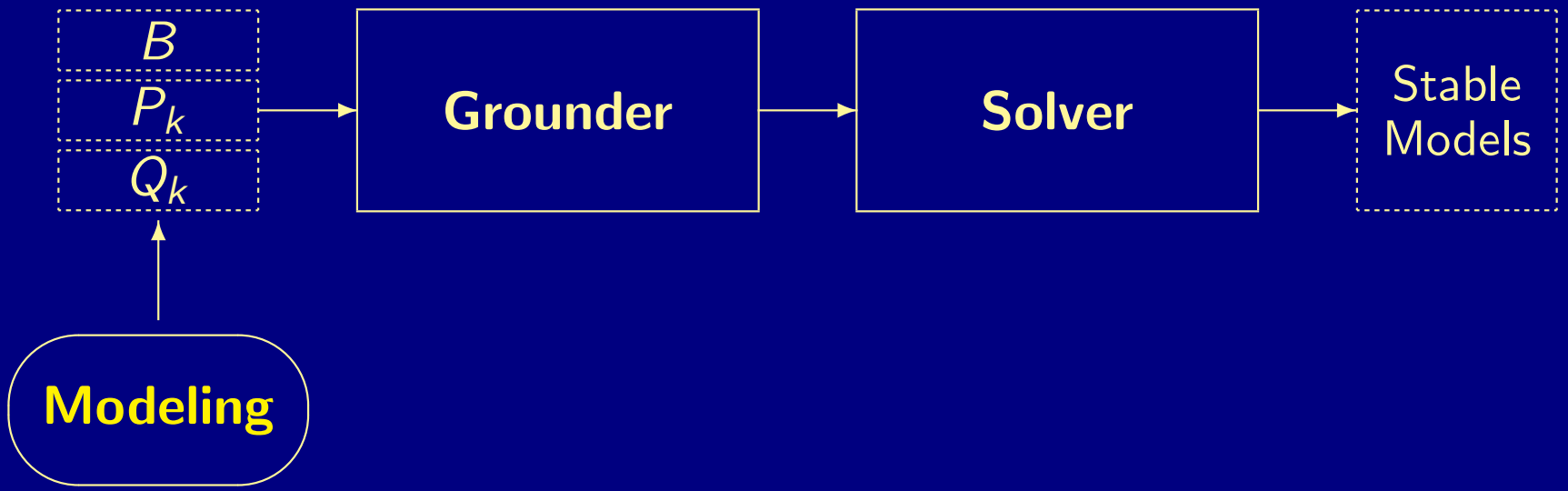
Reactive ASP Solving Process



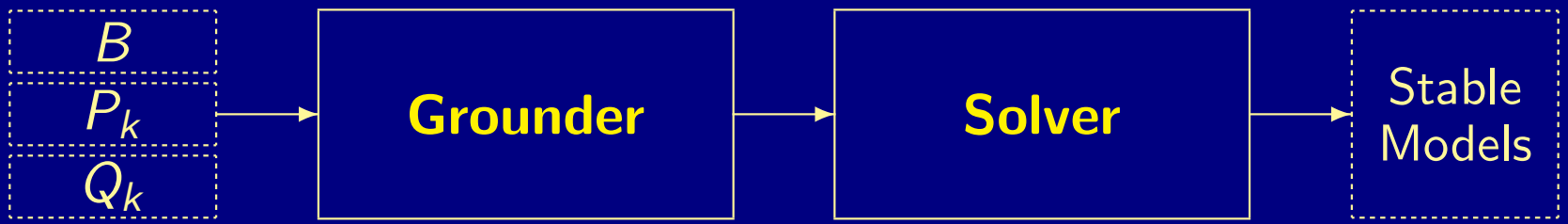
Reactive ASP Solving Process



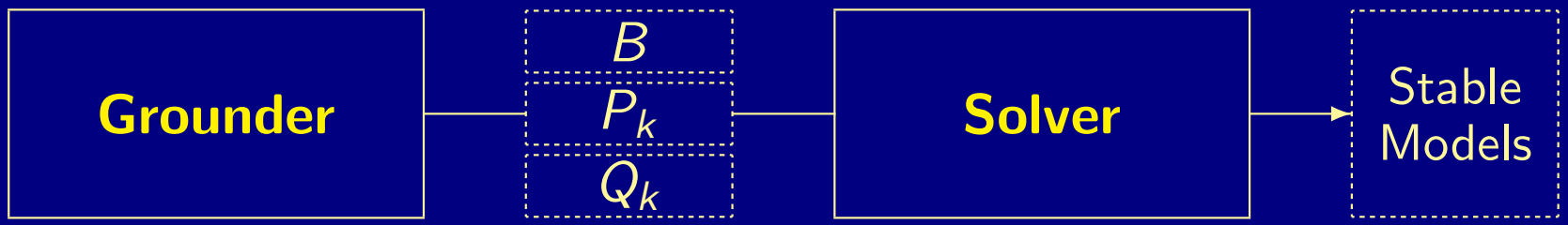
Reactive ASP Solving Process



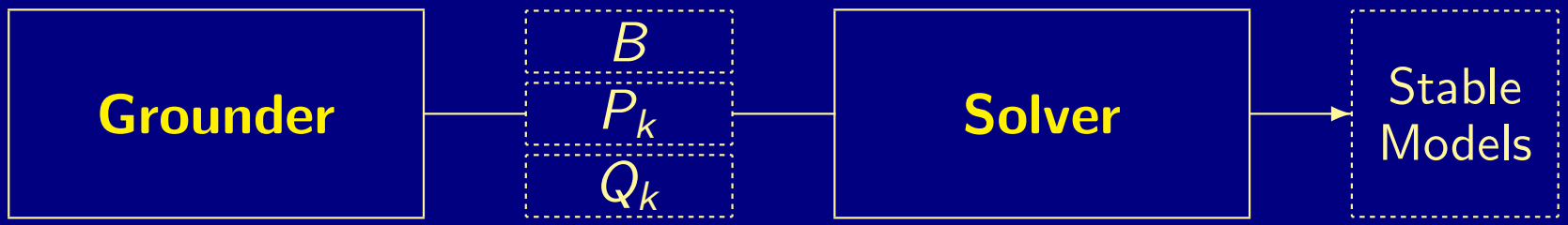
Reactive ASP Solving Process



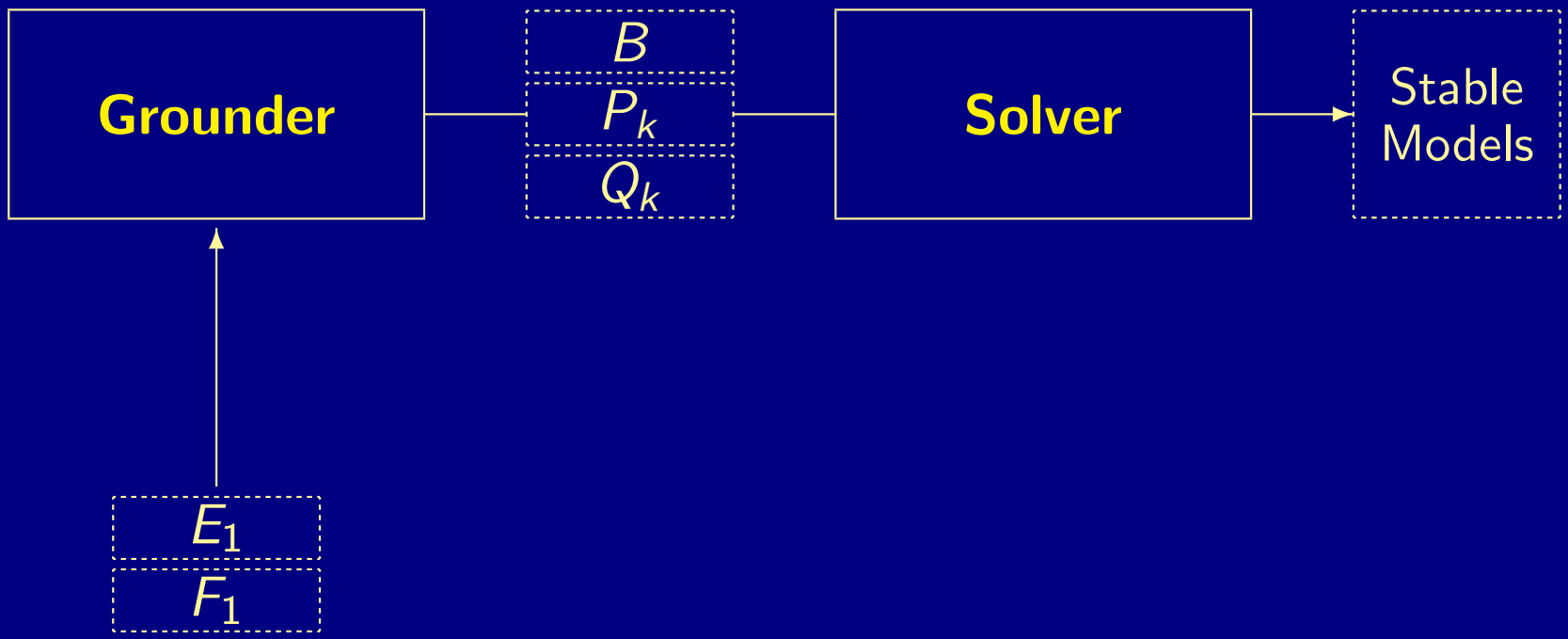
Reactive ASP Solving Process



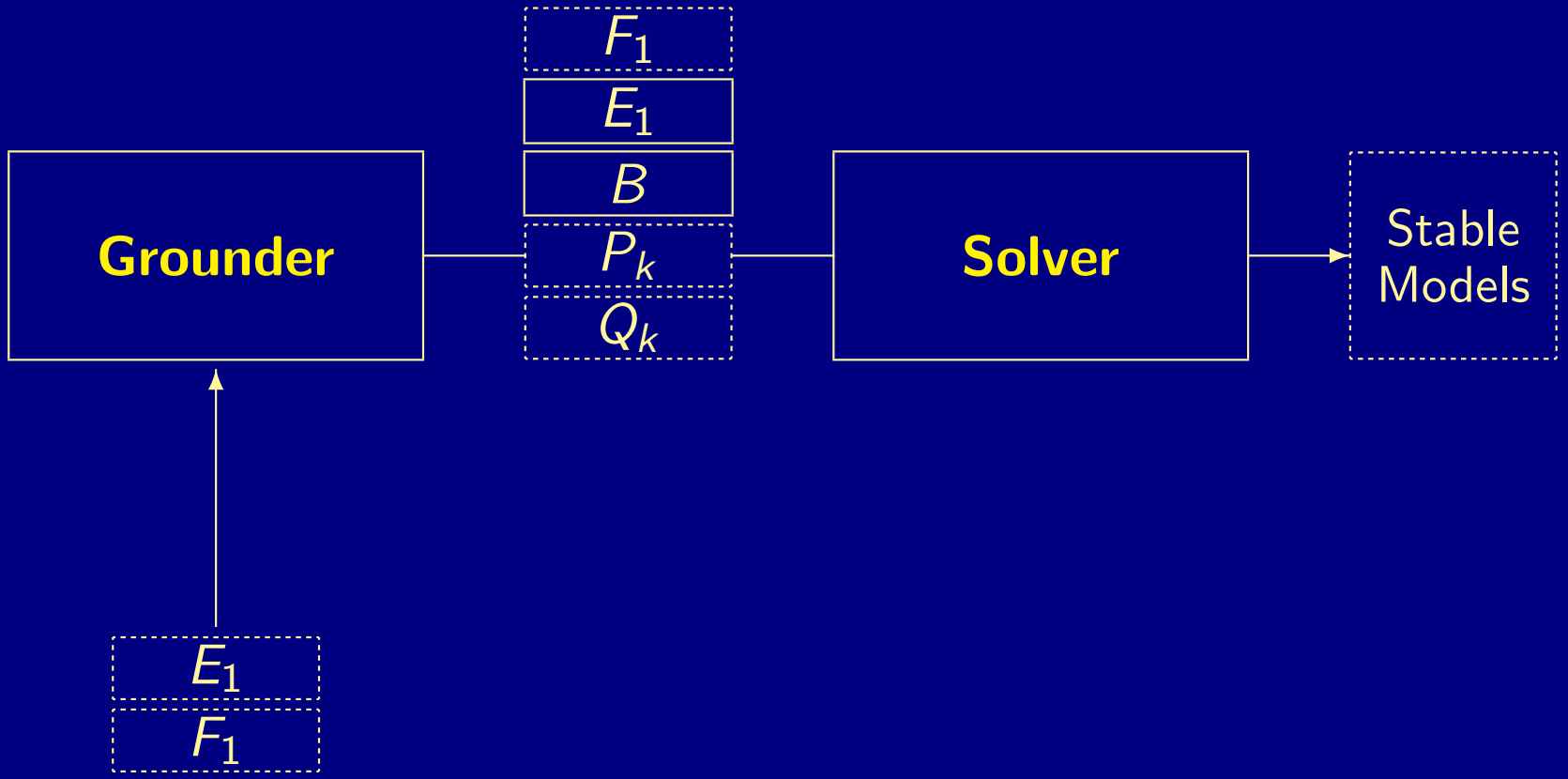
Reactive ASP Solving Process



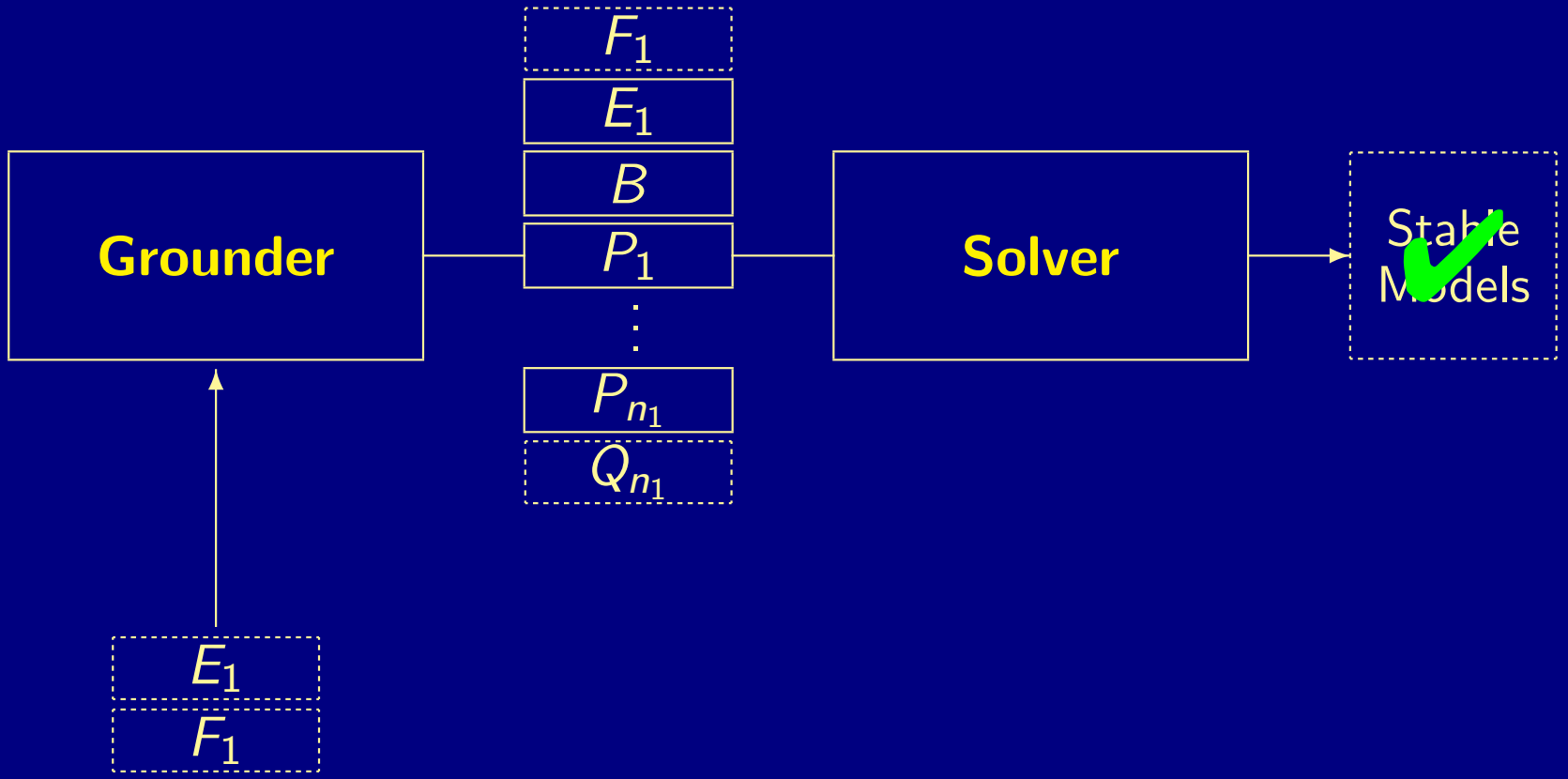
Reactive ASP Solving Process



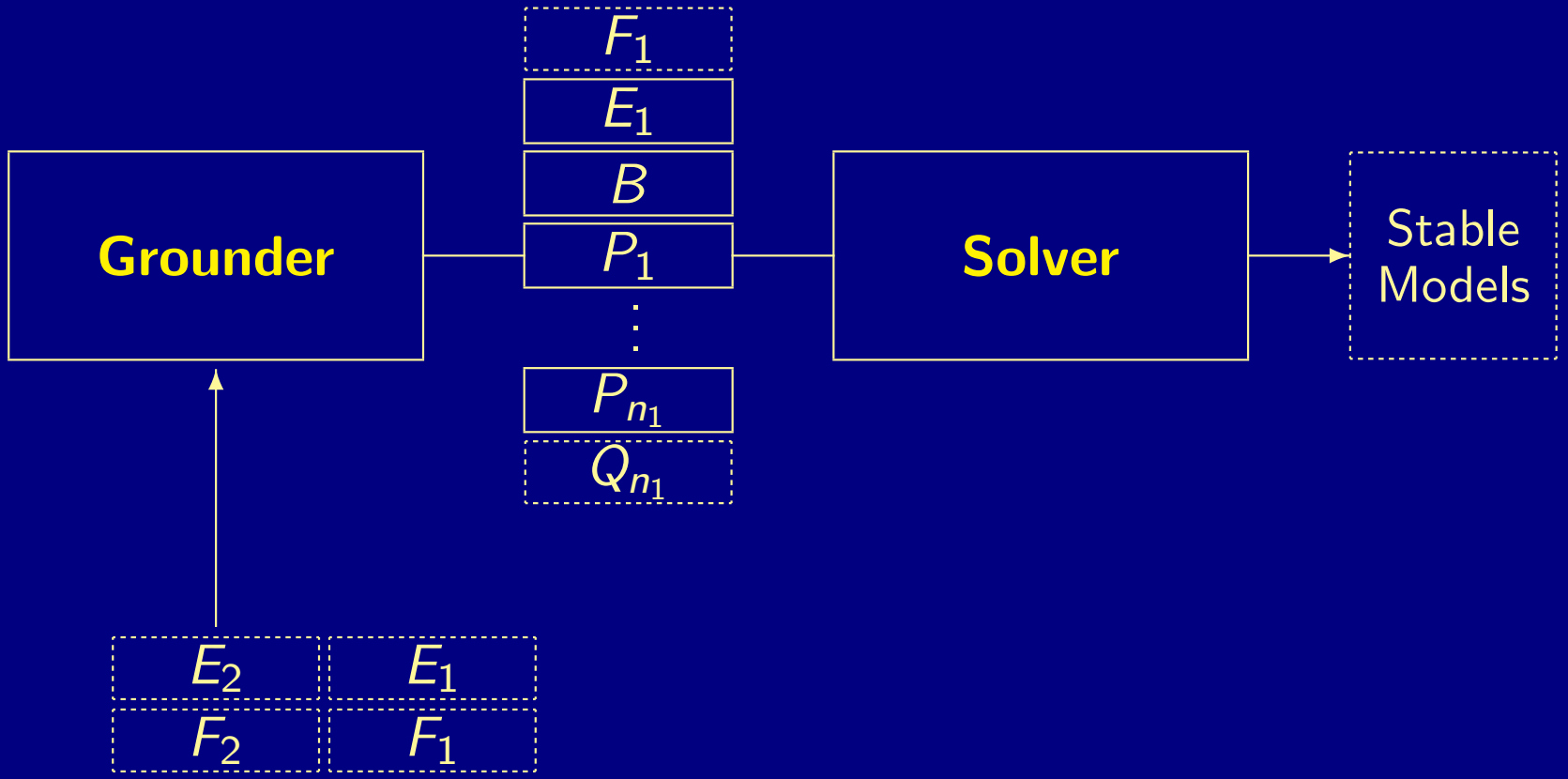
Reactive ASP Solving Process



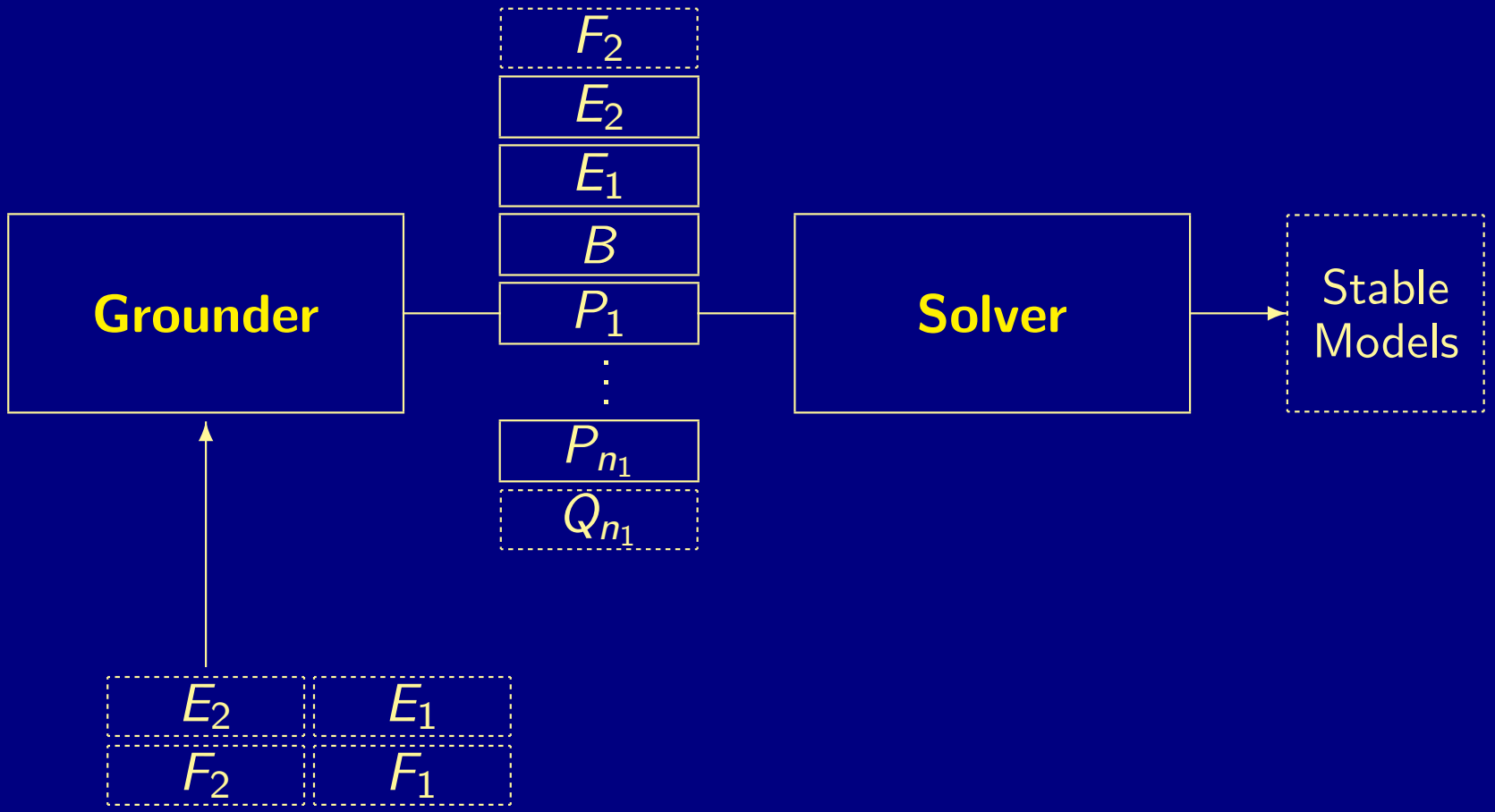
Reactive ASP Solving Process



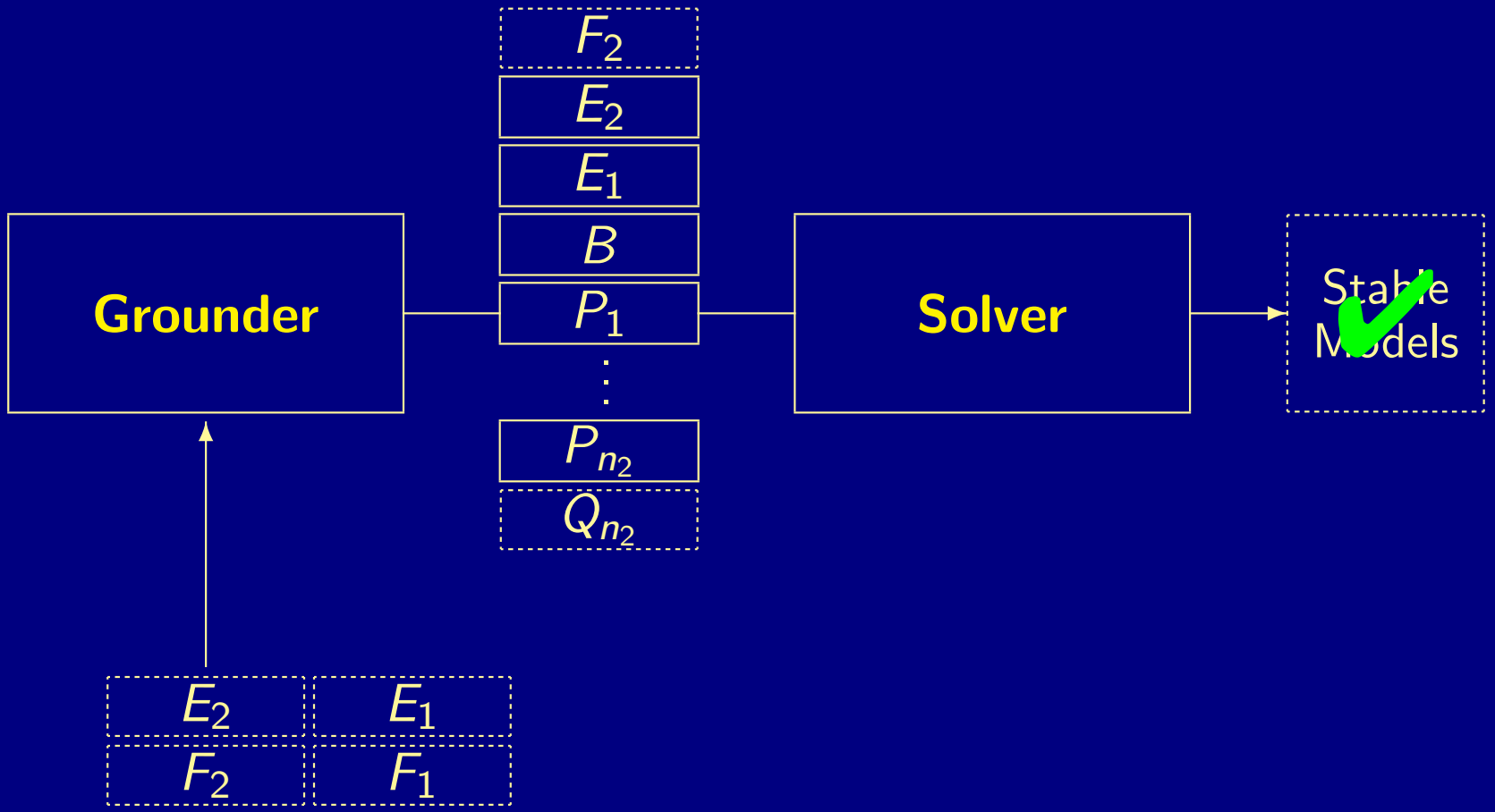
Reactive ASP Solving Process



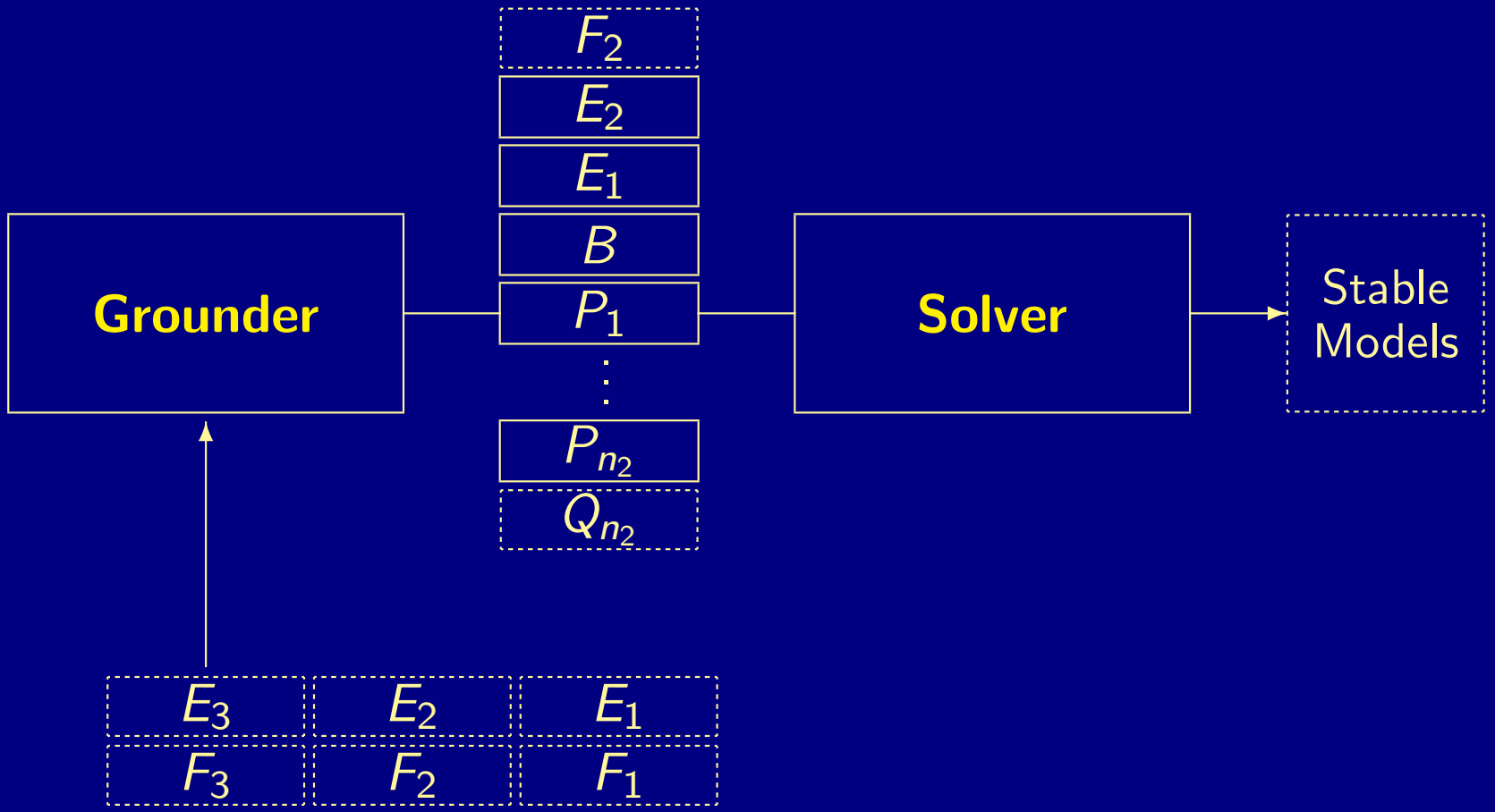
Reactive ASP Solving Process



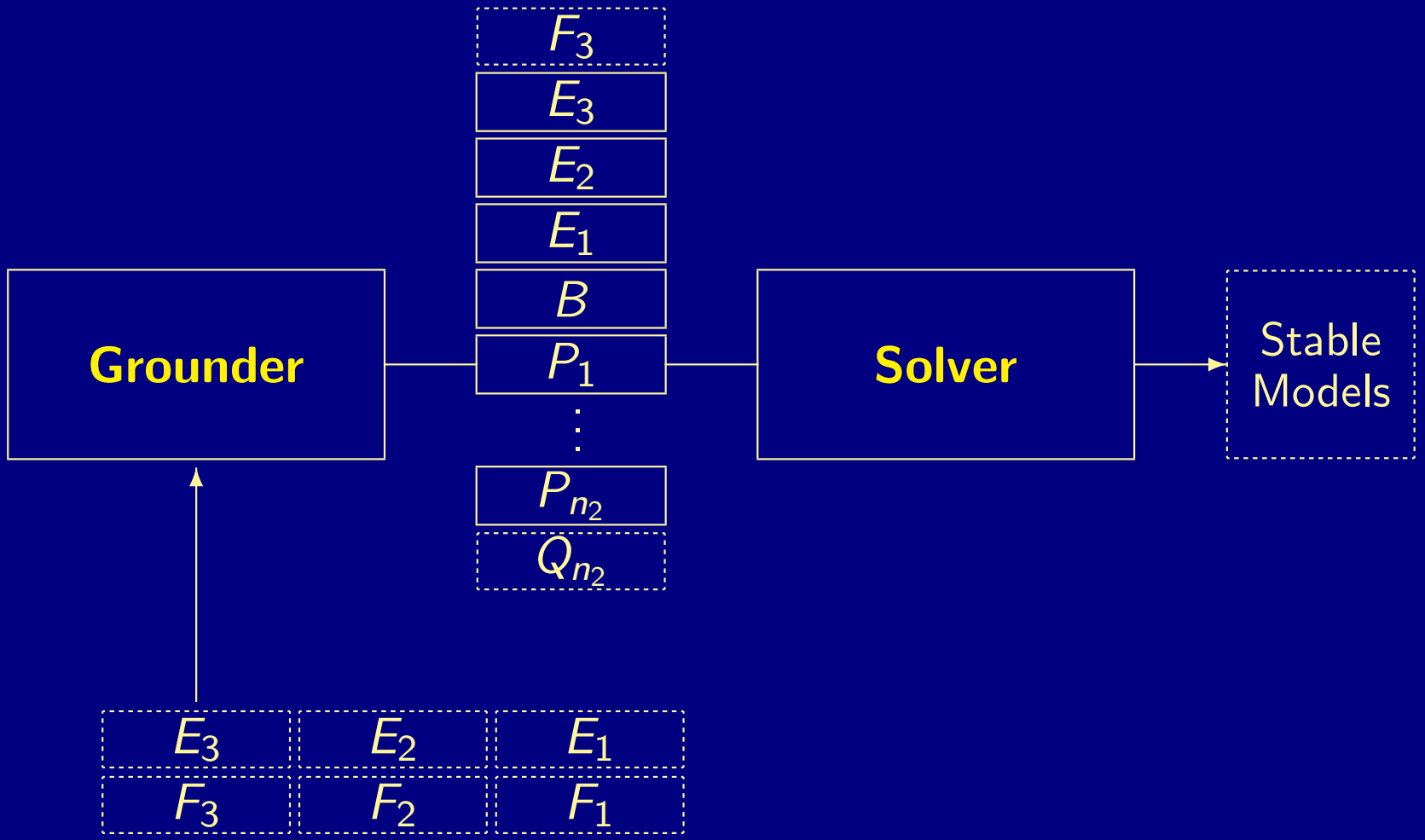
Reactive ASP Solving Process



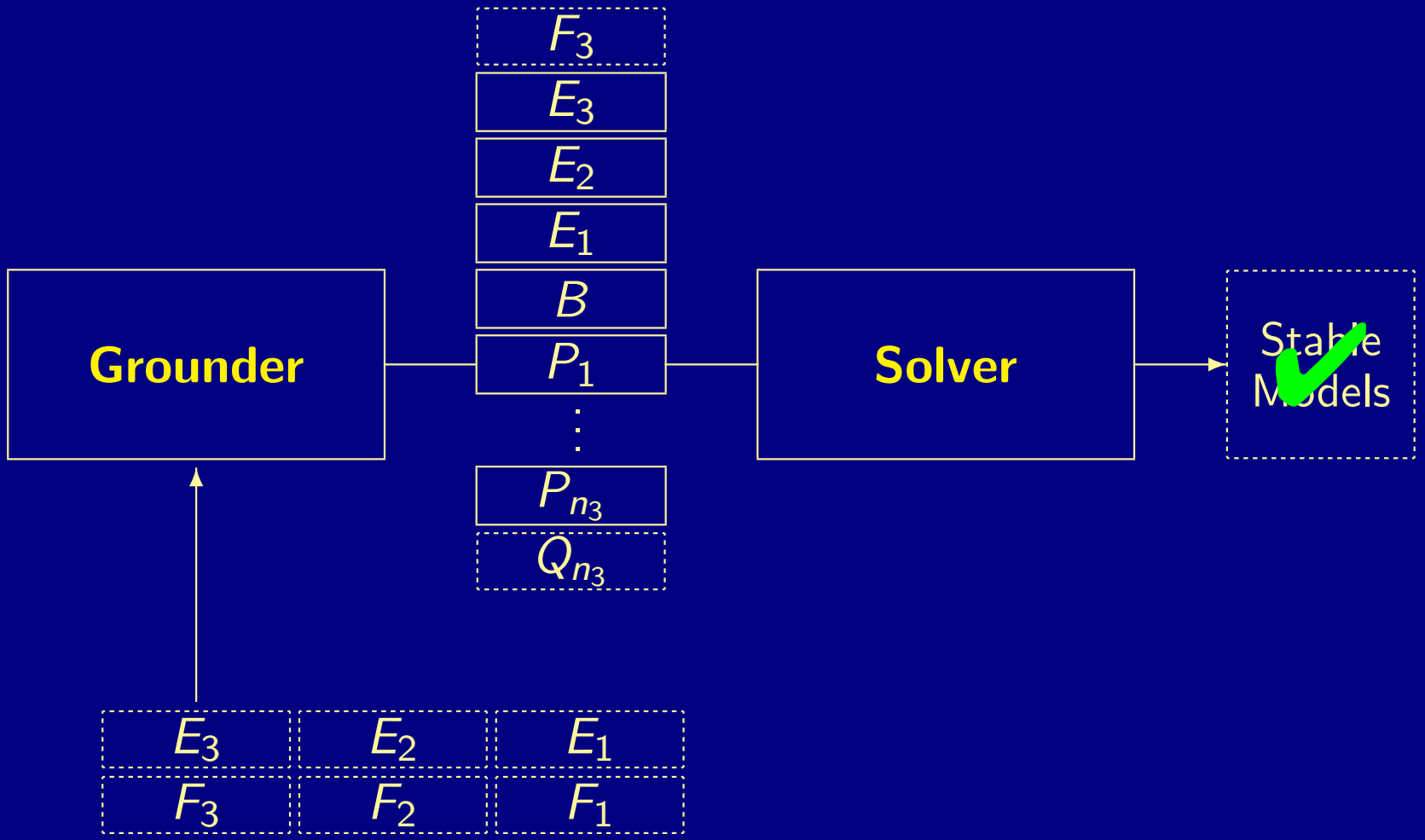
Reactive ASP Solving Process



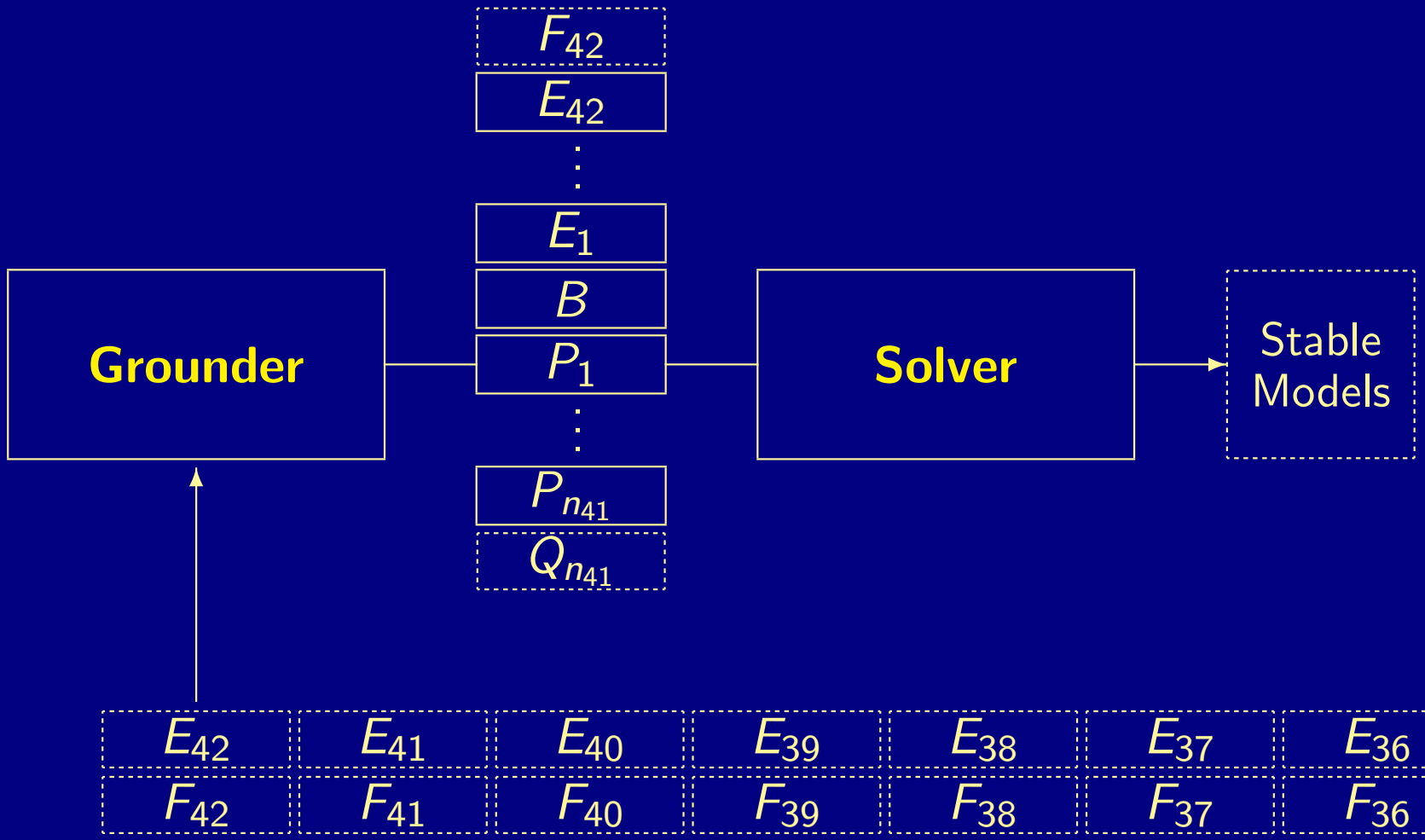
Reactive ASP Solving Process



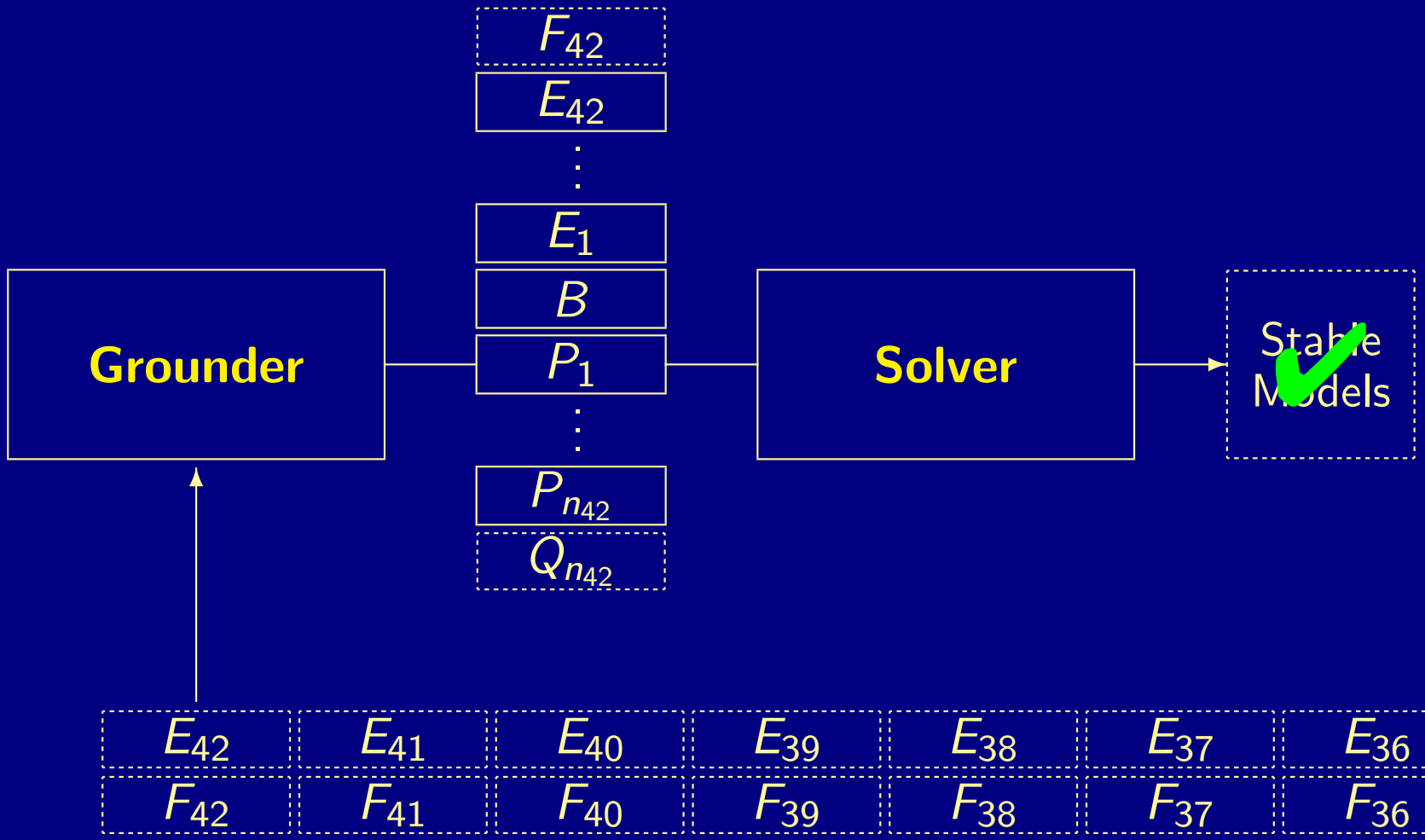
Reactive ASP Solving Process



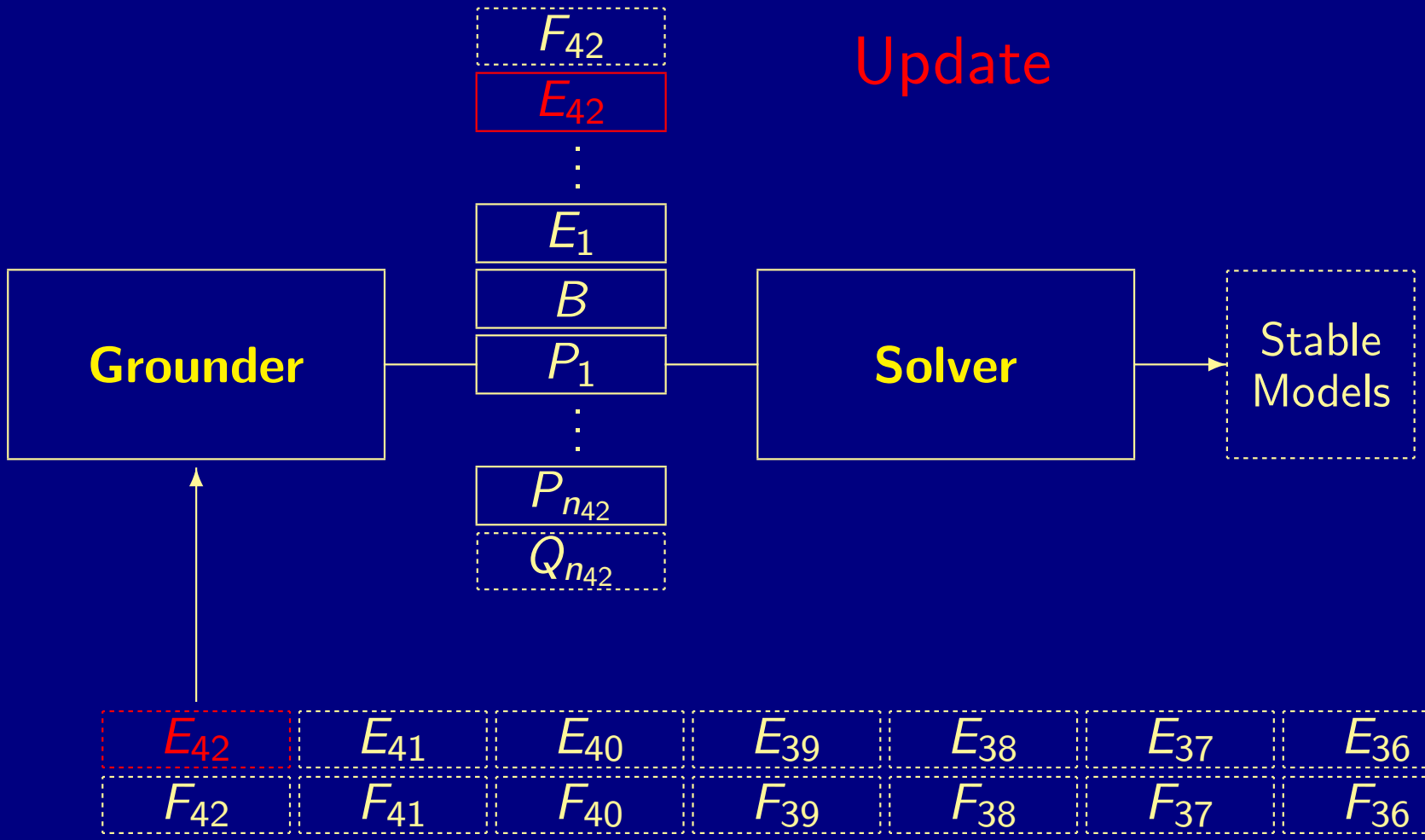
Reactive ASP Solving Process



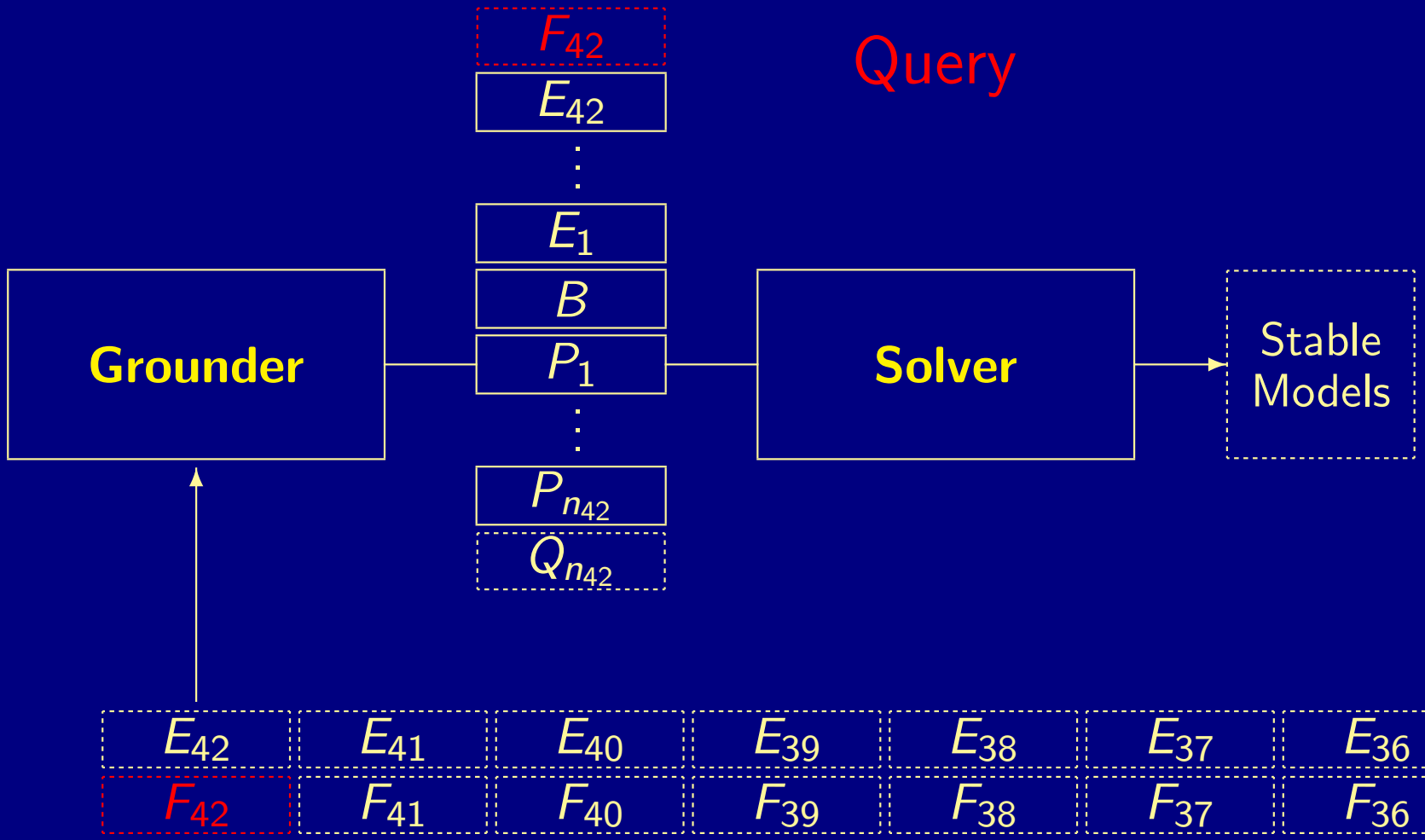
Reactive ASP Solving Process



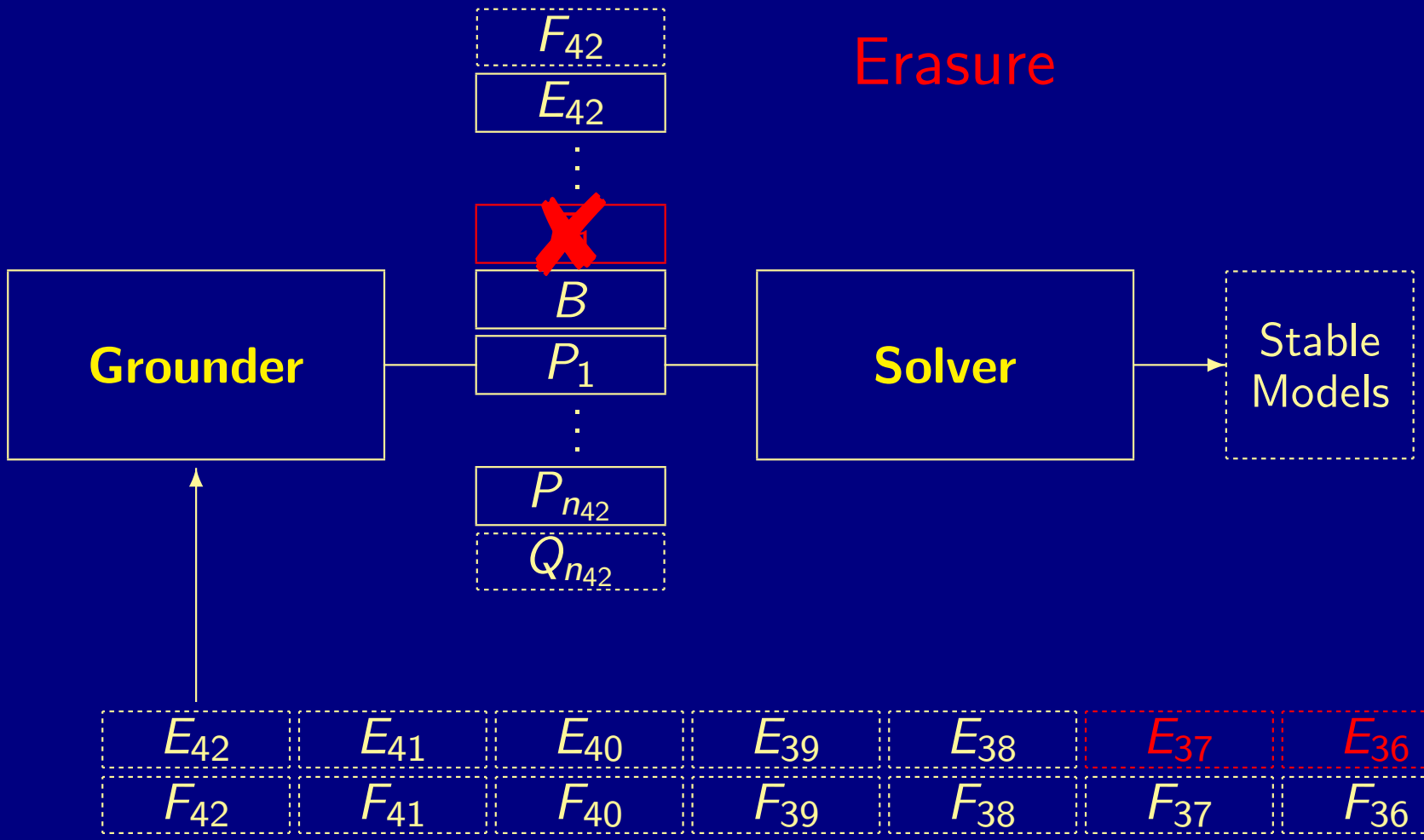
Reactive ASP Solving Process



Reactive ASP Solving Process



Reactive ASP Solving Process



Elevator Control

```
#base.
floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) : floor(F).
1 { atFloor(F-1;F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t),     floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.
:- not goal(t).
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Outline

Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.sourceforge.net`
  - ASP, CASC, MISC, PB, and SAT competitions
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

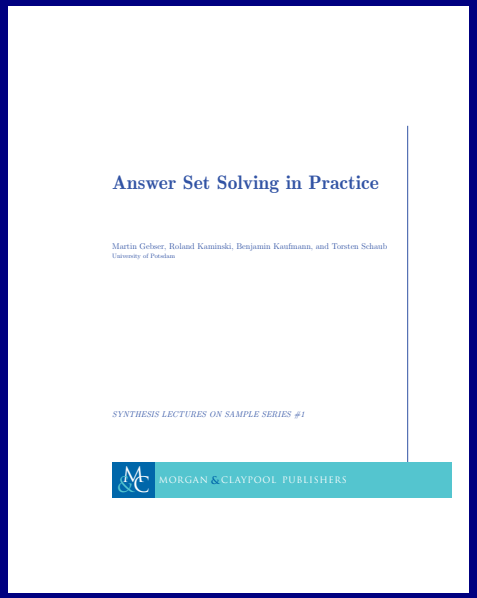
Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.sourceforge.net`
  - ASP, CASC, MISC, PB, and SAT competitions
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

ASP = DB+LP+KR+SAT

The (forthcoming) Potassco Book

1. Motivation
2. Introduction
3. Basic modeling
4. Grounding
5. Characterizations
6. Solving
7. Systems
8. Advanced modeling
9. Conclusions

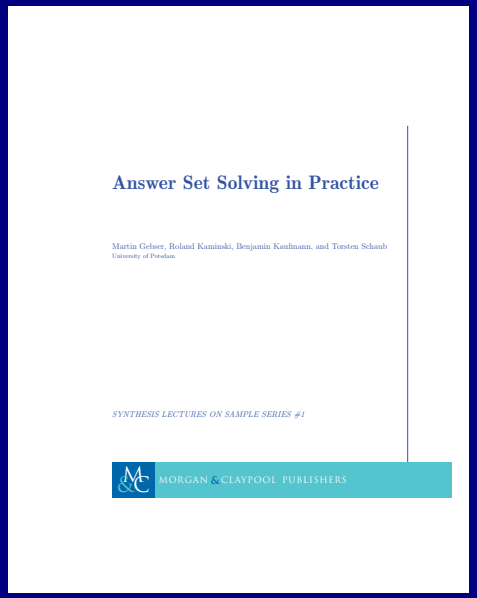


Visit the Potassco project !

- Sourceforge `http://potassco.sourceforge.net`
- Google+ `https://plus.google.com/102537396696345299260`

The (forthcoming) Potassco Book

1. Motivation
2. Introduction
3. Basic modeling
4. Grounding
5. Characterizations
6. Solving
7. Systems
8. Advanced modeling
9. Conclusions



Visit the Potassco project !

- Sourceforge `http://potassco.sourceforge.net`
- Google+ `https://plus.google.com/102537396696345299260`