Stream Reasoning with Answer Set Programming

— Preliminary Report —

M. Gebser T. Grote R. Kaminski P. Obermeier*
O. Sabuncu T. Schaub

University of Potsdam, Germany

*DERI Galway, Ireland

Outline

Outline

Motivation

- Answer Set Programming (ASP) integrates
  1. Concise high-level (first-order) specification
  2. Powerful low-level (propositional) reasoning

↪ Two-phase computations

  1. Instantiation (*gringo*)
  2. Search (*clasp*)

- Stream processing concerned with
  1. High-throughput
  2. Continuous data

- How use ASP for knowledge-intense stream reasoning?
  1. Handle emerging/expiring data upon instantiation
  2. Reuse previously learned information upon search

- Knowledge-intense stream reasoning can benefit from
  1. High automation
  2. Computational power

  offered by ASP systems

Motivation

- Answer Set Programming (ASP) integrates
  1. Concise high-level (first-order) specification
  2. Powerful low-level (propositional) reasoning

↪ Two-phase computations

  1. Instantiation (*gringo*)
  2. Search (*clasp*)

- Stream processing concerned with
  1. High-throughput
  2. Continuous data
- How use ASP for knowledge-intense stream reasoning?
  1. Handle emerging/expiring data upon instantiation
  2. Reuse previously learned information upon search
- Knowledge-intense stream reasoning can benefit from
  1. High automation
  2. Computational power

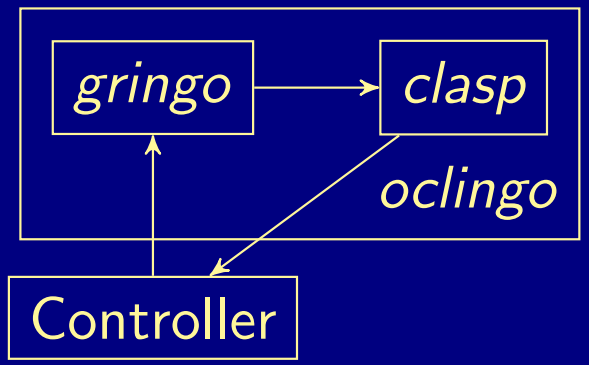
  offered by ASP systems

Outline

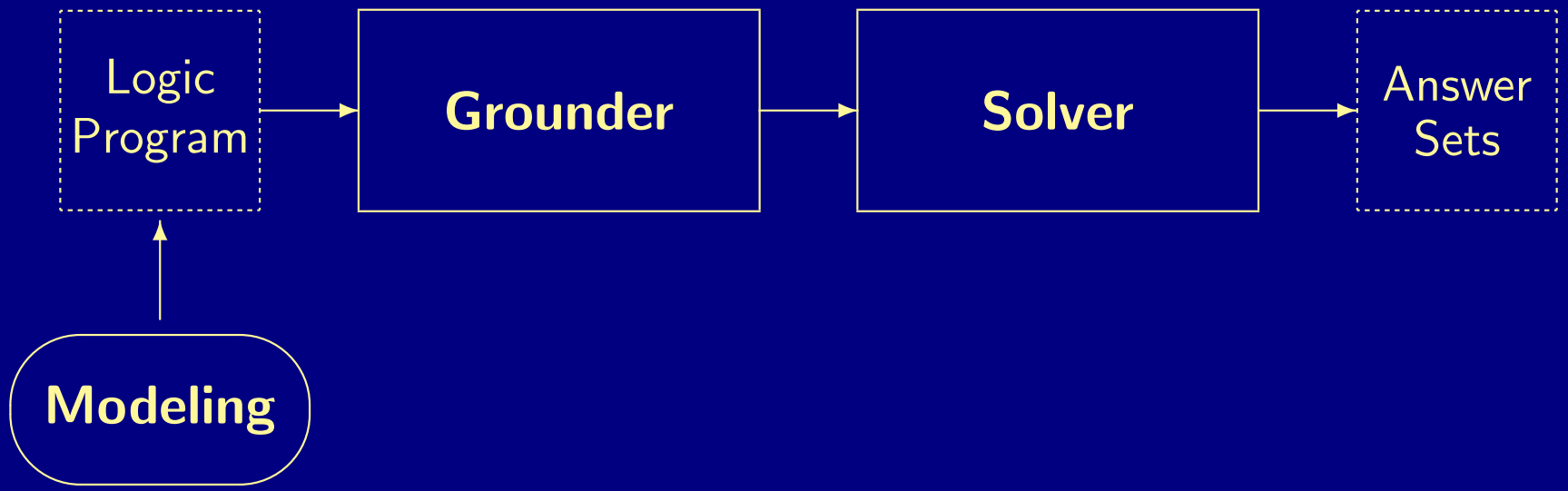
oclingo

- Reactive grounding and solving
- Online solving in dynamic domains, like Robotics

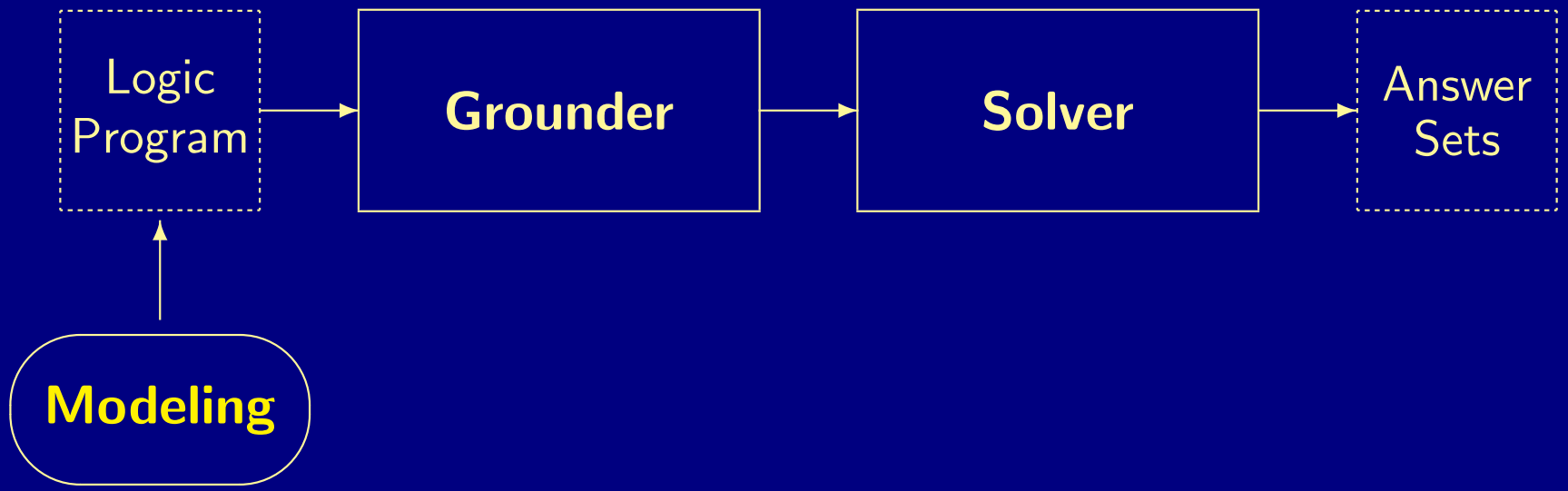
- Basic architecture of *oclingo*:



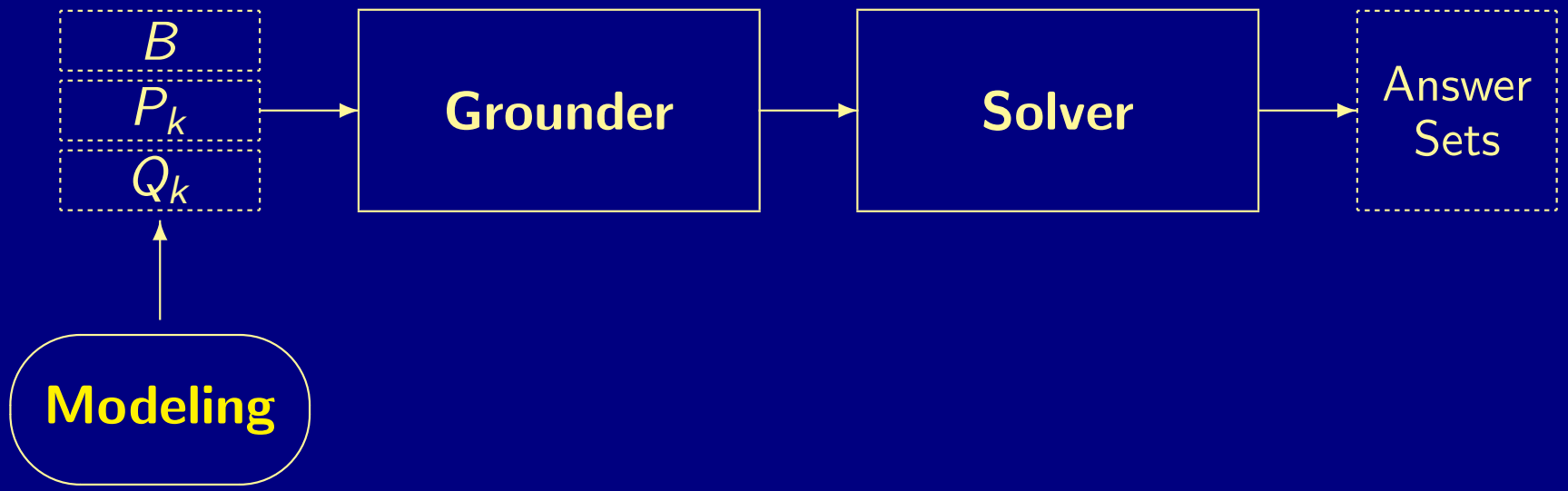
Reactive ASP Solving Process



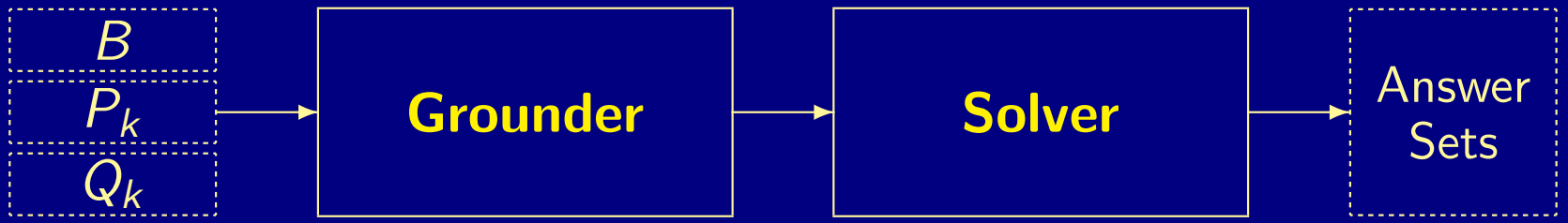
Reactive ASP Solving Process



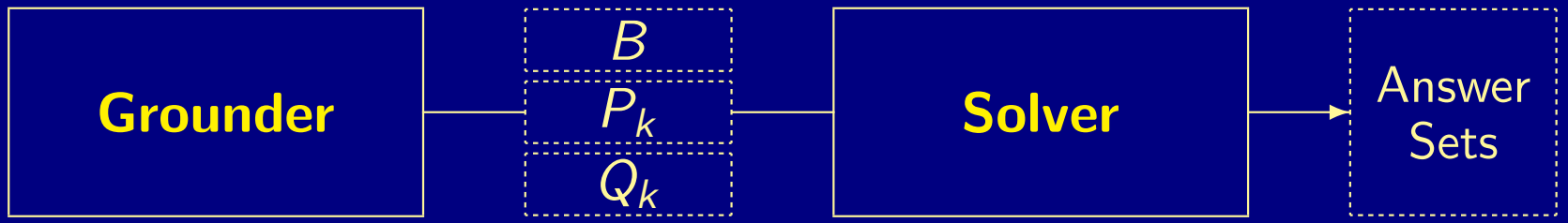
Reactive ASP Solving Process



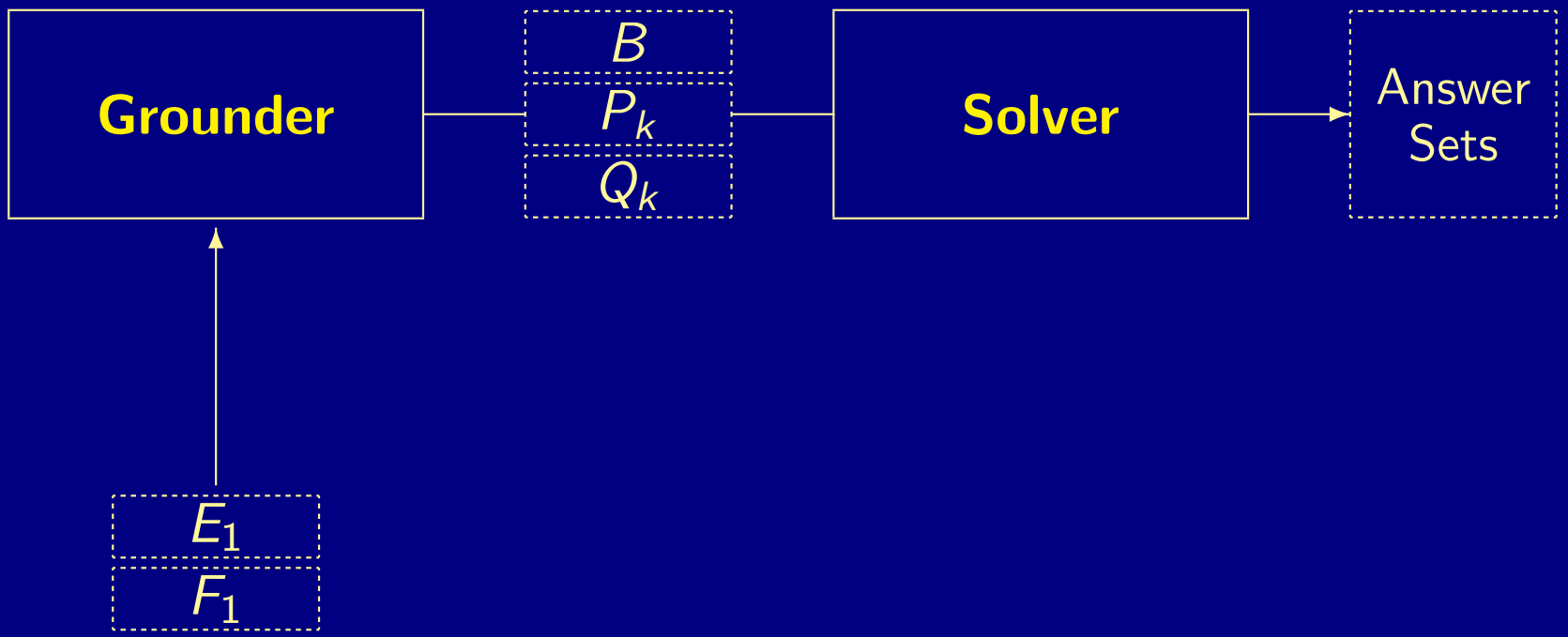
Reactive ASP Solving Process



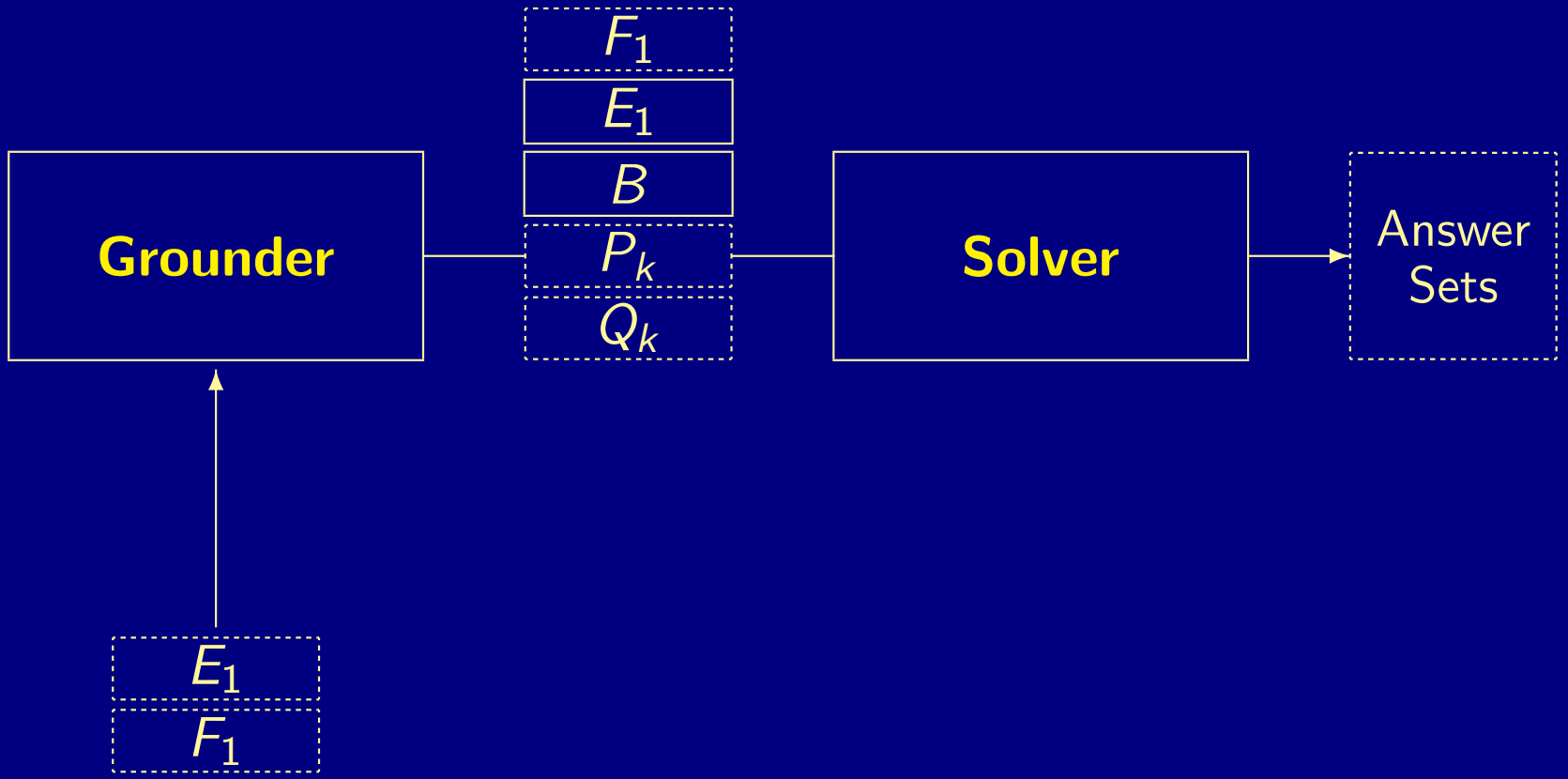
Reactive ASP Solving Process



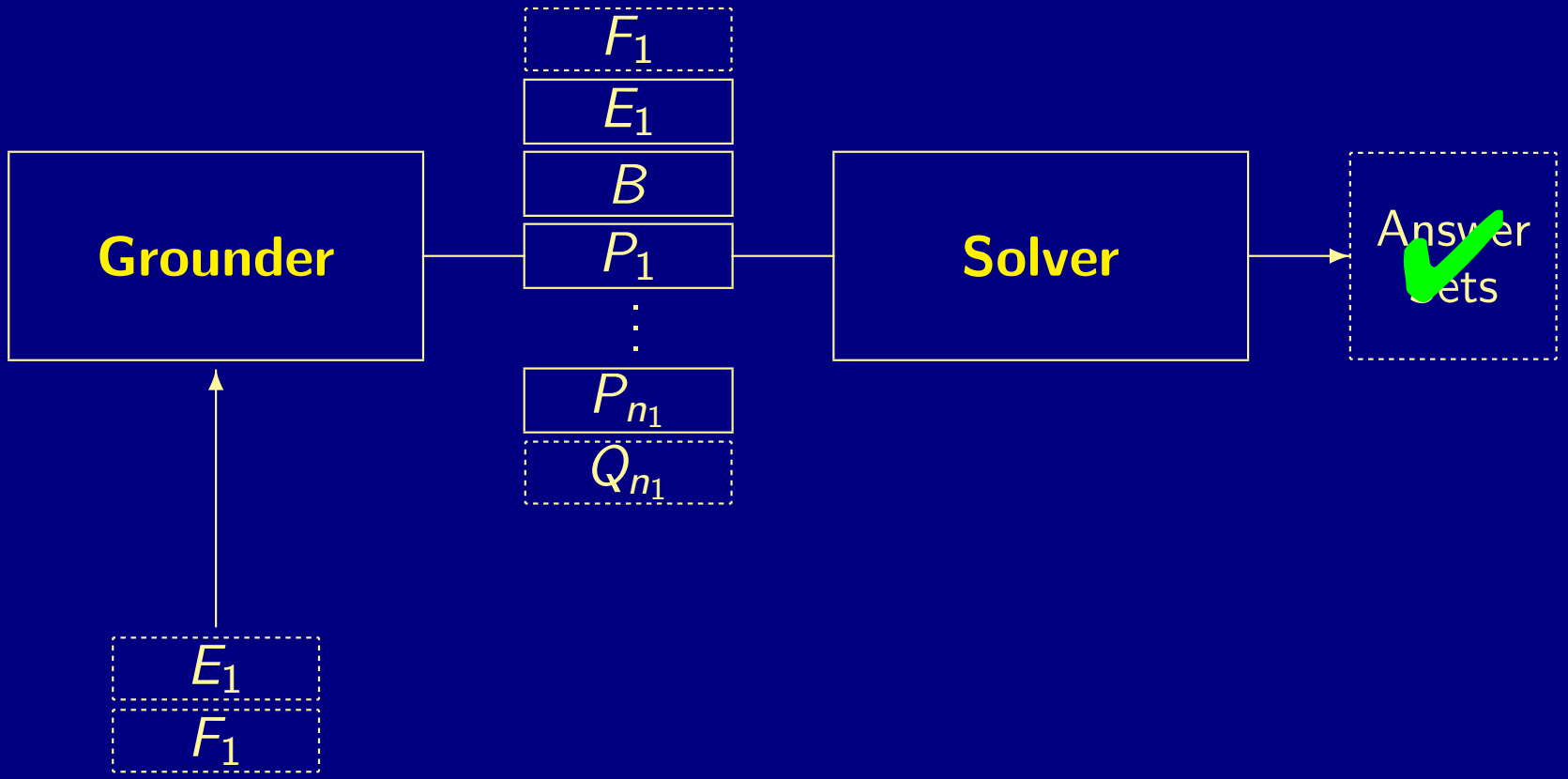
Reactive ASP Solving Process



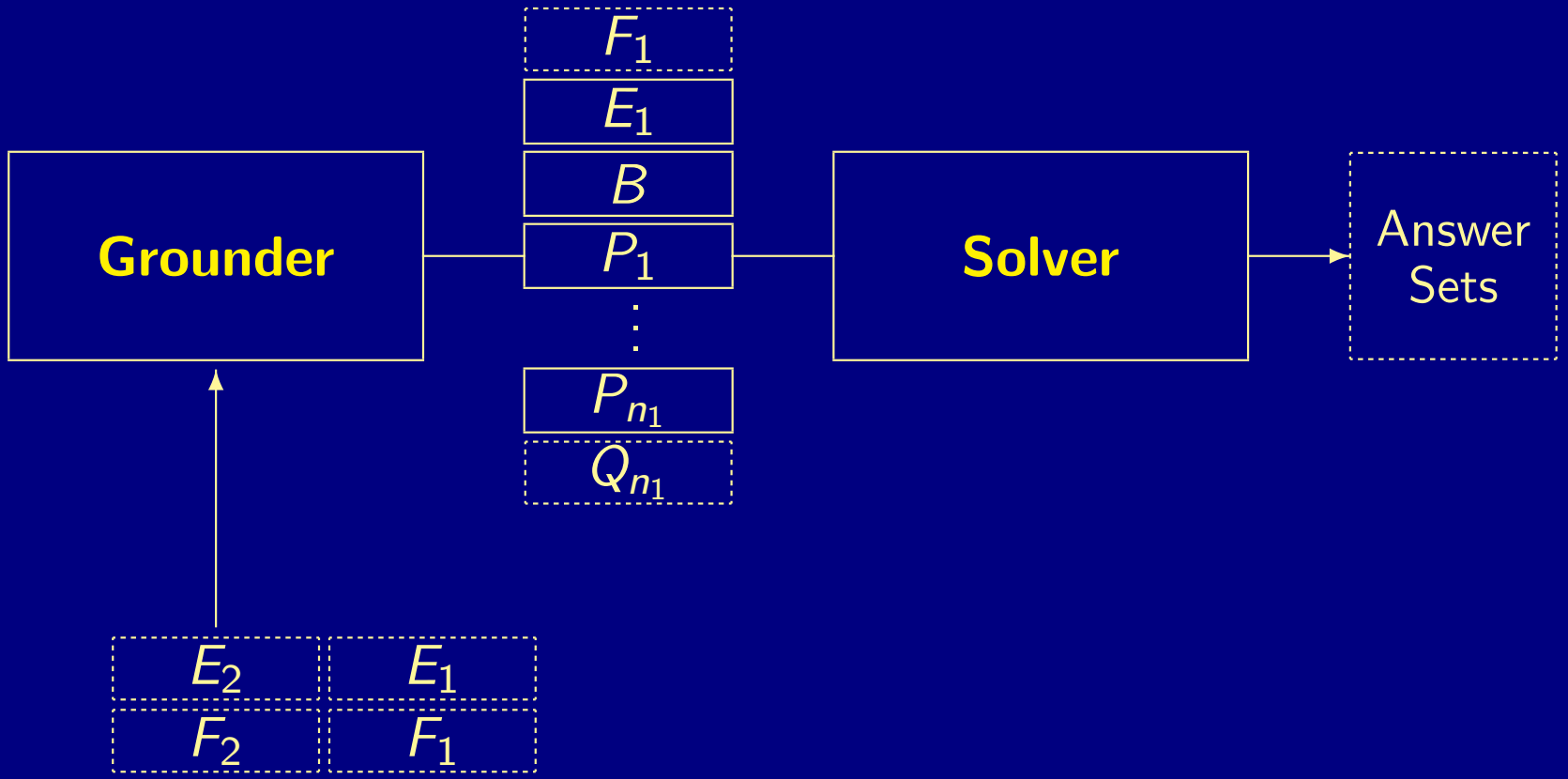
Reactive ASP Solving Process



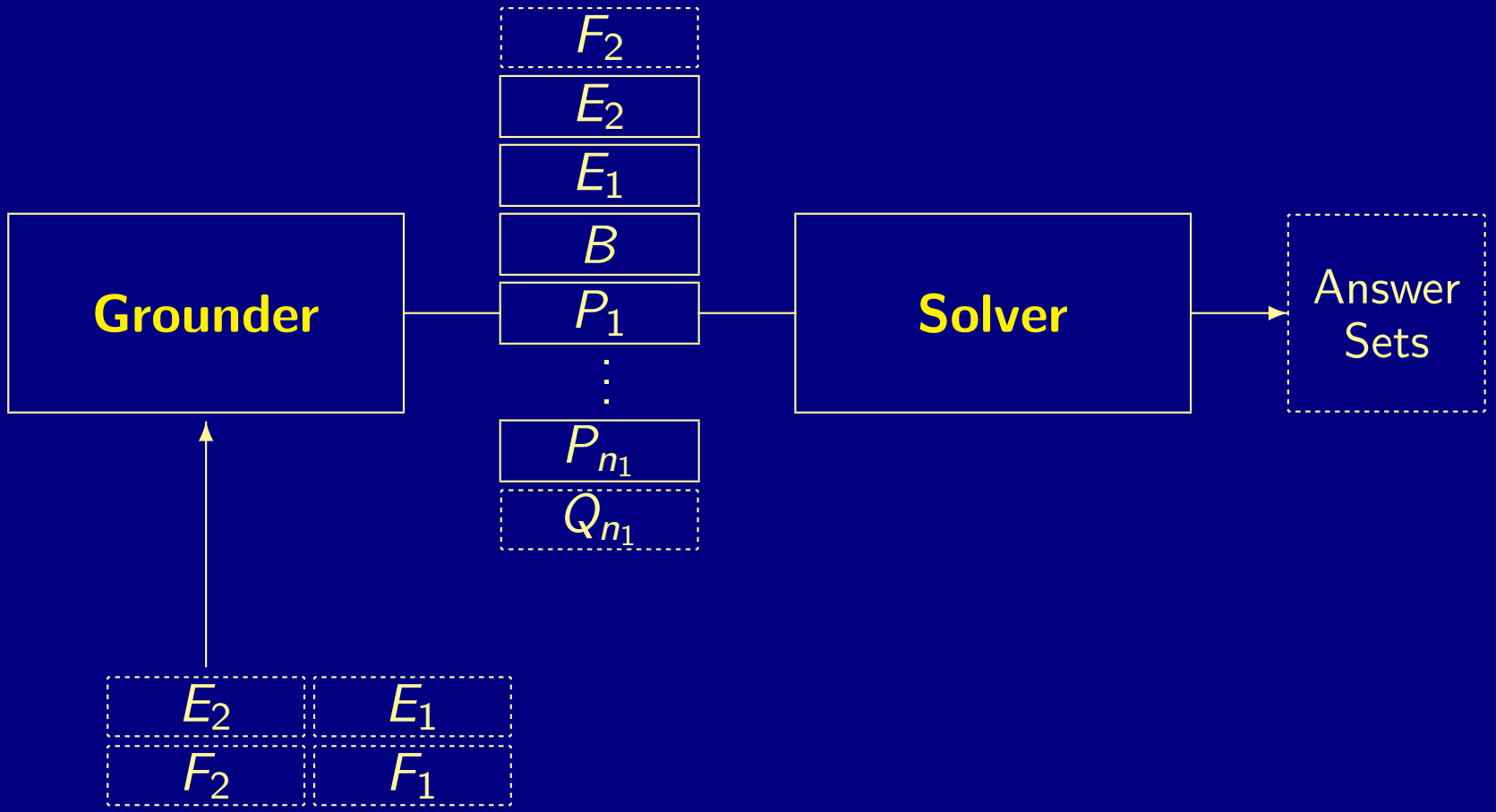
Reactive ASP Solving Process



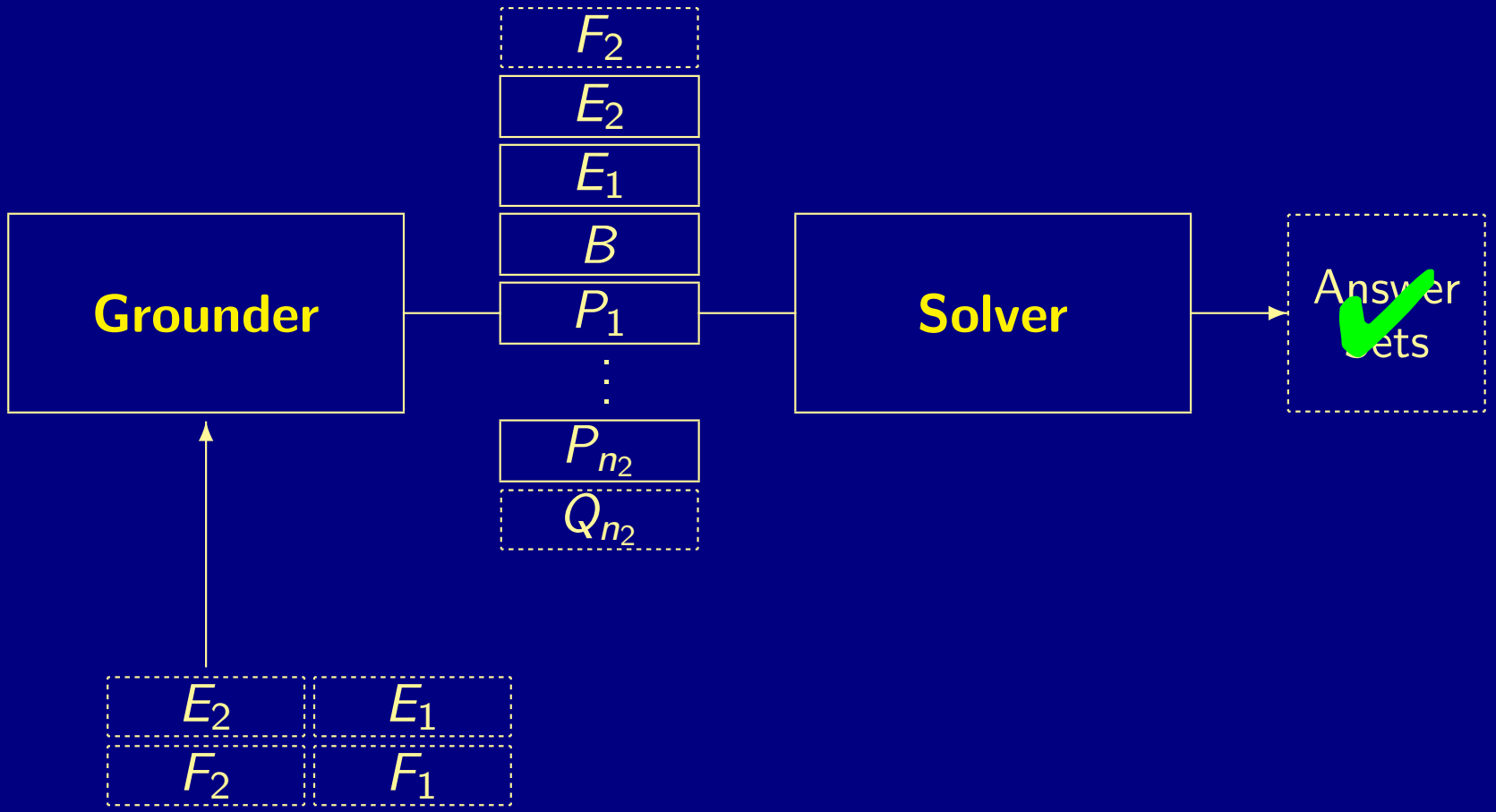
Reactive ASP Solving Process



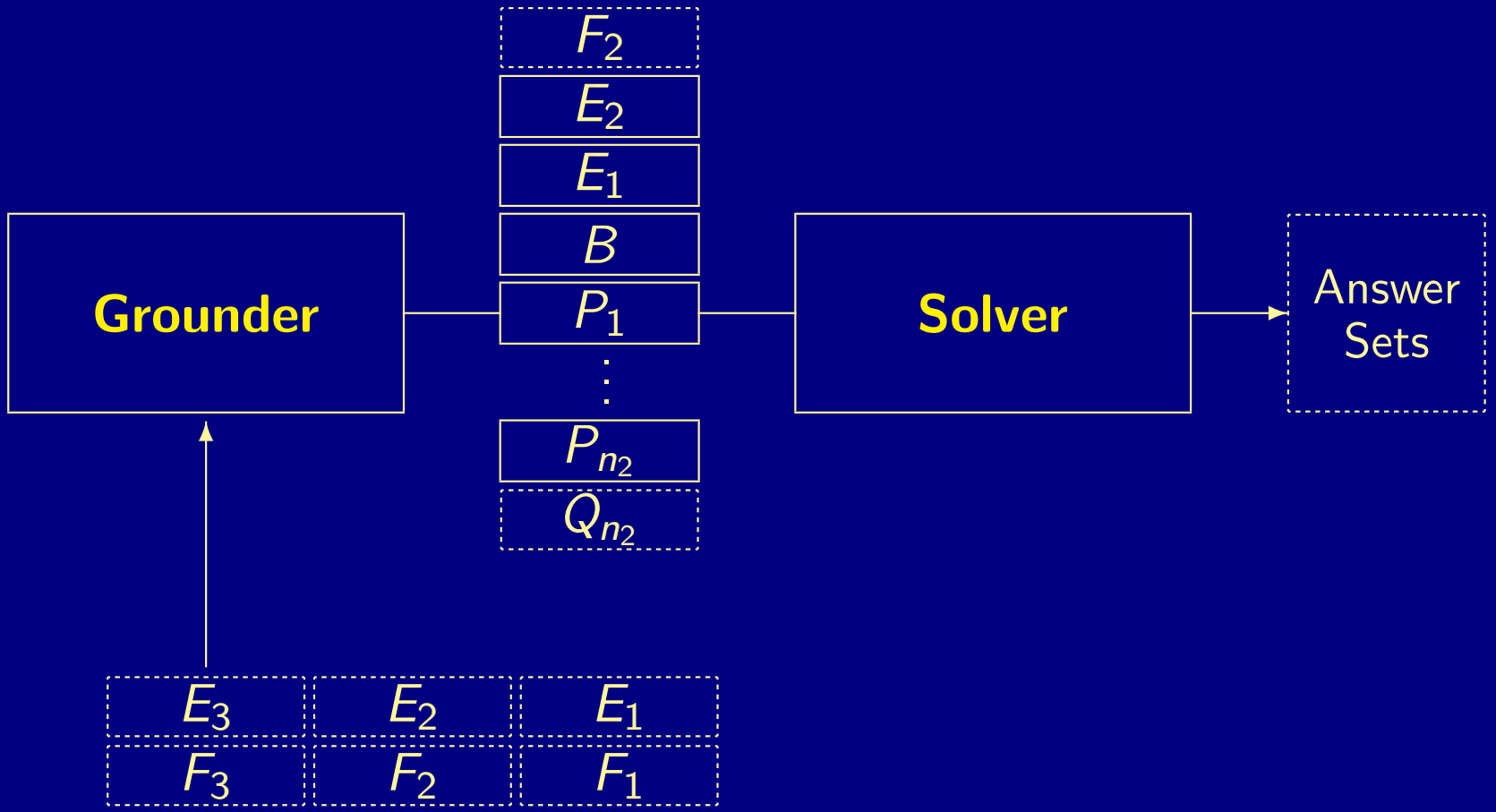
Reactive ASP Solving Process



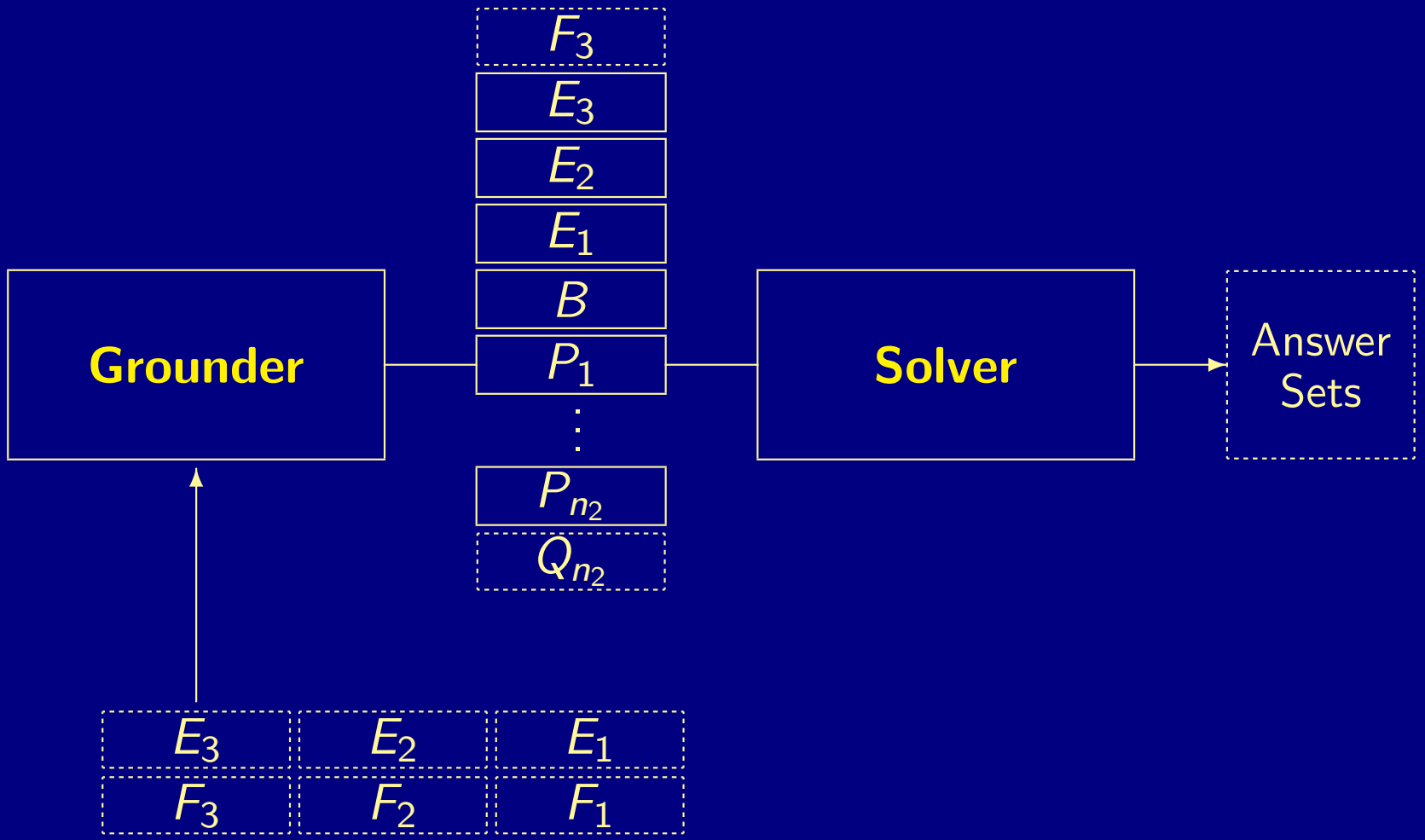
Reactive ASP Solving Process



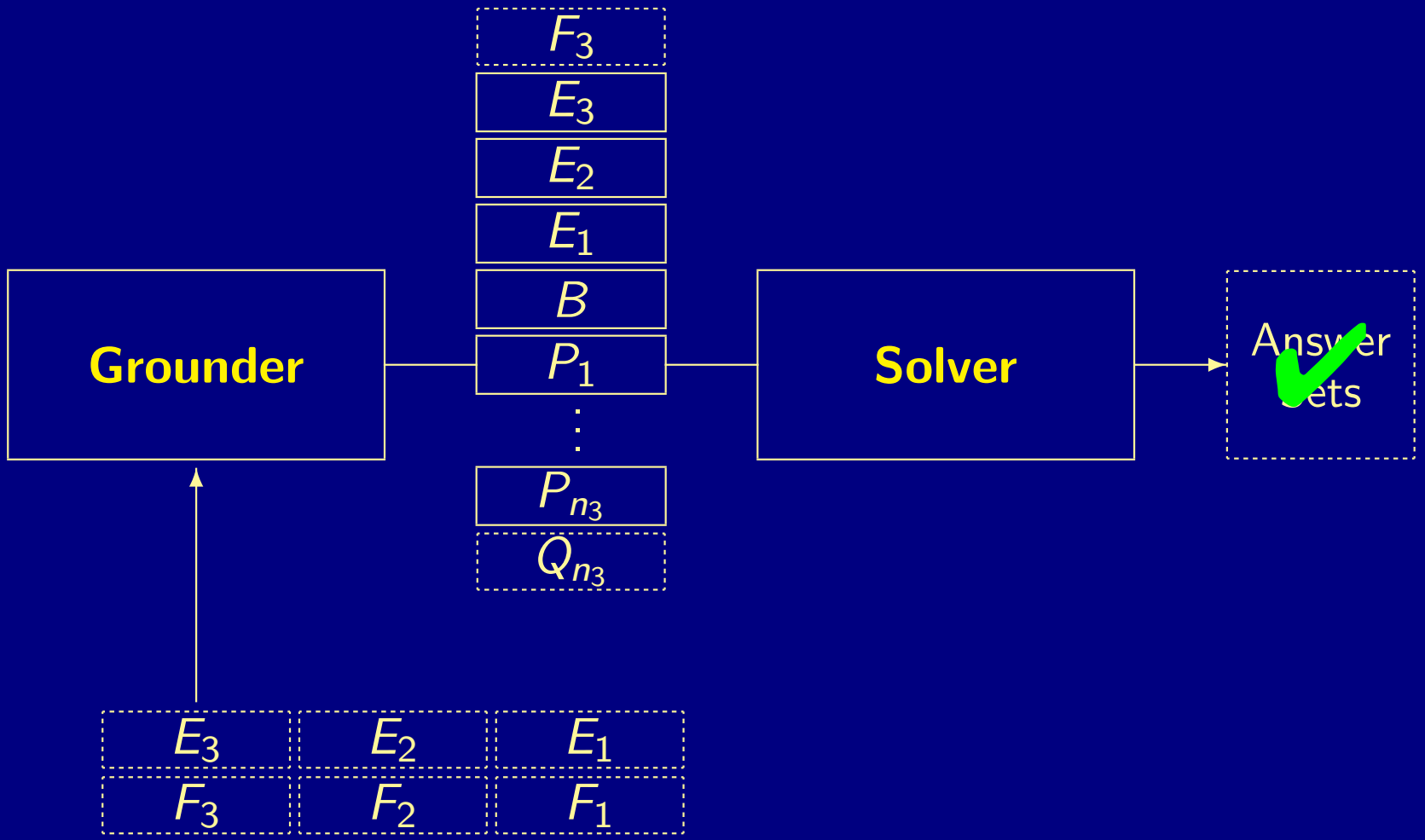
Reactive ASP Solving Process



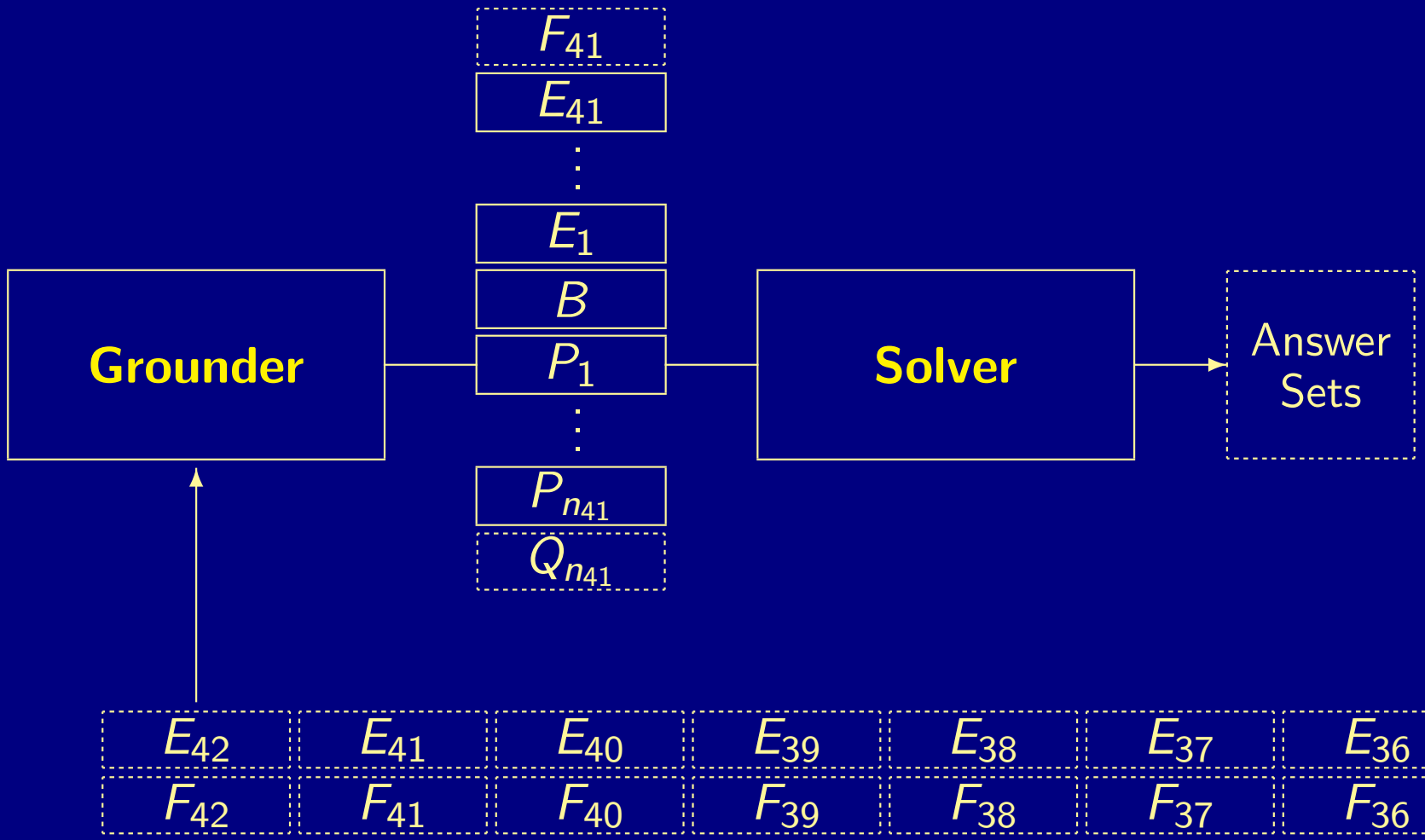
Reactive ASP Solving Process



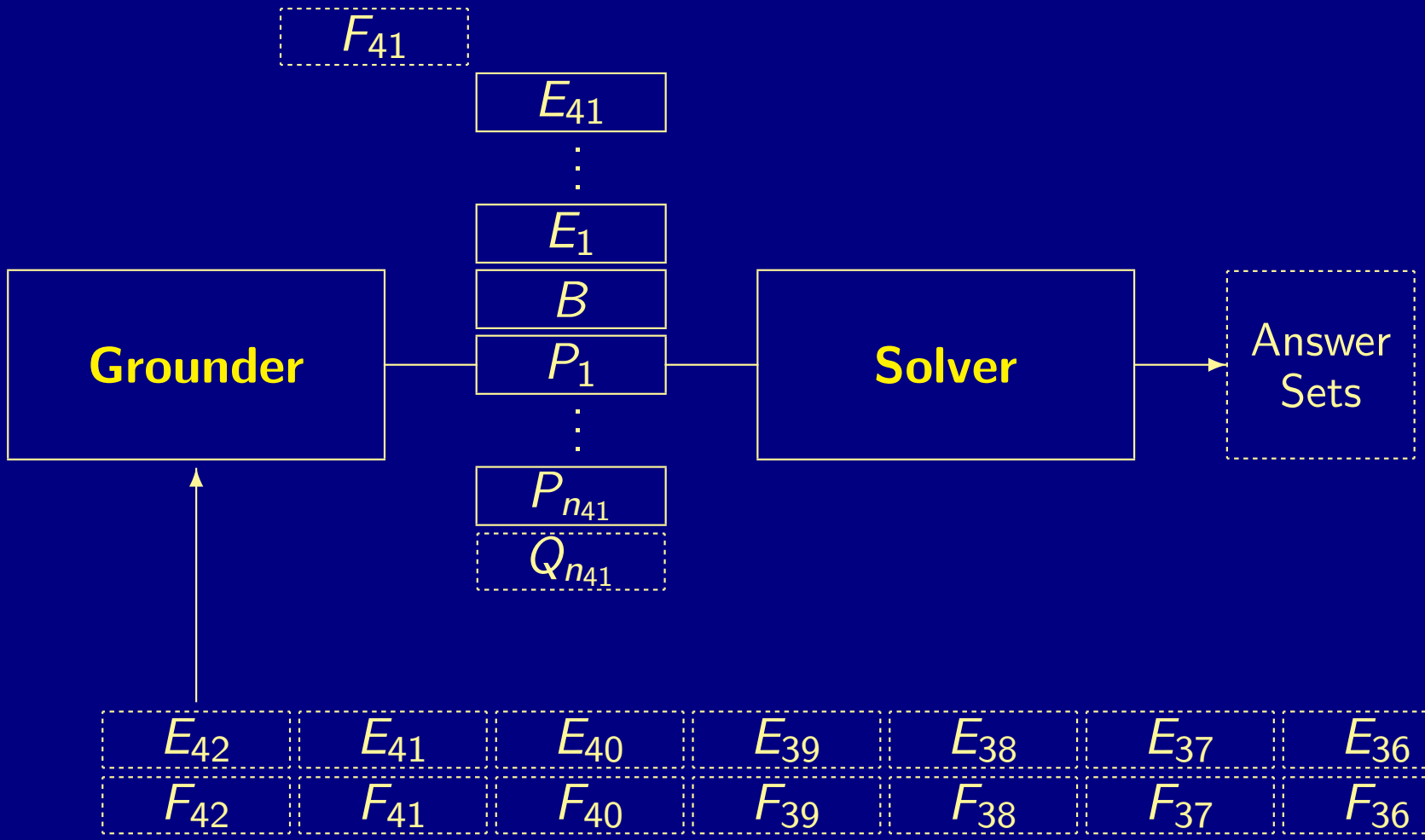
Reactive ASP Solving Process



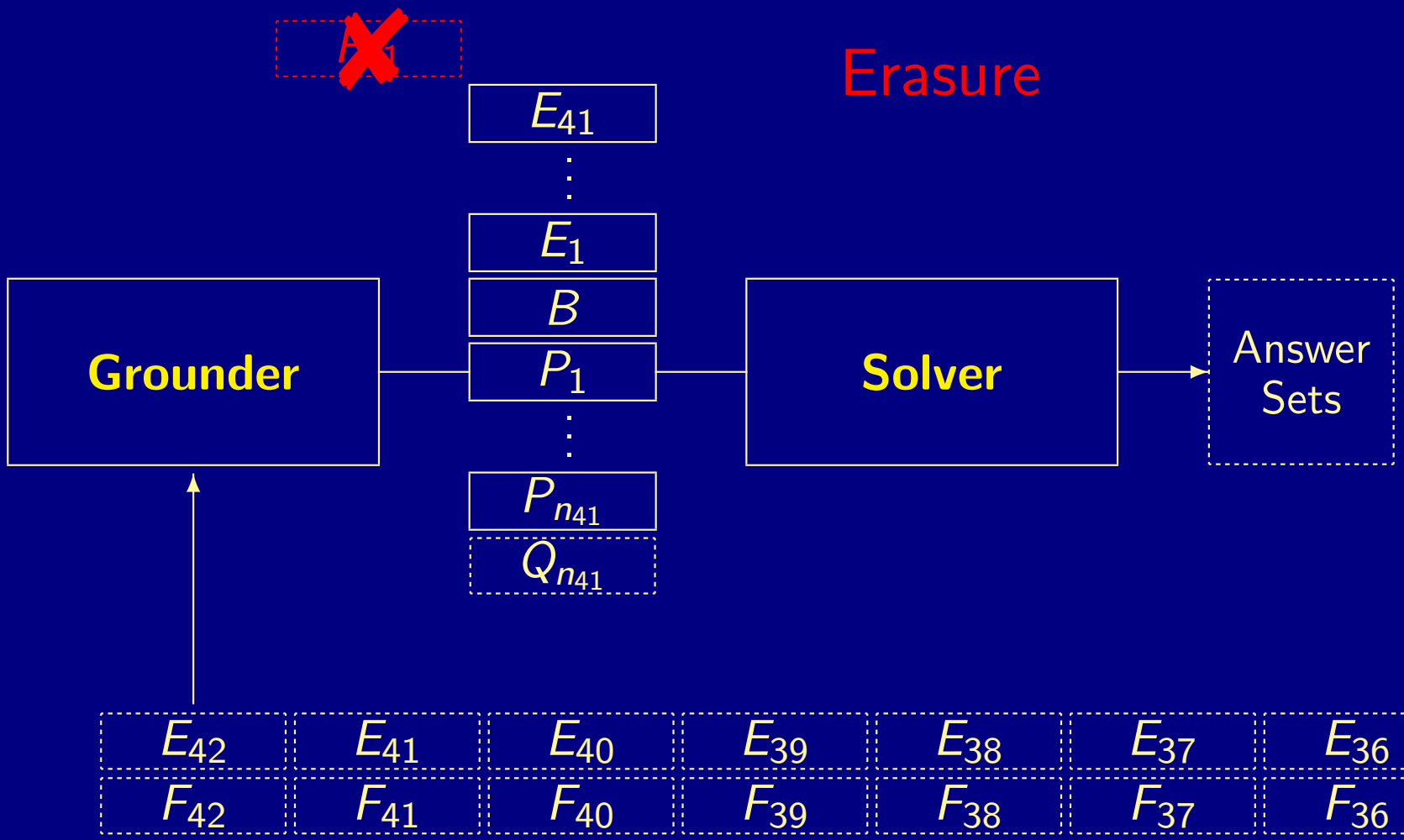
Reactive ASP Solving Process



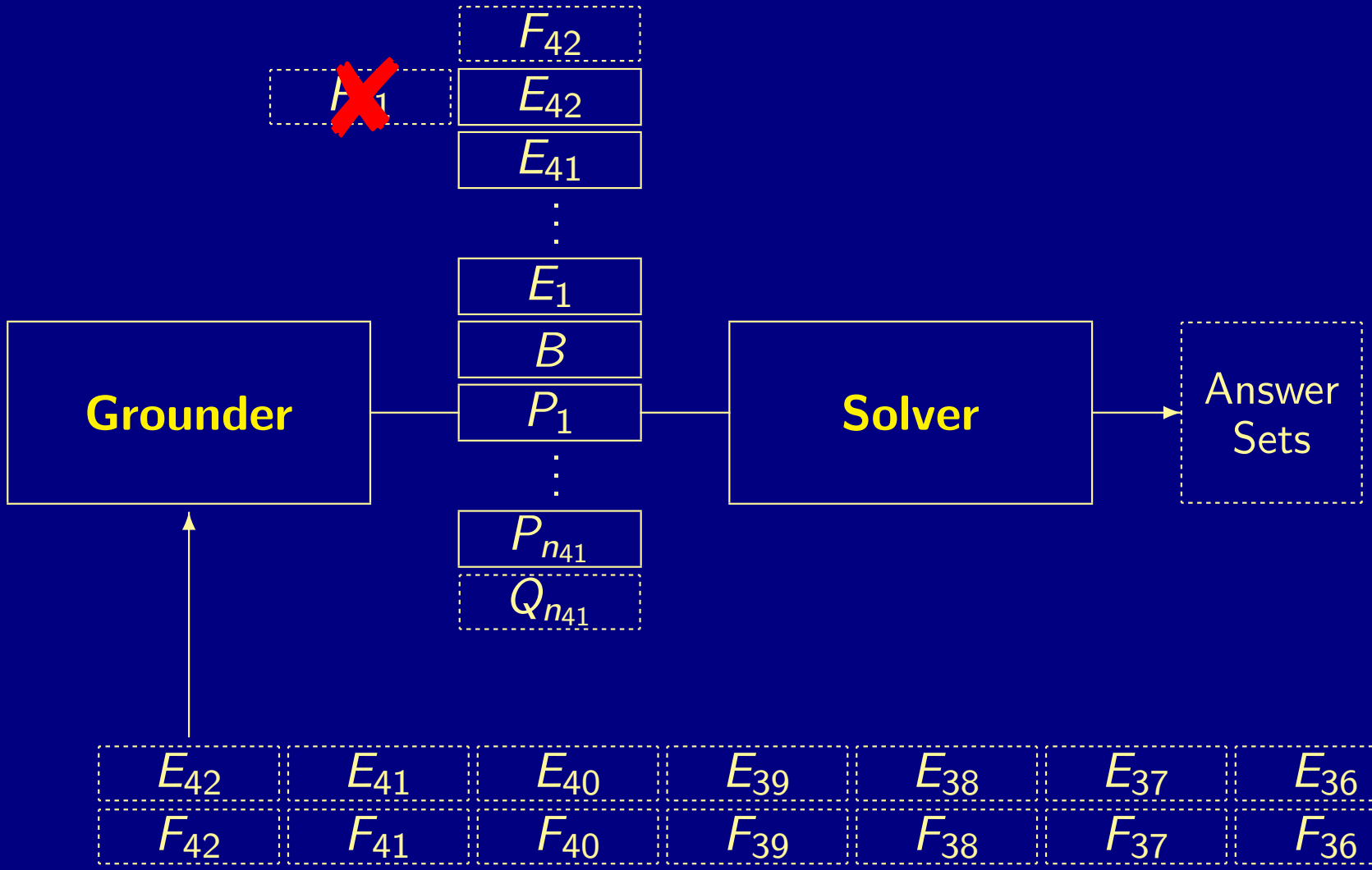
Reactive ASP Solving Process



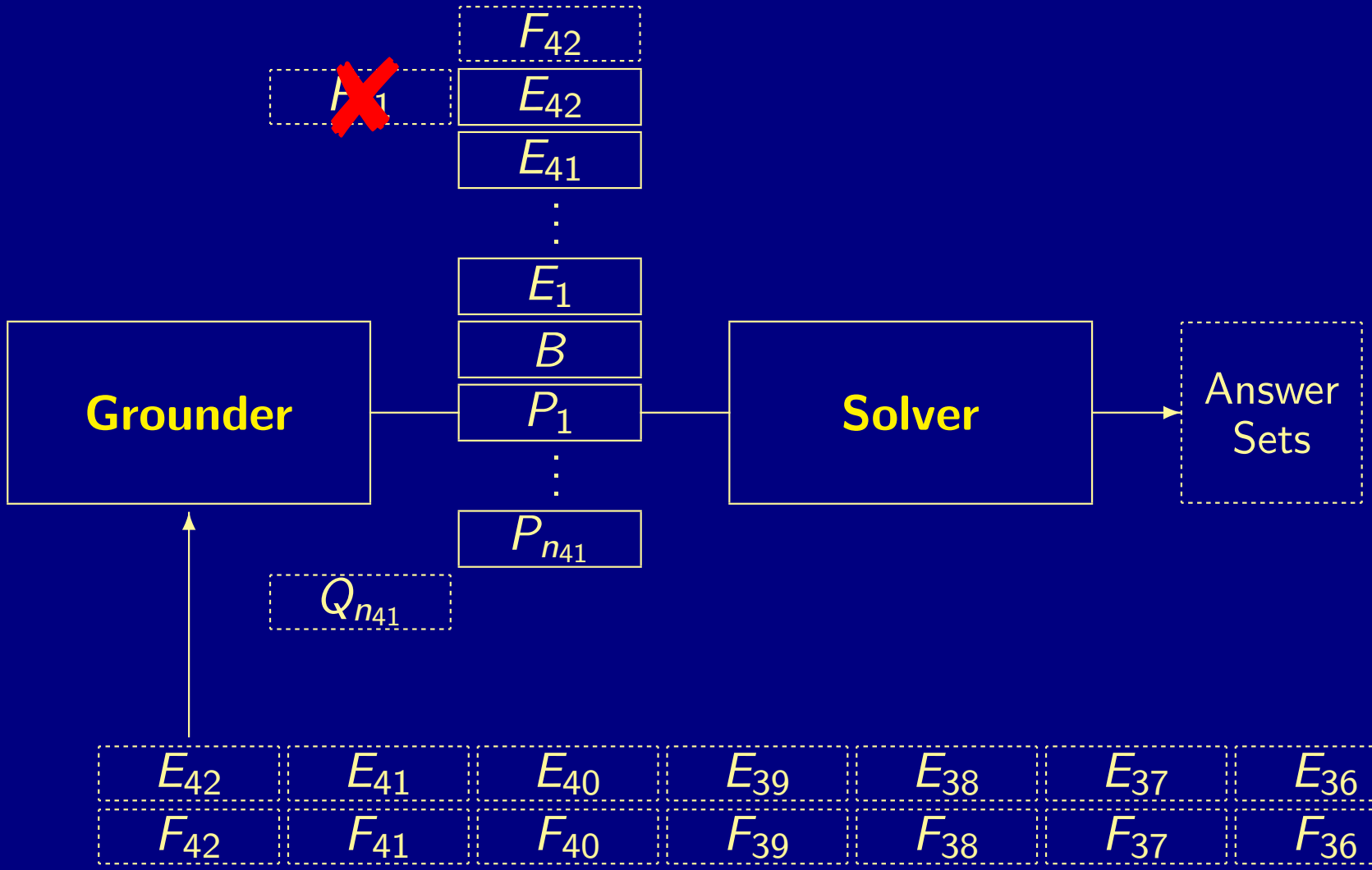
Reactive ASP Solving Process



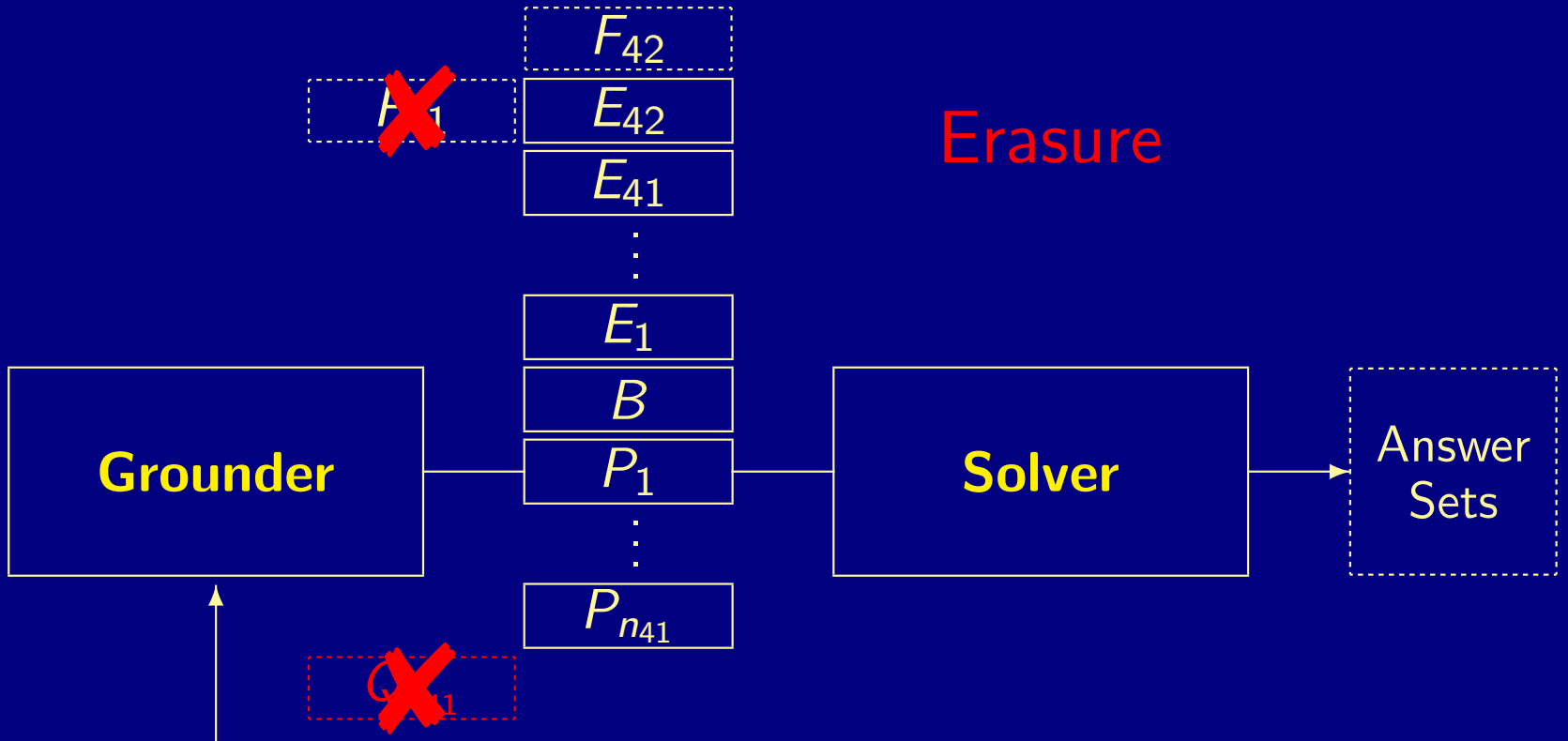
Reactive ASP Solving Process



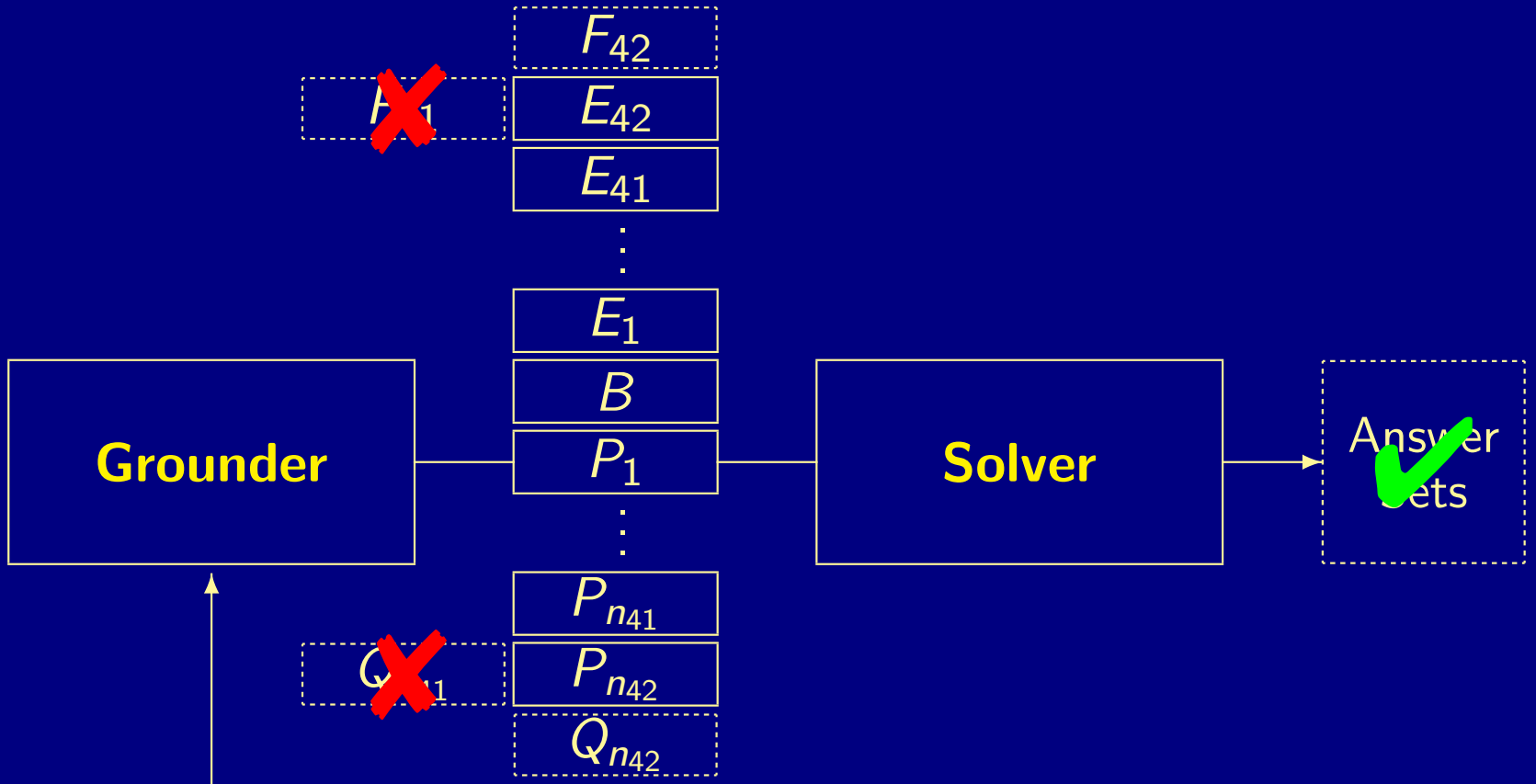
Reactive ASP Solving Process



Reactive ASP Solving Process



Reactive ASP Solving Process



Outline

Sliding Windows

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...

- Continuous data flow
- Reasoning over recent data

↪ Focus on time window

Sliding Windows

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

- Continuous data flow
- Reasoning over recent data

↪ Focus on time window

Sliding Windows

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

- Continuous data flow
- Reasoning over recent data

↪ Focus on time window

Sliding Windows

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✗	✔	✗	

- Continuous data flow
- Reasoning over recent data

↪ Focus on time window

Sliding Windows

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

- Continuous data flow
- Reasoning over recent data

↪ Focus on time window

Sliding Windows

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✗	✓	✗	

- Continuous data flow
- Reasoning over recent data

↪ Focus on time window

Traditional ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding

```
accept :-
    read(a,T),
    read(a,T-1).
```

Data

```
read(a,1).
read(a,2).
read(b,3).
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

Traditional ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	*a*	*a*	*b*	...
	✘	✔	✘	

Encoding

```
accept :-
    read(a,T),
    read(a,T-1).
```

Data

```
read(a,1).
read(a,2).
read(b,3).
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

Traditional ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	*a*	*a*	*b*	...
	✘	✔	✘	

Encoding

```
accept :-
    read(a,T),
    read(a,T-1).
```

Data

```
read(a,1).
read(a,2).
read(b,3).
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

Traditional ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding

```
accept :-
    read(a,T),
    read(a,T-1).
```

Data

```
read(a,1).
read(a,2).
read(b,3).
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

Traditional ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding

```
accept :-
    read(a,T),
    read(a,T-1),
not read(a;b,T+1).
```

Data

```
read(a,1).
read(a,2).
read(b,3).
```

Instantiation

```
accept :- read(a,2), read(a,1).

accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

Traditional ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding

```
accept :-
    read(a,T),
    read(a,T-1),
not read(a;b,T+1).
```

Data

```
read(a,1).
read(a,2).
read(b,3).
```

Instantiation

```
accept :- read(a,2), read(a,1),
          not read(a;b,3).
accept :- read(a,3), read(a,2),
          not read(a;b,4).
```

✘ Data remain persistent !

Traditional ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding

```
accept :-
    read(a,T),
    read(a,T-1),
not read(a;b,T+1).
```

Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,2), read(a,1),
          not read(a;b,3).
accept :- read(a,3), read(a,2),
          not read(a;b,4).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1),
not read(a;b,t+1).
```

E-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,2), read(a,1),
          not read(a;b,3).
accept :- read(a,3), read(a,2),
          not read(a;b,4).
```

- ✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1),
not read(a;b,t+1).
```

E-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,2), read(a,1),
          not read(a;b,3).
accept :- read(a,3), read(a,2),
          not read(a;b,4).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	*a*	*a*	*b*	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	*a*	*a*	*b*	...
	✘	✘	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✘	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

- ✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✘	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	*a*	*a*	*b*	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data remain persistent !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data must be replayed !

(Simplified) Reactive ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data must be replayed !

NEW (Simplified) Stream ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data must be replayed !

NEW (Simplified) Stream ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F:L-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data must be replayed !

NEW (Simplified) Stream ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F:2-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data must be replayed !

NEW (Simplified) Stream ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F:2-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data must be replayed !

NEW (Simplified) Stream ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	*a*	*a*	*b*	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F:2-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
```

✘ Data must be replayed !

NEW (Simplified) Stream ASP

Running Example

- Continuously recognize strings matching: $(a|b)^*aa$

Position	1	2	3	...
Reading	a	a	b	...
	✘	✔	✘	

Encoding[t]

```
accept :-
    read(a,t),
    read(a,t-1).
```

F:L-Data

```
read(a,1).
read(a,2).
read(b,3).
...
```

Instantiation

```
accept :- read(a,1), read(a,0).
accept :- read(a,2), read(a,1).
accept :- read(a,3), read(a,2).
...
```

- **Seamless integration of online data with offline encoding !**

Outline

Discussion

- We provide
  1. Language
  2. Semantics
  3. Modeling techniques
  4. (Prototype) implementation

  for transparent grounding and solving of (time-decaying) logic programs w.r.t. emerging/expiring stream data
- Future work addresses
  1. Data cleansing
  2. Query answering
  3. Optimization support

➥ ASP technology for knowledge-intense stream reasoning

- Application areas include
  1. Ambient Assisted Living
  2. Robotics Planning and Control
  3. Policy Monitoring and Enforcement
  4. etc.