Lådmodellen

...eller ”procedure box control flow model”.

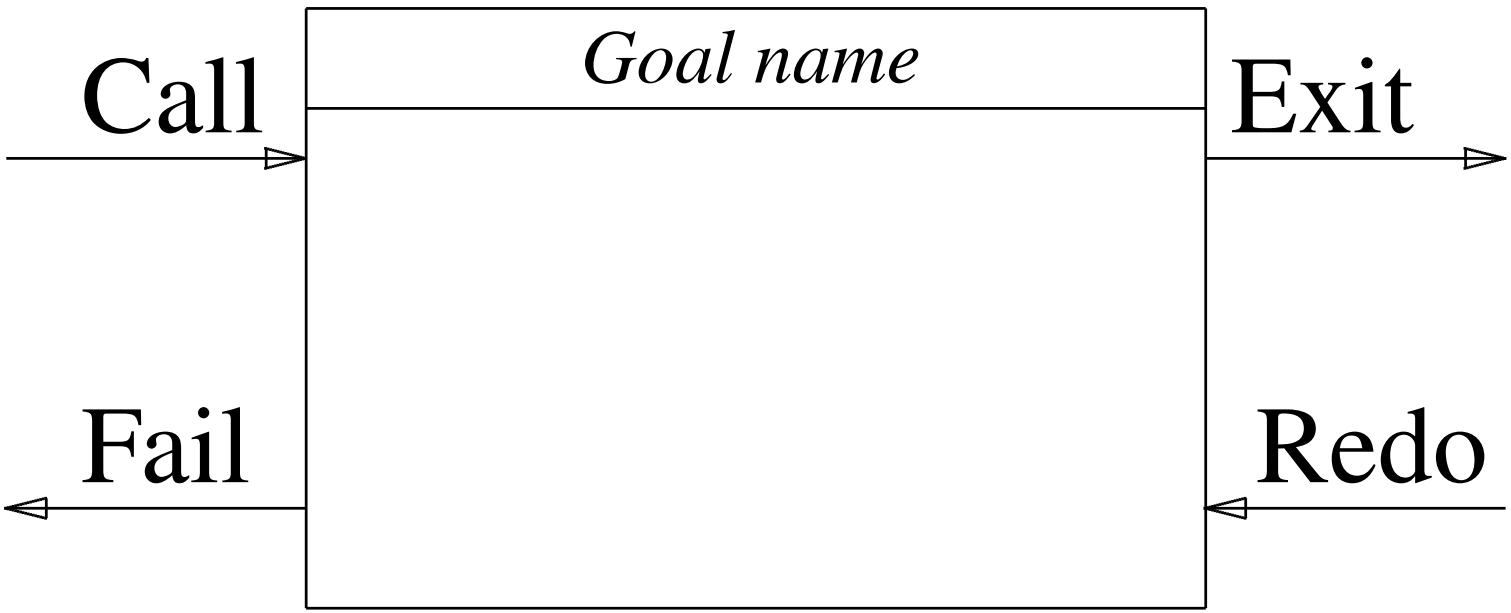
- *Dynamisk* bild av ett prologprogram
- Förklarar i detalj *procedurell läsning* av ett program.

Lådmodellen

...eller ”procedure box control flow model”.

- *Dynamisk* bild av ett prologprogram
- Förklarar i detalj *procedurell läsning* av ett program.
- Bra förklaringsmodell för ett Prologsystem
- Bra för avlusning!
- SWI Prolog använder lådmodellen i sin `trace`-funktion.

Lådmodellen: byggstenen



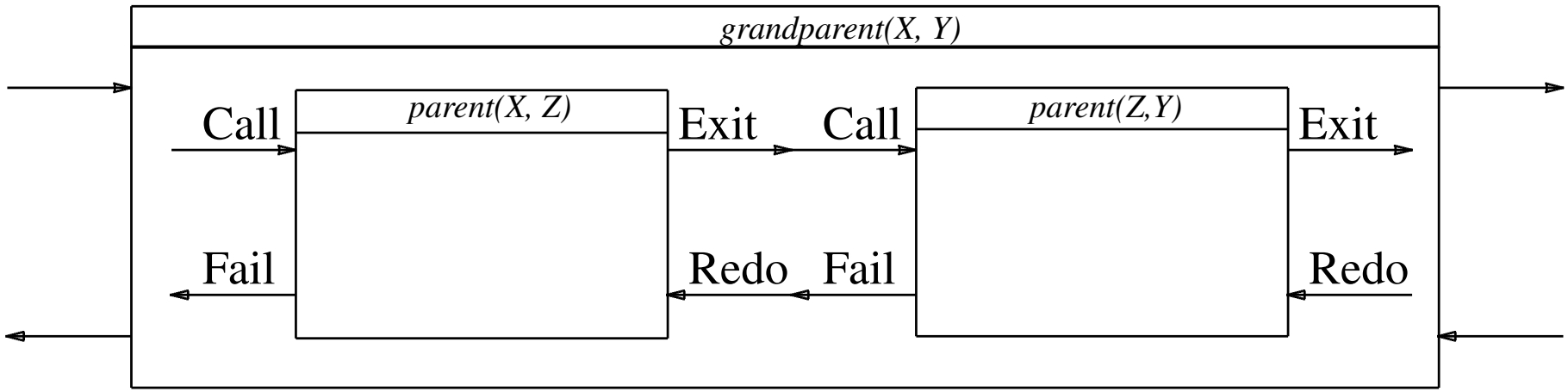
En låda per mål/predikat

Lådmodellen: princip

- Varje mål får en egen låda.
- Konjunktioner kopplas ihop.
- Disjunktioner är alternativa vägar.
- Prologs unifieringsalgoritm motsvaras av vandringar över lådorna.

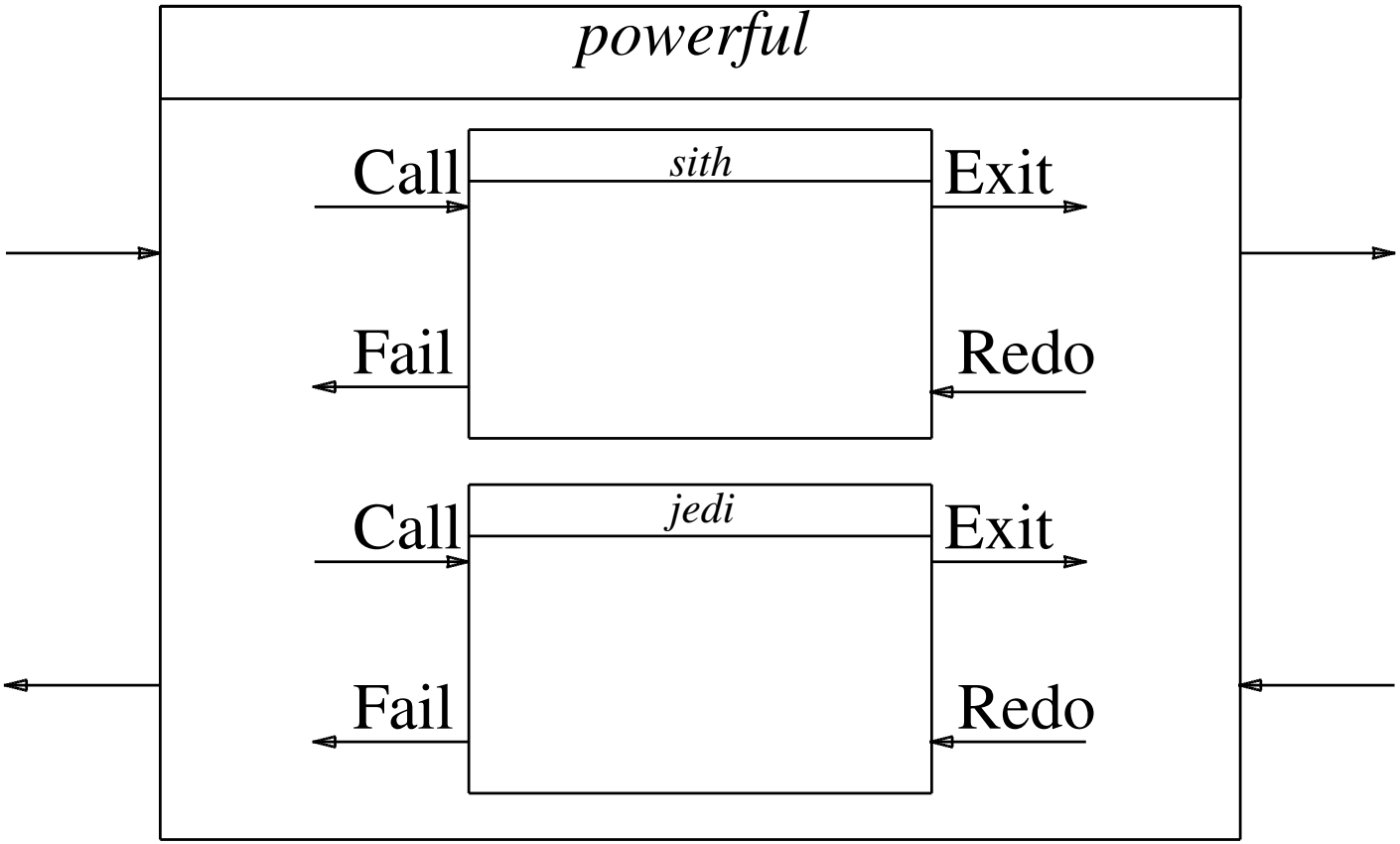
Exempel: grandparent

```
grandparent(X,Y) :-
        parent(X,Z),parent(Z,Y).
```



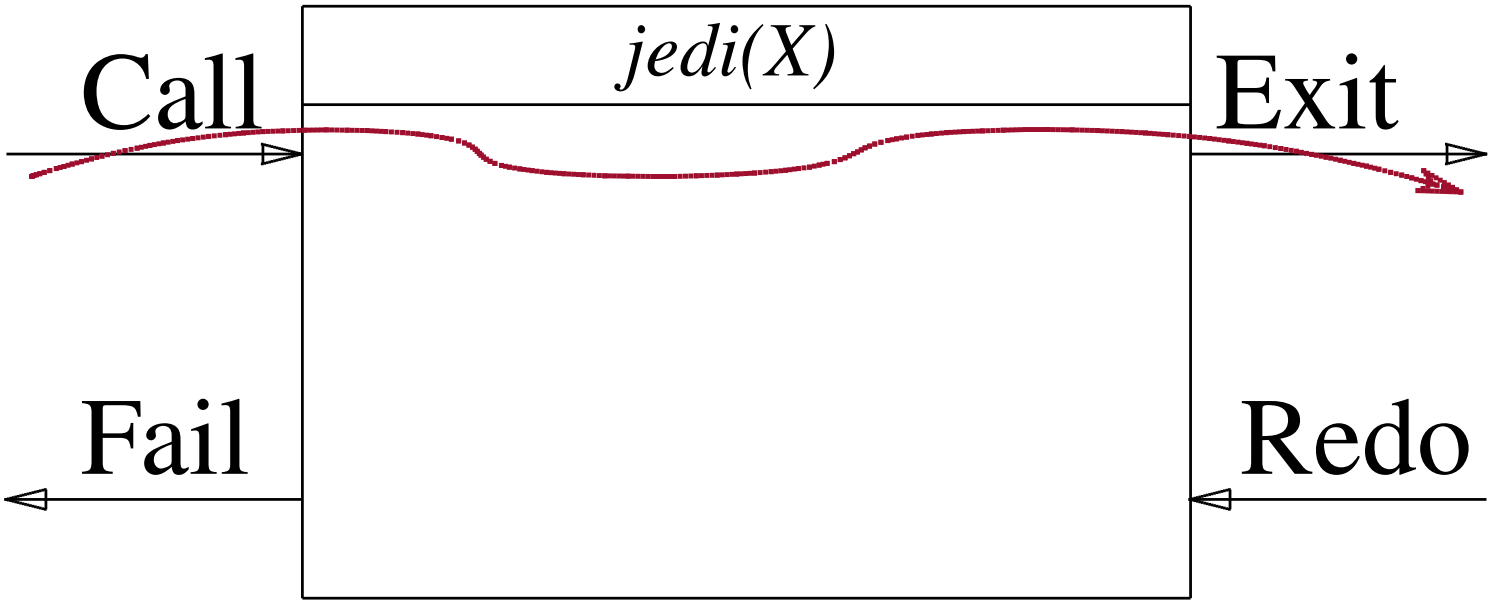
Exempel: powerful

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```



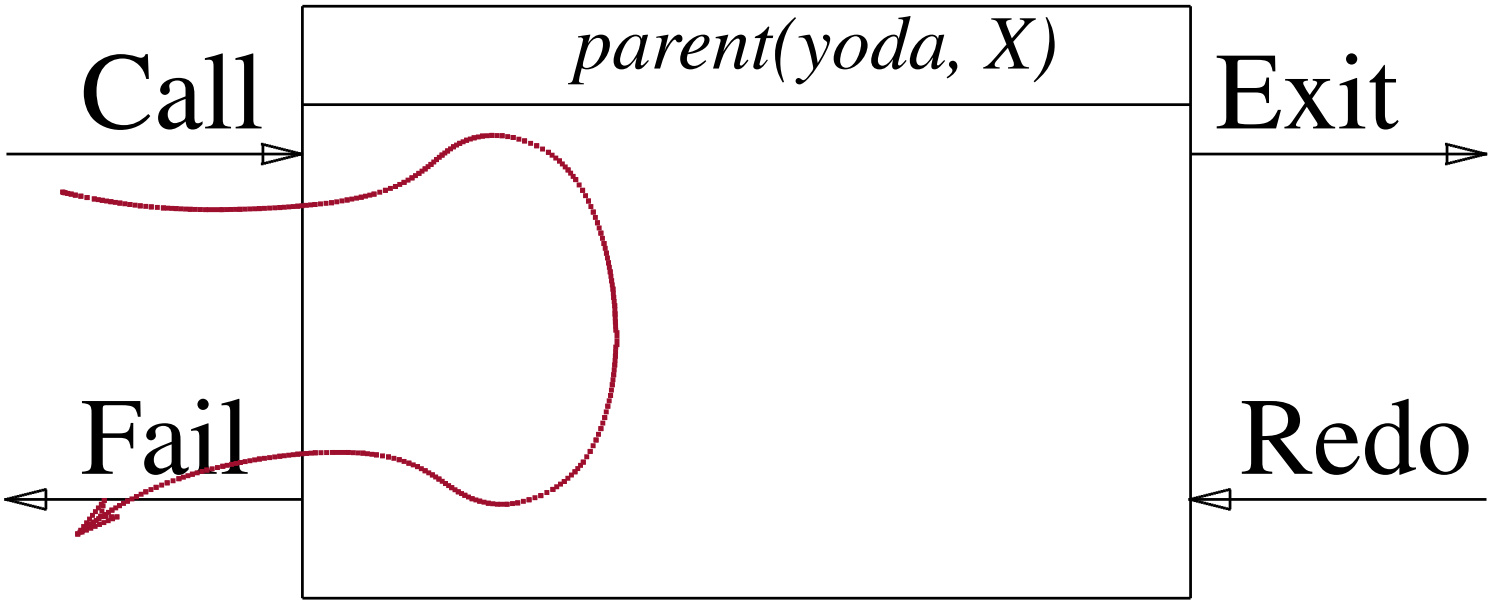
Semantik: Unifiering med faktum

`jedi(yoda)` ger *X* = `yoda`.



Semantik: Ingen unifiering

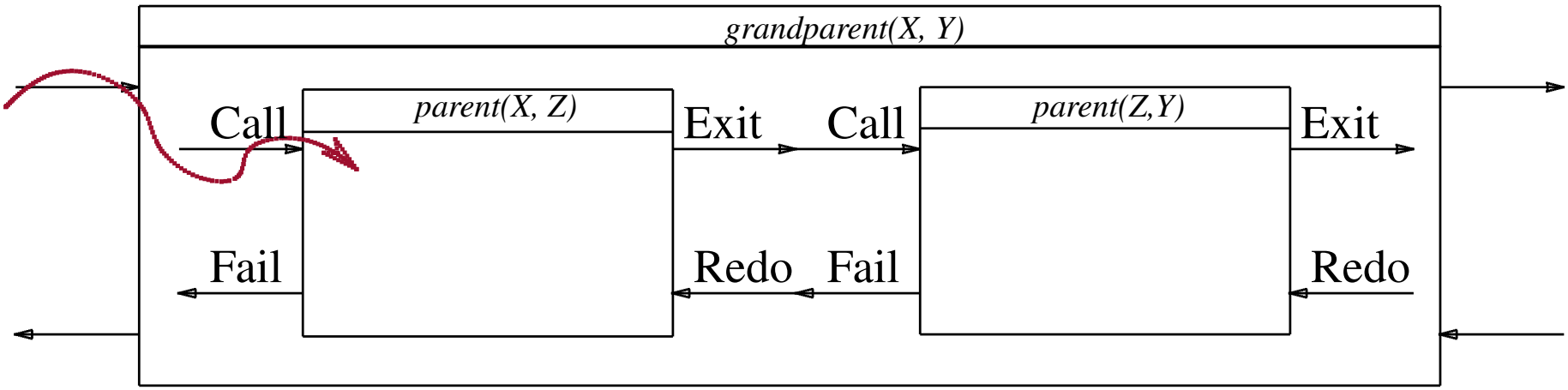
Om unifiering misslyckas:



Semantik: Unifiering med regel

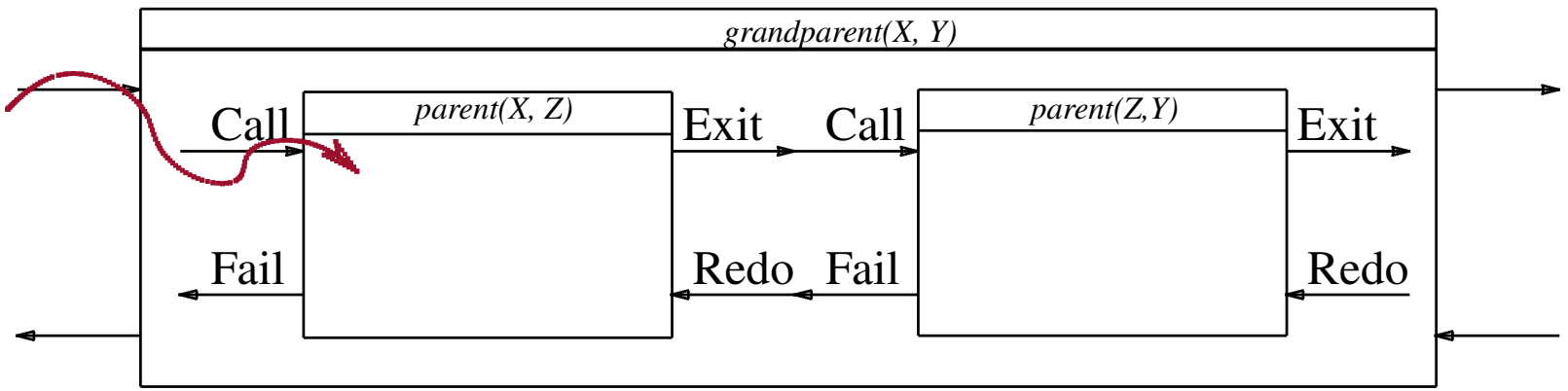
Hittar att

```
grandparent(X, Y) :- parent(X, Z),...
```



Ett programflöde

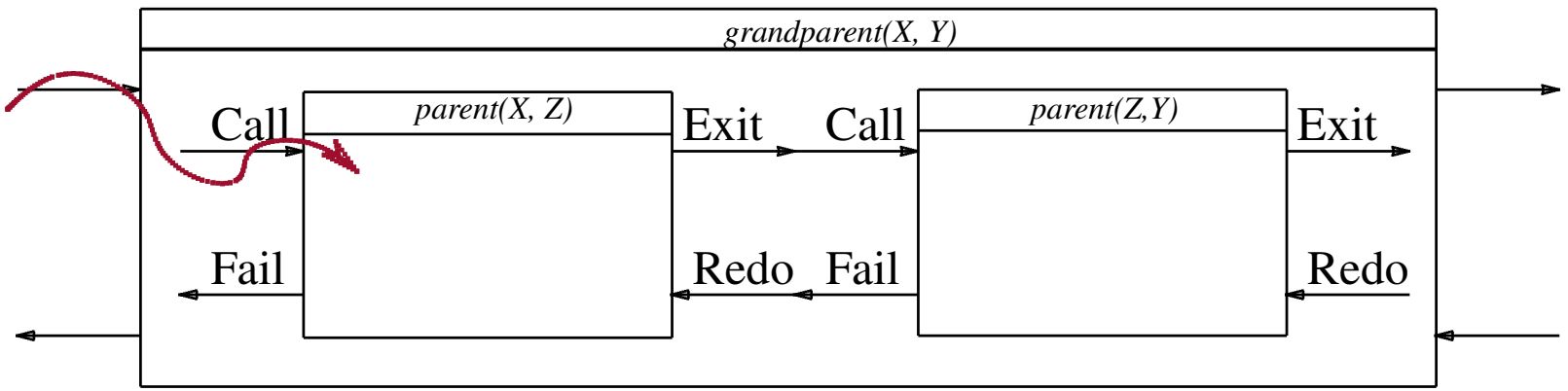
```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```



Ett programflöde

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```

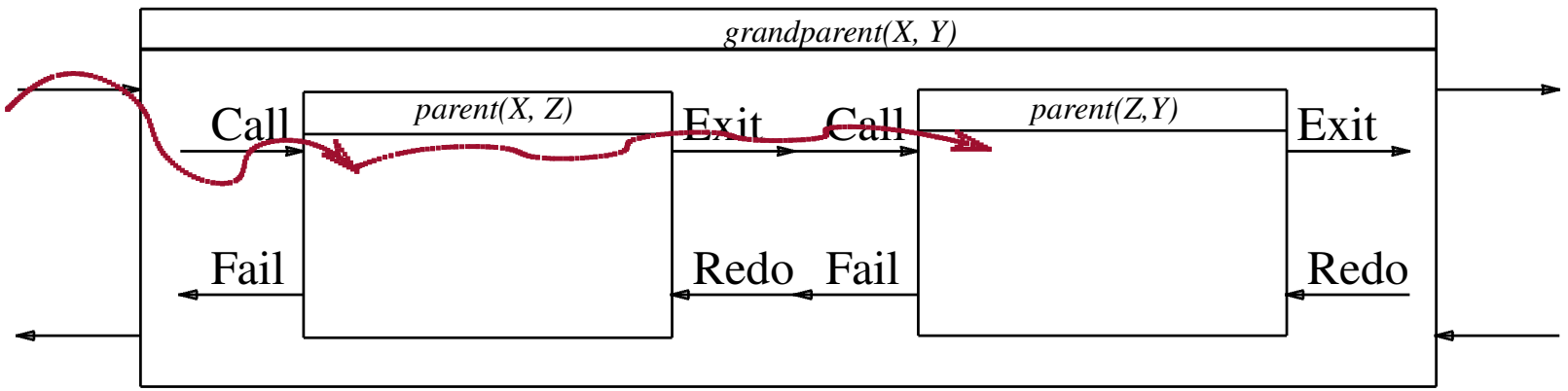
X=anakin, Z=luke



Ett programflöde

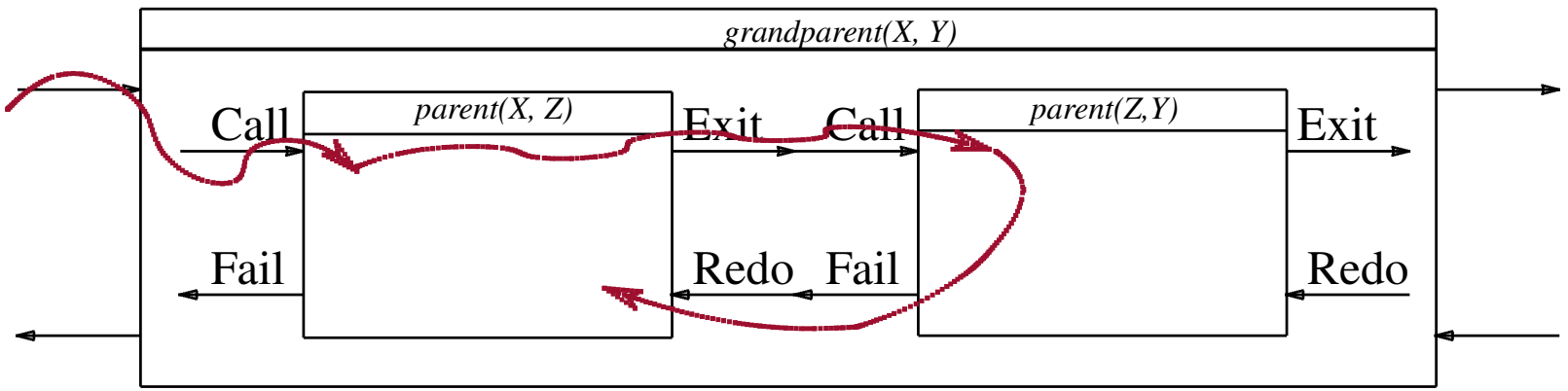
```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```

X=anakin, Z=luke



Ett programflöde

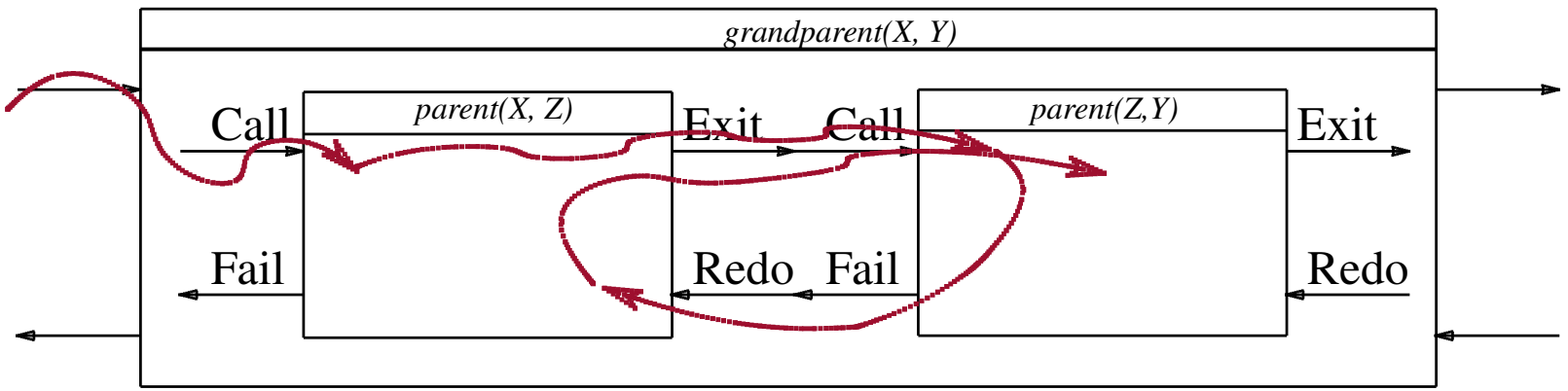
```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```



Ett programflöde

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```

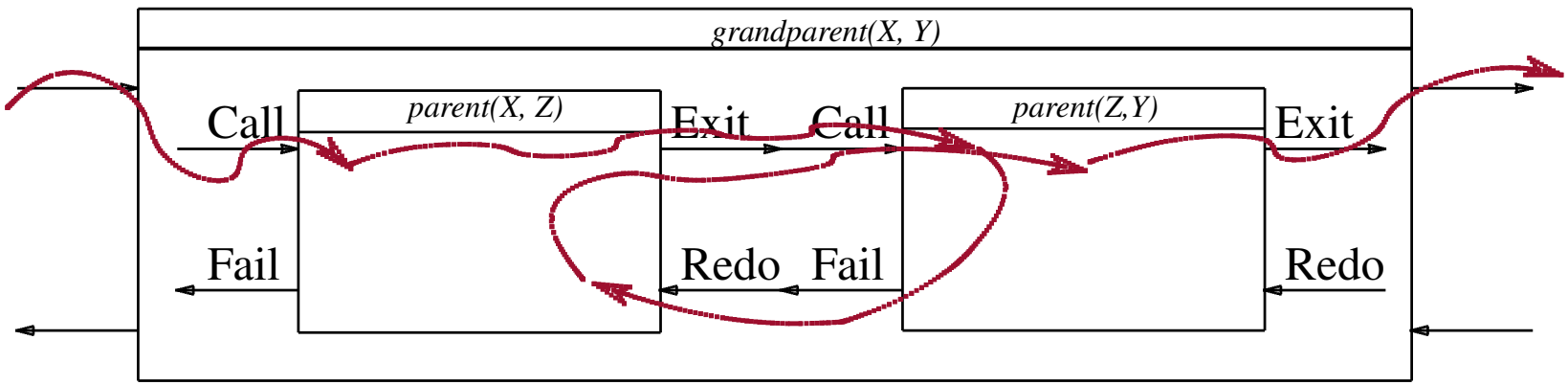
X=schmi, Z=anakin



Ett programflöde

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```

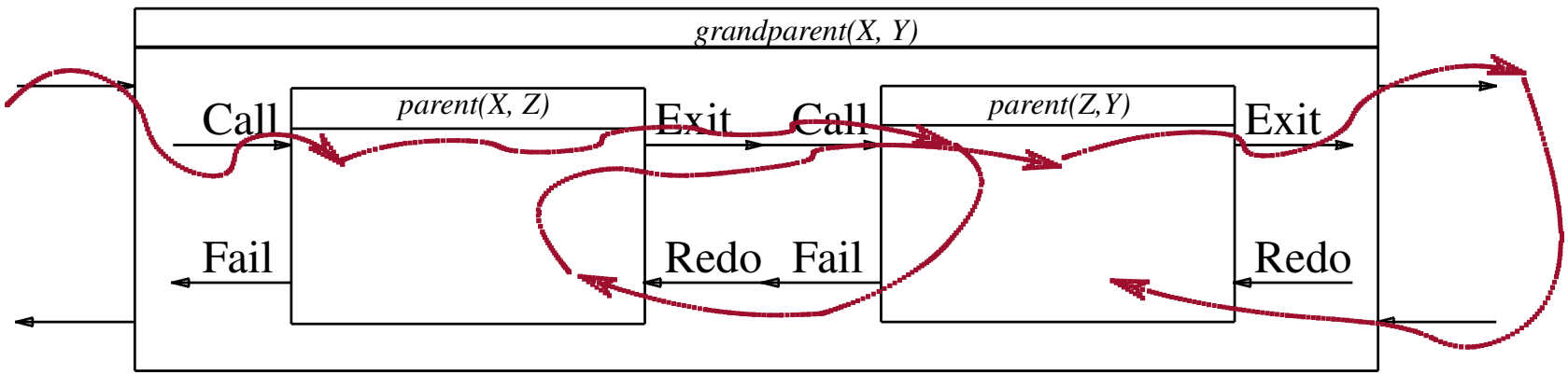
X=schmi, Z=anakin Y=luke



Ett programflöde

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```

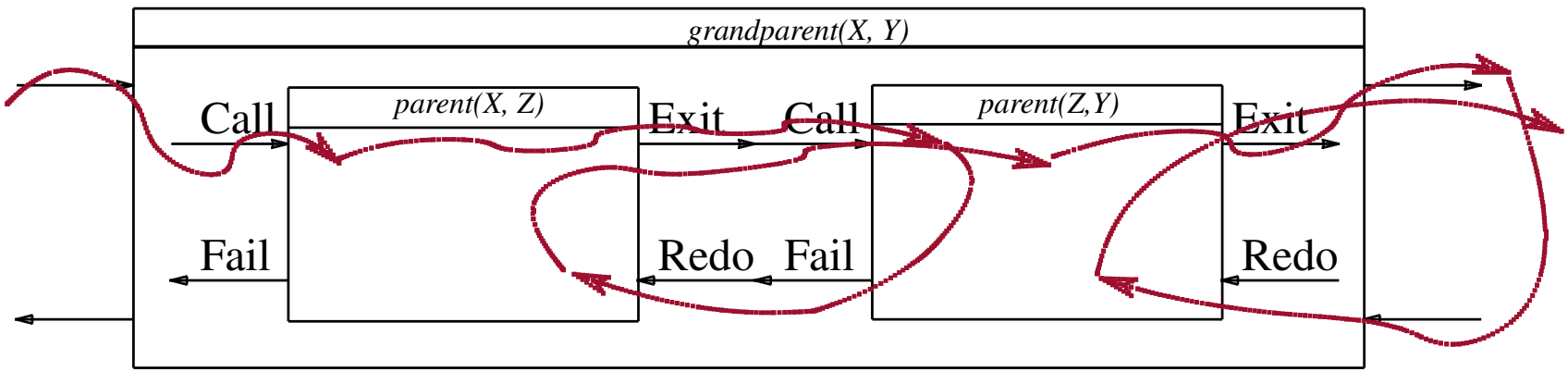
X=schmi, Z=anakin



Ett programflöde

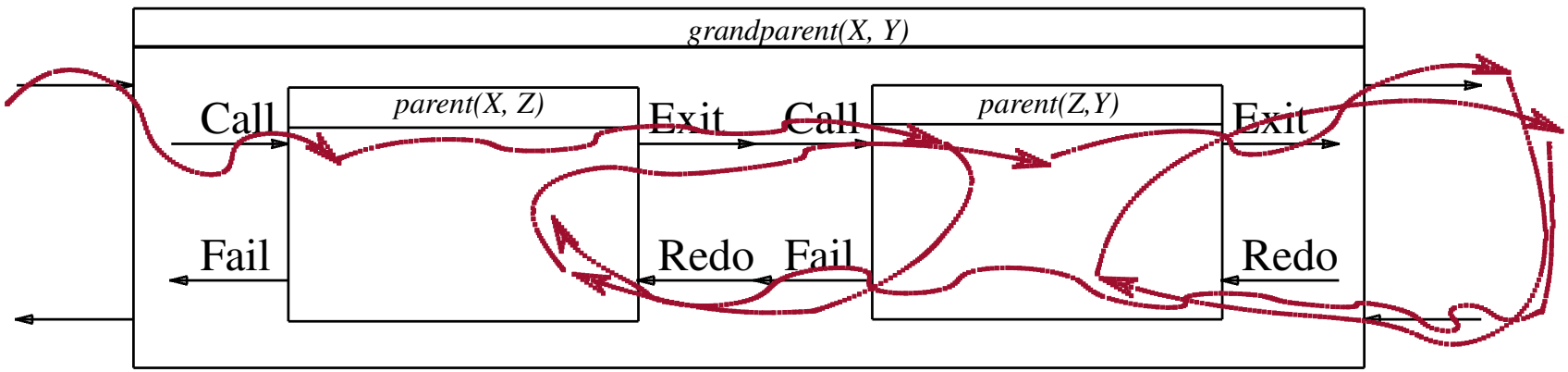
```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```

X=schmi, Z=anakin Y=leia



Ett programflöde

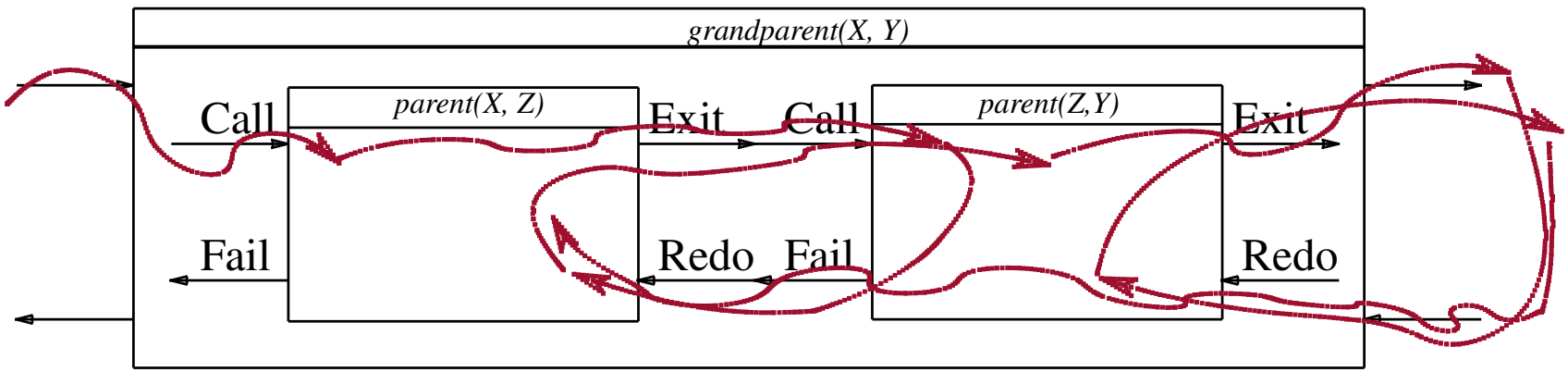
```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```



Ett programflöde

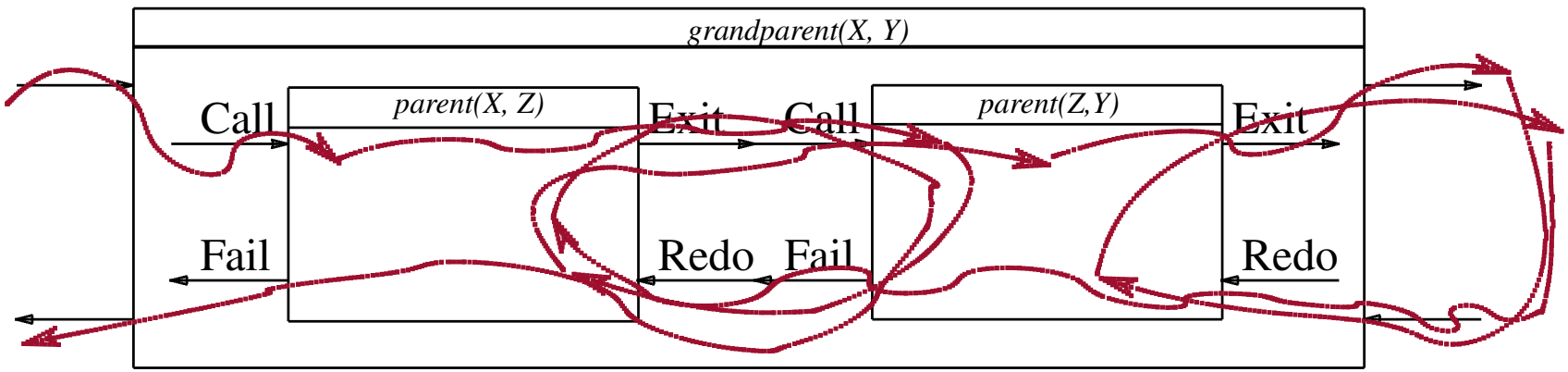
```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```

X=anakin, Z=leia



Ett programflöde

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
parent(anakin, luke).
parent(schmi, anakin).
parent(anakin, leia).
```



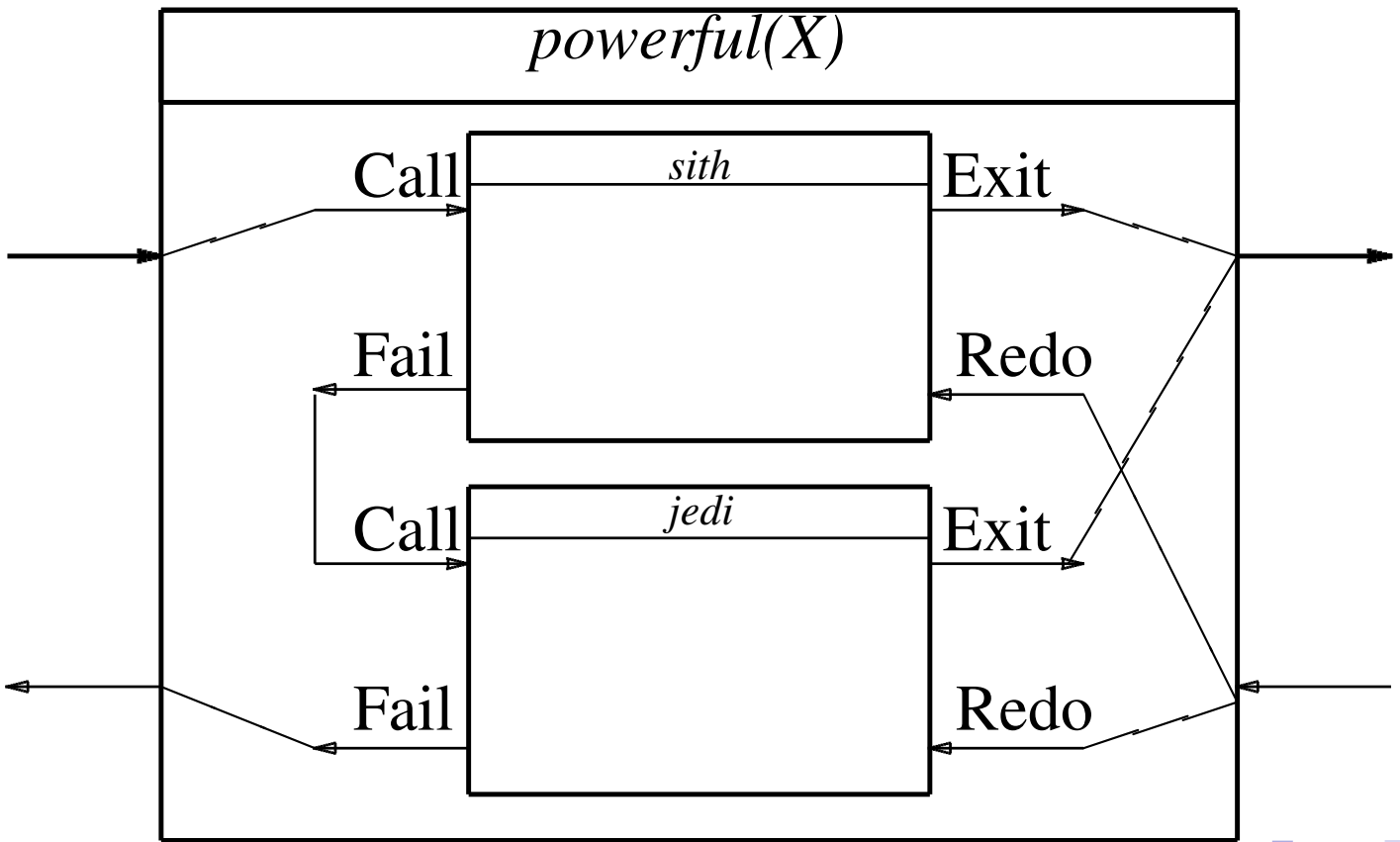
Lådmodellen för `powerful`

Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```



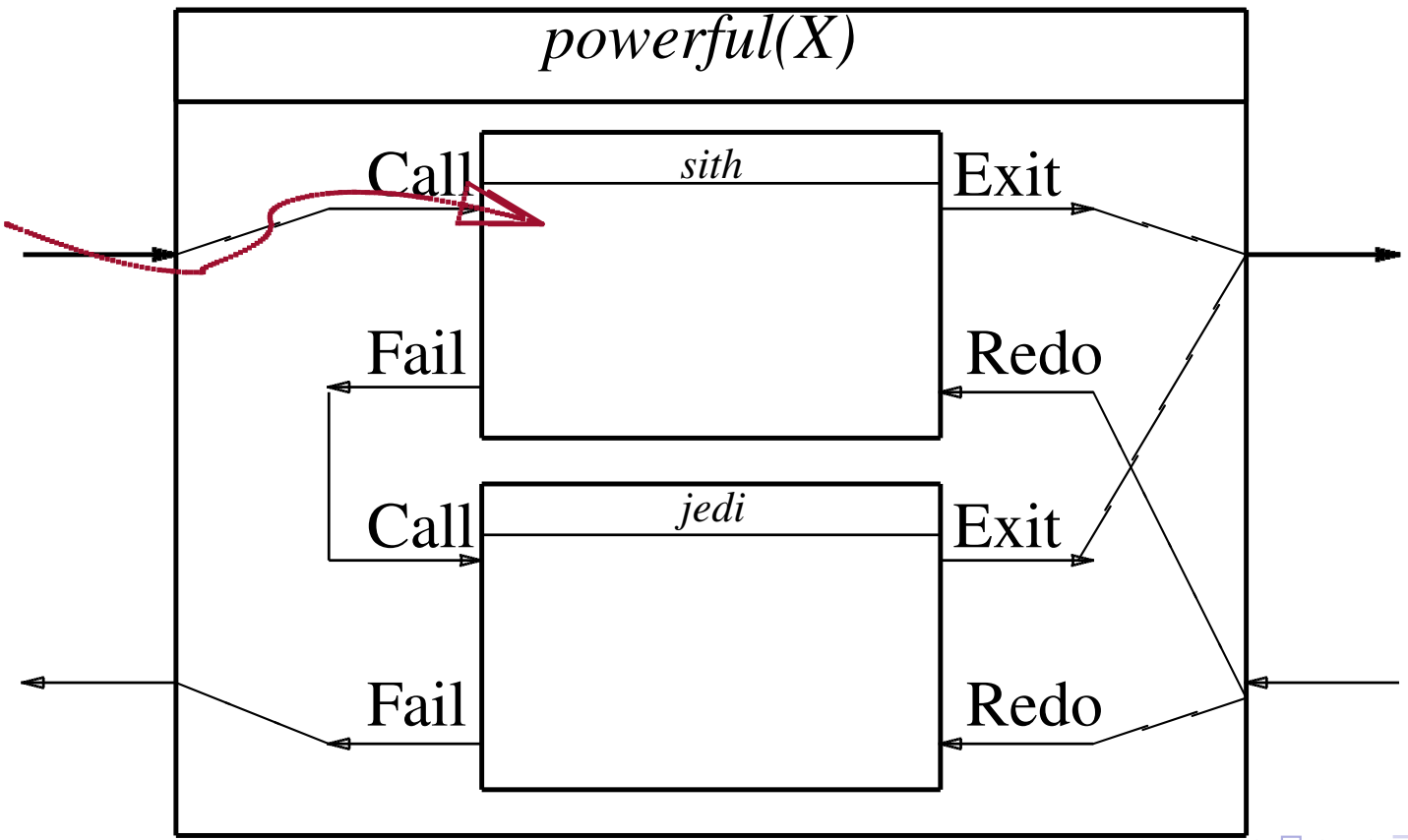
Lådmodellen för `powerful`

Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```



Lådmodellen för `powerful`

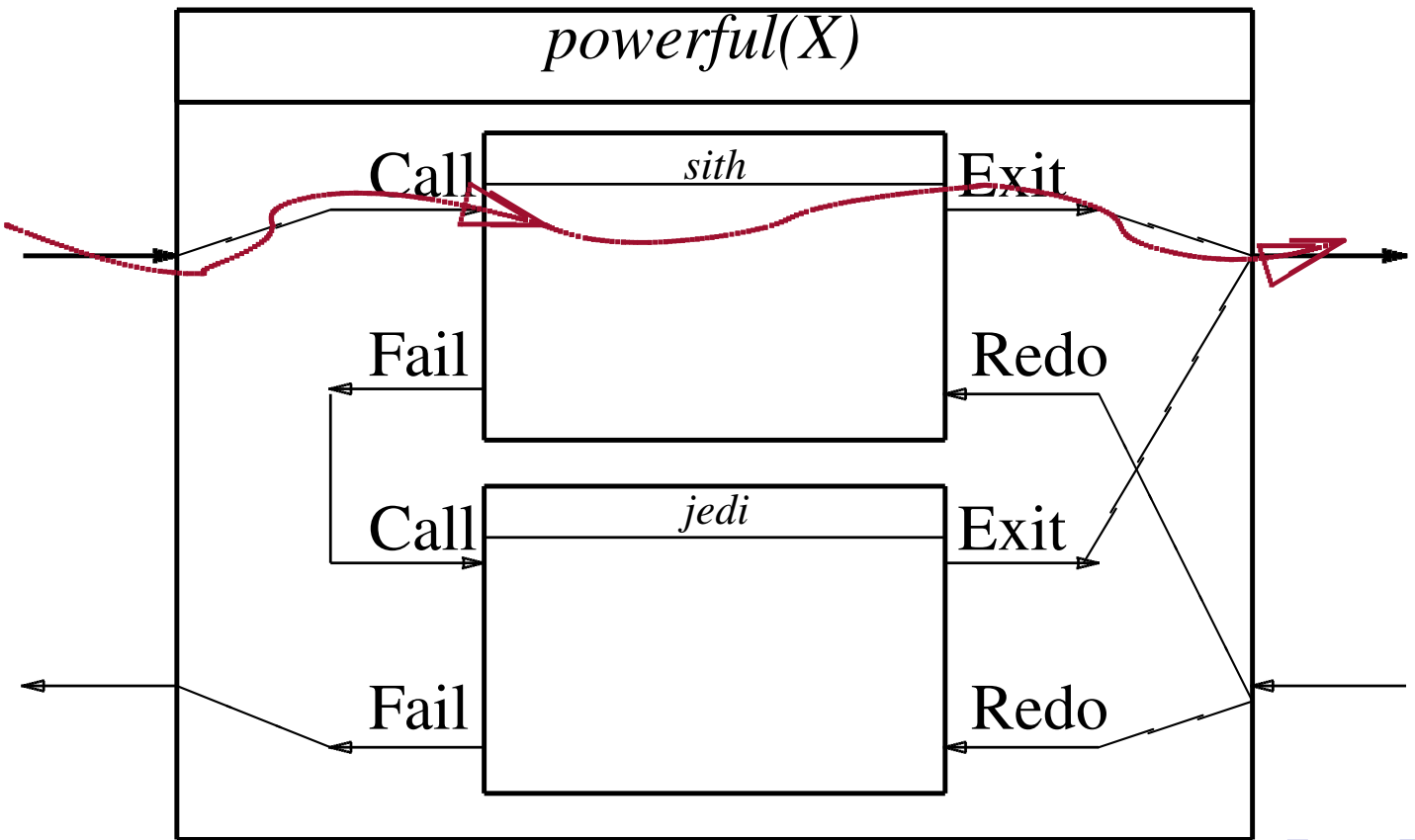
Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```

X=darthSidious



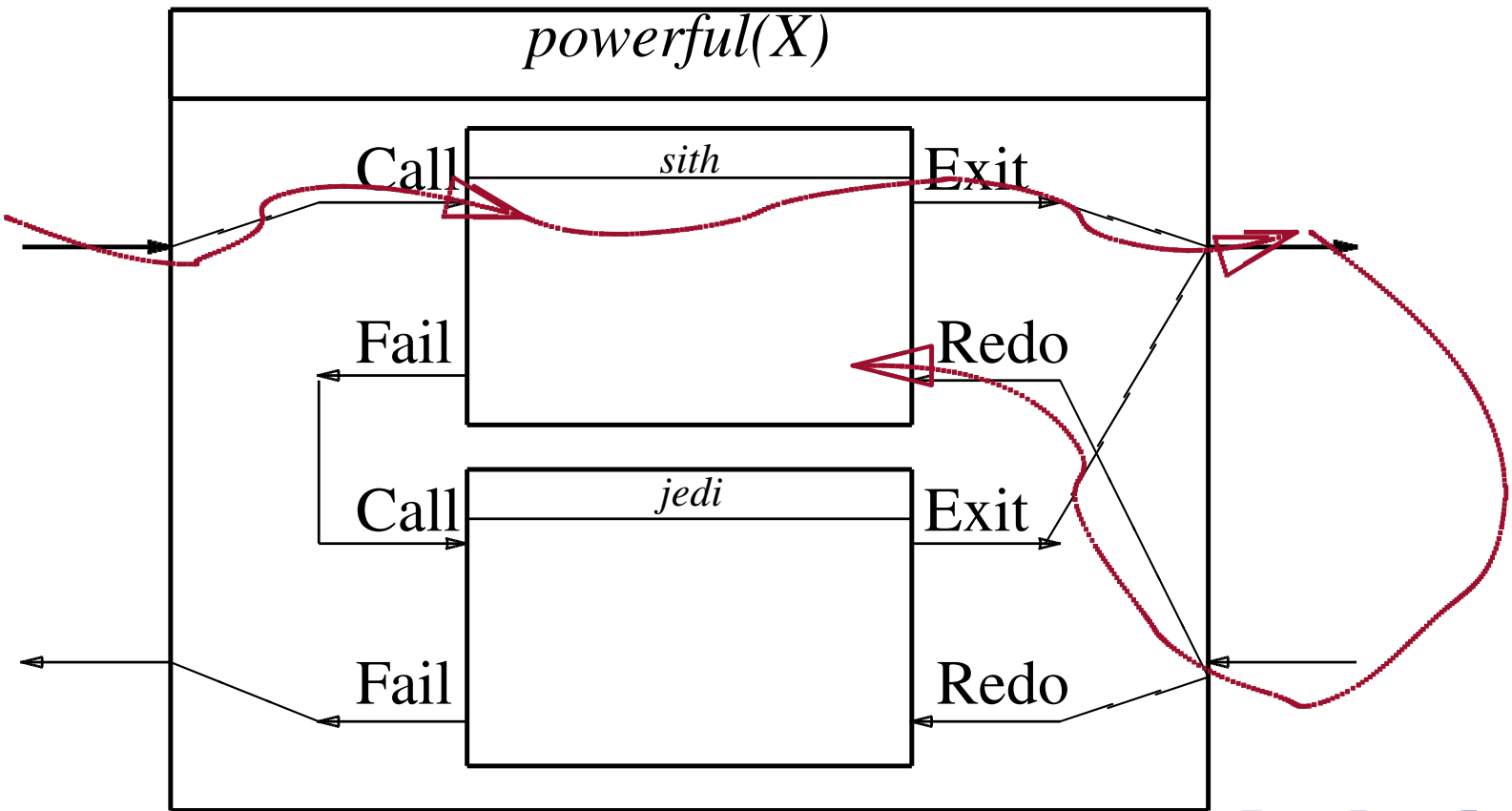
Lådmodellen för `powerful`

Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```



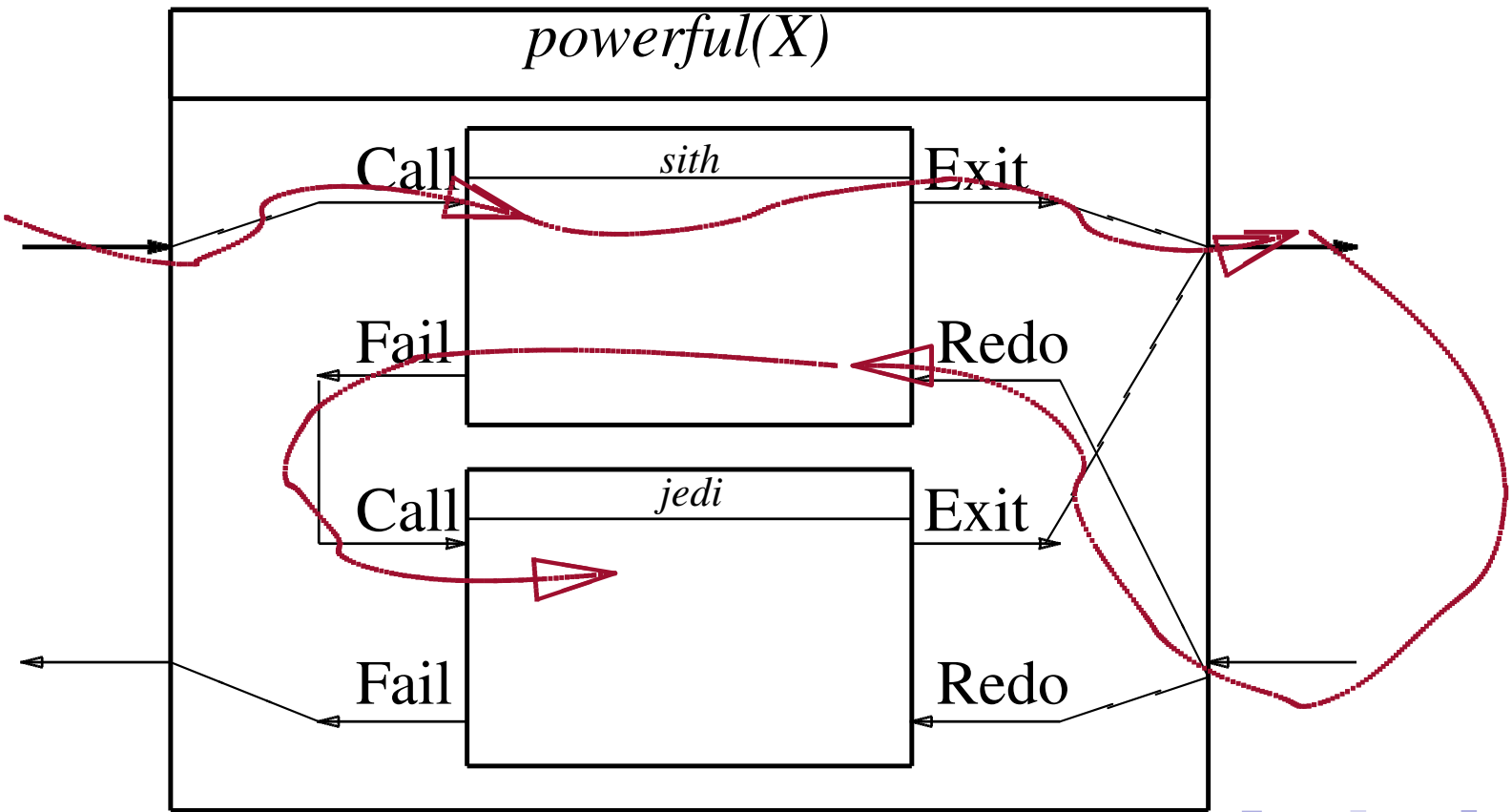
Lådmodellen för `powerful`

Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```



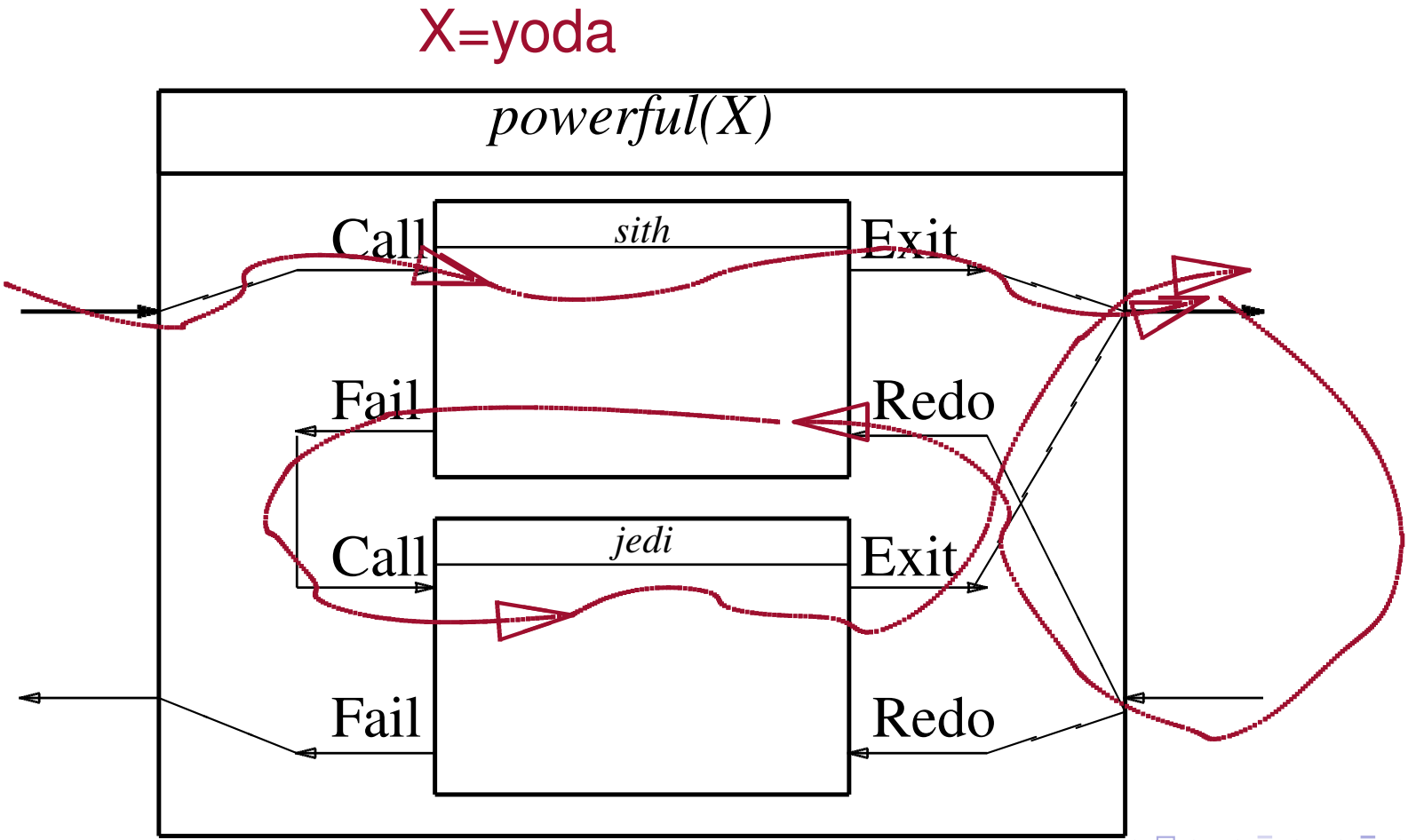
Lådmodellen för `powerful`

Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```



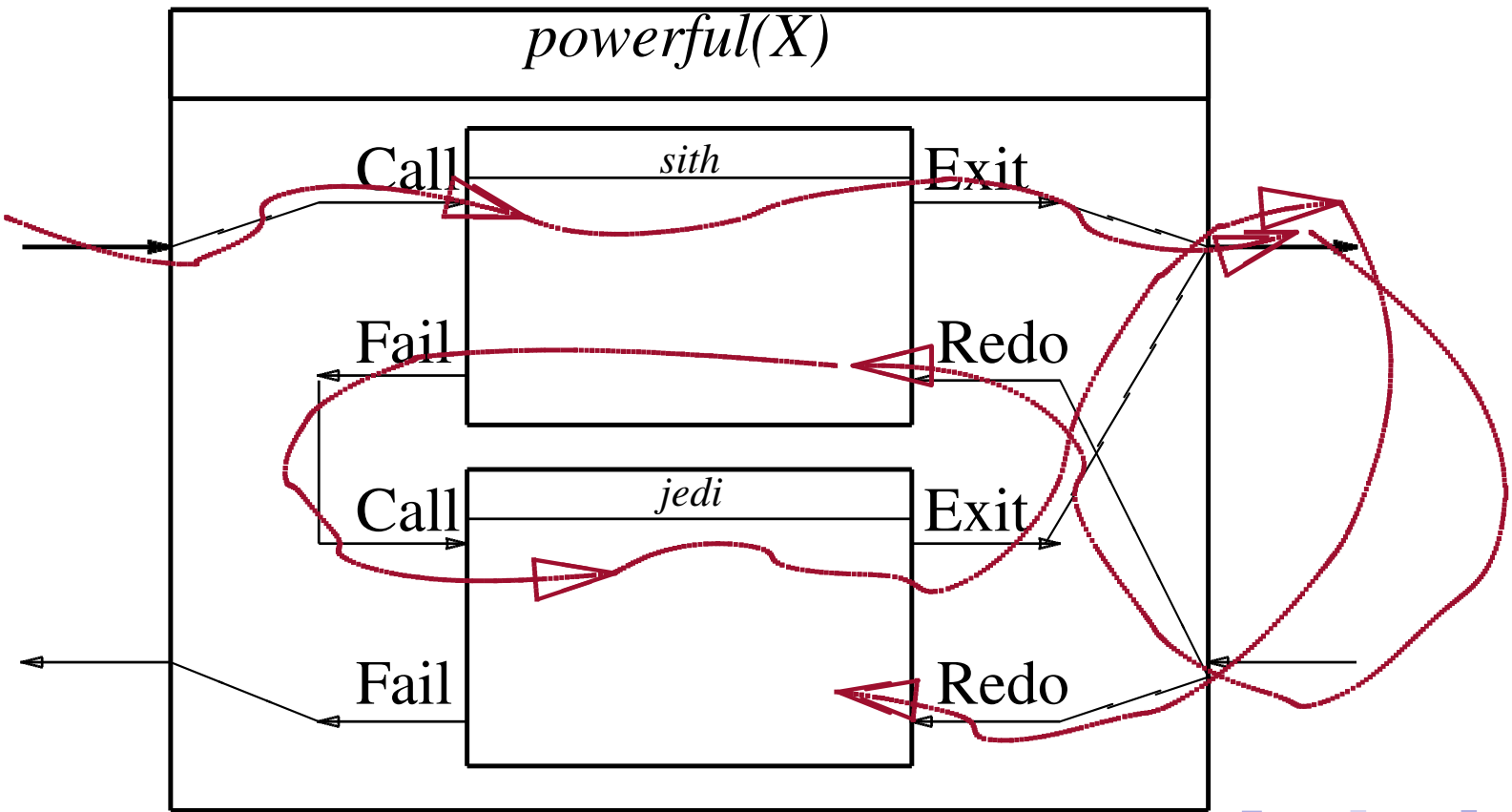
Lådmodellen för `powerful`

Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```



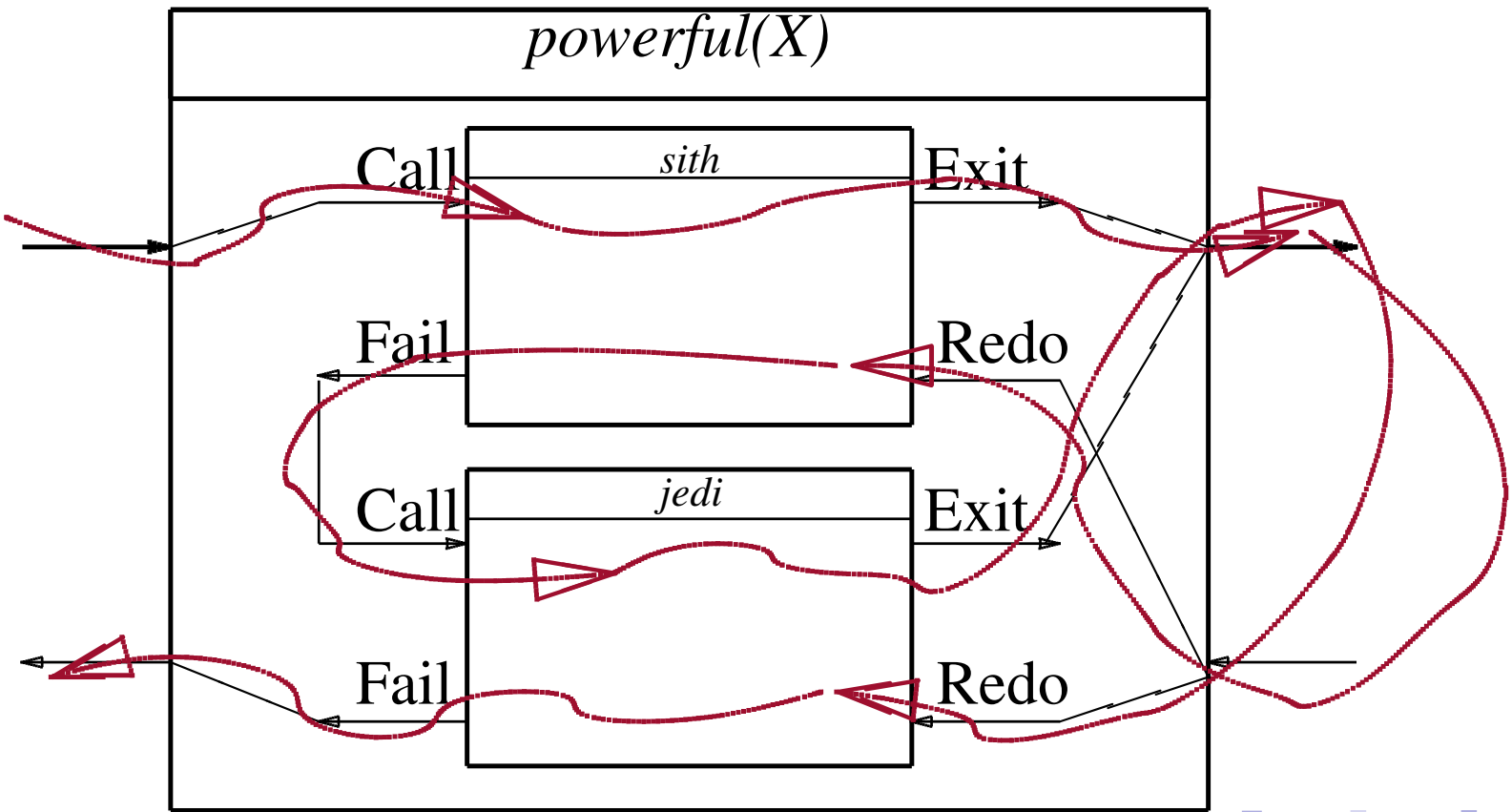
Lådmodellen för `powerful`

Regel

```
powerful(X) :- sith(X).
powerful(X) :- jedi(X).
```

Fakta

```
sith(darthSidious).
jedi(yoda).
```



Lådmodellen och rekursion

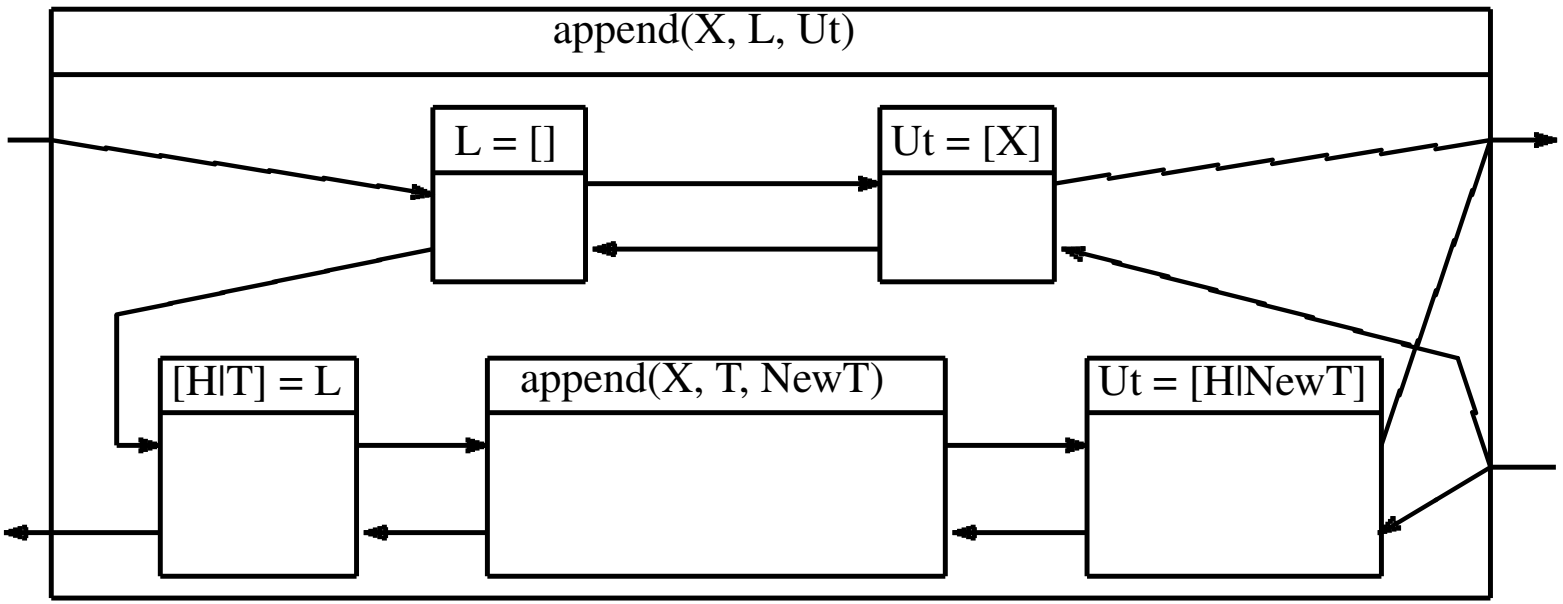
```
append(X, [], [X]).
append(X, [H|Tail], [H|NewTail]) :-
    append(X, Tail, NewTail).
```

Lådmodellen och rekursion

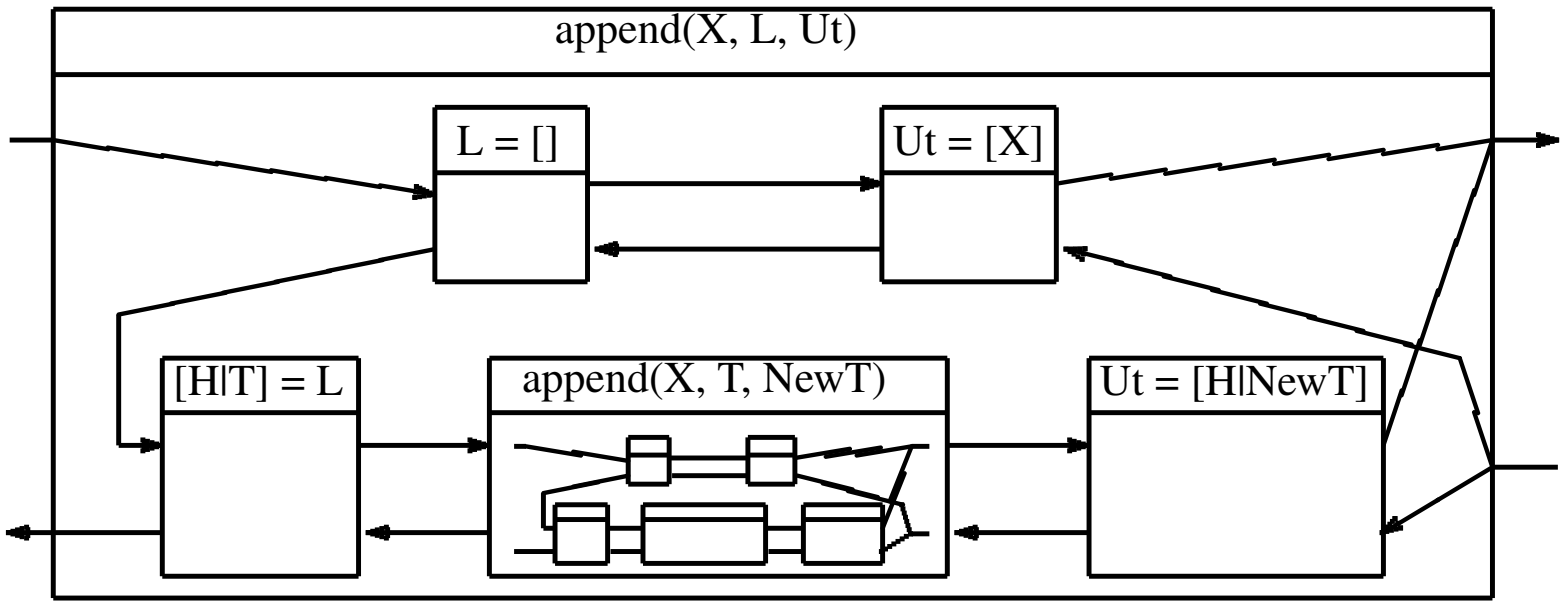
```
append(X, [], [X]).
append(X, [H|Tail], [H|NewTail]) :-
    append(X, Tail, NewTail).
```

```
append(X, L, Ut) :- L = [], Ut = [X].
append(X, L, Ut) :-
    [H|Tail] = L,
    append(X, Tail, NewTail),
    Ut = [H|NewTail].
```

Lådmodell för `append`



Lådmodell för `append`



Avlusning med SWI Prolog

- Använder boxmodellen (utökad)

Avlusning med SWI Prolog

- Använder boxmodellen (utökad)
- Starta debug med `trace.`

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
```

Avlusning med SWI Prolog

- Använder boxmodellen (utökad)
- Starta debug med `trace.`

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
```

- Kör i Emacs så markerar SWI Prolog var du är!

Avlusning med SWI Prolog

- Använder boxmodellen (utökad)
- Starta debug med `trace.`

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
```

- Kör i Emacs så markerar SWI Prolog var du är!
- SWI Prolog stannar *i varje port*.

Avlusning med SWI Prolog

- Använder boxmodellen (utökad)
- Starta debug med `trace.`

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
```

- Kör i Emacs så markerar SWI Prolog var du är!
- SWI Prolog stannar *i varje port*.
- Bestäm vilka portar som SWI Prolog ska stanna vid med `leash`:

```
| ?- leash([exit,fail]).
% Using leashing stopping at [exit,fail] ports
yes
```

Debuginformation

- Unikt ”invocation number”
- Rekursionsdjup
- Aktuell port
- Aktuell mål, unifieringar

Exempel: spåra grandparent/2

```
|?- grandparent(X, Y).
```

Exempel: spåra grandparent/2

```
|?- grandparent(X, Y).
        1    1 Call: grandparent(_430,_450)
        2    2 Call: parent(_430,_636)
?       2    2 Exit: parent(schmi,anakin)
        3    2 Call: parent(anakin,_450)
?       3    2 Exit: parent(anakin,luke)
?       1    1 Exit: grandparent(schmi,luke)
X = schmi,
Y = luke ?
```

Exempel: spåra grandparent/2

```
|?- grandparent(X, Y).
        1    1 Call: grandparent(_430,_450)
        2    2 Call: parent(_430,_636)
?       2    2 Exit: parent(schmi,anakin)
        3    2 Call: parent(anakin,_450)
?       3    2 Exit: parent(anakin,luke)
?       1    1 Exit: grandparent(schmi,luke)
X = schmi,
Y = luke ? n
        1    1 Redo: grandparent(schmi,luke)
        3    2 Redo: parent(anakin,luke)
        3    2 Exit: parent(anakin,leia)
?       1    1 Exit: grandparent(schmi,leia)
X = schmi,
Y = leia ?
yes
```

Avlusa married/2

Program:

```
married(X,Y) :- married(Y,X).
married(anakin, padme).
```

```
| ?-
```

Avlusa married/2

Program:

```
married(X,Y) :- married(Y,X).
married(anakin, padme).
```

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
% trace,source_info
| ?-
```

Avlusa married/2

Program:

```
married(X,Y) :- married(Y,X).
married(anakin, padme).
```

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
% trace,source_info
| ?- married(X, Y).
      1    1 Call: married(_410,_430) ?
```

Avlusa married/2

Program:

```
married(X,Y) :- married(Y,X).
married(anakin, padme).
```

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
% trace,source_info
| ?- married(X, Y).
        1    1 Call: married(_410,_430) ?
        2    2 Call: married(_430,_410) ?
```

Avlusa married/2

Program:

```
married(X,Y) :- married(Y,X).
married(anakin, padme).
```

```
| ?- trace.
% The debugger will first creep -- showing eve
yes
% trace,source_info
| ?- married(X, Y).
        1    1 Call: married(_410,_430) ?
        2    2 Call: married(_430,_410) ?
        3    3 Call: married(_410,_430) ?
        4    4 Call: married(_430,_410) ?
        5    5 Call: married(_410,_430) ?
```