Effective Modeling in Answer Set Programming modulo Theories

Martin Gebser

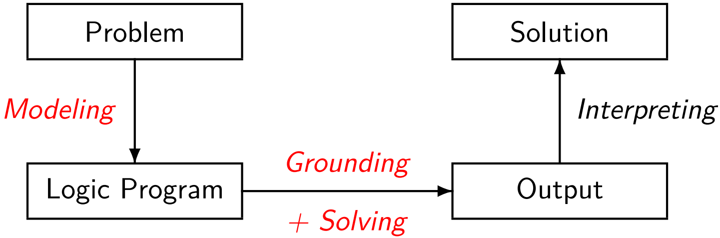
University of Klagenfurt   Graz University of Technology

thanks to:

Dirk Abels, Mohammed El-Kholany, Giulia Francescutto, Gerhard Friedrich, Clémence Frioux, Christian Haubelt, Tomi Janhunen, Julian Jordi, Roland Kaminski, Kai Neubauer, Max Ostrowski, Konstantin Schekotihin, Sebastian Schellhorn, Anne Siegel, Erich Teppan, Ambra Toletti, Torsten Schaub, Philipp Wanko

Answer Set Programming

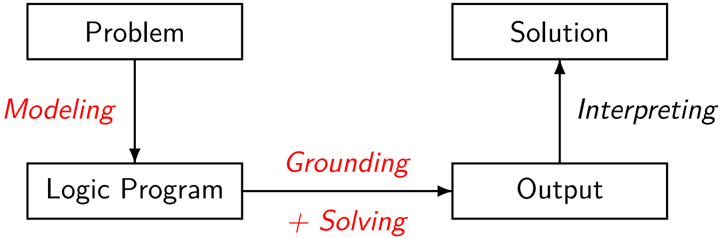
- Answer Set Programming (ASP) offers expressive first-order modeling language and powerful reasoning technology
  - Ground instantiation by semi-naive database evaluation [KS23]
  - Search/Optimization by conflict-driven learning [ADMR20, GKS12]



- Uniform problem representations separate instance data from high-level problem encoding for elaboration-tolerant modeling
  - $\text{Logic Program} = \underbrace{\text{Facts}}_{\text{Instance}} + \underbrace{\text{Generate} + \text{Define} + \text{Test} + \text{Optimize}}_{\text{Encoding}}$

Answer Set Programming

- Answer Set Programming (ASP) offers expressive first-order modeling language and powerful reasoning technology
  - Ground instantiation by semi-naive database evaluation [KS23]
  - Search/Optimization by conflict-driven learning [ADMR20, GKS12]



- Uniform problem representations separate instance data from high-level problem encoding for elaboration-tolerant modeling
  - $\text{Logic Program} = \underbrace{\text{Facts}}_{\text{Instance}} + \underbrace{\text{Generate} + \text{Define} + \text{Test} + \text{Optimize}}_{\text{Encoding}}$

Answer Set Programming modulo Theories

- Large domains, even if discrete, lead to grounding bottleneck
- Even linear/logarithmic propositional representation by binary or order encoding [CB94, War98] prohibitive for large ranges
  - Quantitative resources, spatial coordinates, time horizons
- Extensions like Constraint Answer Set Programming (CASP) [Lie23] enable succinct representations of integer/real values
- Difference Logic (DL) constitutes a tractable CASP fragment
  - `clingo-dl` [JKO+17] supplies propagator, similar to stability [GKS12], acyclicity [BGJ+16], or constraint [CDRS20] checking

☞ Inspiration by Satisfiability modulo Theories (SMT) [BSST09]

From plain ASP to ASP modulo DL

```
% at(Y) = at(X) + D
at(Y,T+D) :- at(X,T), duration(X,Y,D).
```

Answer Set Programming modulo Theories

- Large domains, even if discrete, lead to grounding bottleneck
- Even linear/logarithmic propositional representation by binary or order encoding [CB94, War98] prohibitive for large ranges
  - Quantitative resources, spatial coordinates, time horizons
- Extensions like Constraint Answer Set Programming (CASP) [Lie23] enable succinct representations of integer/real values
- Difference Logic (DL) constitutes a tractable CASP fragment
  - `clingo-dl` [JKO$^+$17] supplies propagator, similar to stability [GKS12], acyclicity [BGJ$^+$16], or constraint [CDRS20] checking

☞ Inspiration by Satisfiability modulo Theories (SMT) [BSST09]

From plain ASP to ASP modulo DL

```
% at(Y) = at(X) + D
at(Y,T+D) :- at(X,T), duration(X,Y,D).
```

Answer Set Programming modulo Theories

- Large domains, even if discrete, lead to grounding bottleneck
- Even linear/logarithmic propositional representation by binary or order encoding [CB94, War98] prohibitive for large ranges
  - Quantitative resources, spatial coordinates, time horizons
- Extensions like Constraint Answer Set Programming (CASP) [Lie23] enable succinct representations of integer/real values
- Difference Logic (DL) constitutes a tractable CASP fragment
  - `clingo-dl` [JKO+17] supplies propagator, similar to stability [GKS12], acyclicity [BGJ+16], or constraint [CDRS20] checking

☞ Inspiration by Satisfiability modulo Theories (SMT) [BSST09]

From plain ASP to ASP modulo DL

```
% at(Y) = at(X) + D
at(Y,T+D) :- at(X,T), duration(X,Y,D).
```

Answer Set Programming modulo Theories

- Large domains, even if discrete, lead to grounding bottleneck
- Even linear/logarithmic propositional representation by binary or order encoding [CB94, War98] prohibitive for large ranges
  - Quantitative resources, spatial coordinates, time horizons
- Extensions like Constraint Answer Set Programming (CASP) [Lie23] enable succinct representations of integer/real values
- Difference Logic (DL) constitutes a tractable CASP fragment
  - `clingo-dl` [JKO+17] supplies propagator, similar to stability [GKS12], acyclicity [BGJ+16], or constraint [CDRS20] checking

☞ Inspiration by Satisfiability modulo Theories (SMT) [BSST09]

From plain ASP to ASP modulo DL

```
% at(X) - at(Y) <= -D  iff  at(Y) >= at(X) + D
&diff{at(X) - at(Y)} <= -D :- duration(X,Y,D).
```

Answer Set Programming modulo Theories

- Large domains, even if discrete, lead to grounding bottleneck
- Even linear/logarithmic propositional representation by binary or order encoding [CB94, War98] prohibitive for large ranges
  - Quantitative resources, spatial coordinates, time horizons
- Extensions like Constraint Answer Set Programming (CASP) [Lie23] enable succinct representations of integer/real values
- Difference Logic (DL) constitutes a tractable CASP fragment
  - `clingo-dl` [JKO+17] supplies propagator, similar to stability [GKS12], acyclicity [BGJ+16], or constraint [CDRS20] checking

☞ Inspiration by Satisfiability modulo Theories (SMT) [BSST09]

From plain ASP to ASP modulo DL

```
% at(X) - at(Y) <= -D  iff  at(Y) >= at(X) + D
&diff{at(X) - at(Y)} <= -D :- duration(X,Y,D).
```

Flexible Job-Shop Scheduling Problem (FJSSP)

- Classical NP-hard optimization problem [BEP+14]
- Various applications in manufacturing and logistics [XGP+19]
- Benchmark for Answer Set Programming (ASP), Constraint Programming (CP), Satisfiability (SAT) solver competitions

Problem Specification

- An instance provides jobs and machines to process operations
  - Jobs = sequences of operations
  - An operation's processing time can vary between machines
- A schedule assigns a machine and start time to each operation
  - Processing times of operations on same machine do not overlap
  - Successor operations don't start before predecessor completion
- Minimization of makespan = maximum job completion time

☞ Discrete start times give rise to potentially large integer ranges

Flexible Job-Shop Scheduling Problem (FJSSP)

- Classical NP-hard optimization problem [BEP+14]
- Various applications in manufacturing and logistics [XGP+19]
- Benchmark for Answer Set Programming (ASP), Constraint Programming (CP), Satisfiability (SAT) solver competitions

Problem Specification

- An instance provides jobs and machines to process operations
  - Jobs = sequences of operations
  - An operation's processing time can vary between machines
- A schedule assigns a machine and start time to each operation
  - Processing times of operations on same machine do not overlap
  - Successor operations don't start before predecessor completion
- Minimization of makespan = maximum job completion time

☞ Discrete start times give rise to potentially large integer ranges

Flexible Job-Shop Scheduling Problem (FJSSP)

- Classical NP-hard optimization problem [BEP+14]
- Various applications in manufacturing and logistics [XGP+19]
- Benchmark for Answer Set Programming (ASP), Constraint Programming (CP), Satisfiability (SAT) solver competitions

Problem Specification

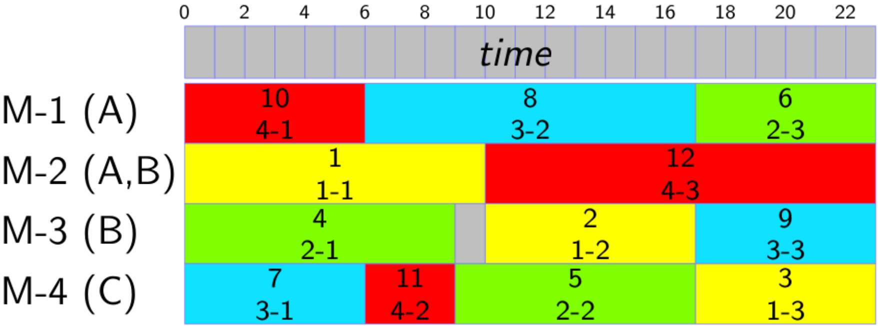
- An instance provides jobs and machines to process operations
  - Jobs = sequences of operations
  - An operation's processing time can vary between machines
- A schedule assigns a machine and start time to each operation
  - Processing times of operations on same machine do not overlap
  - Successor operations don't start before predecessor completion
- Minimization of makespan = maximum job completion time

☞ Discrete start times give rise to potentially large integer ranges

FJSSP Instance

OpId	JobId	Pos	Type	M-1	M-2	M-3	M-4
1	1	1	A	11	10		
2	1	2	B		6	7	
3	1	3	C				6
4	2	1	B		8	9	
5	2	2	C				8
6	2	3	A	6	5		
7	3	1	C				6
8	3	2	A	11	10		
9	3	3	B		5	6	
10	4	1	A	6	5		
11	4	2	C				3
12	4	3	B		13	14	

Optimal FJSSP Solution



☞ Makespan: 23 time units

Instance Representation

```
operation(1,1,1).   mode(1,1,11).   mode(1,2,10).
operation(2,1,2).   mode(2,2,6).    mode(2,3,7).
operation(3,1,3).   mode(3,4,6).

operation(4,2,1).   mode(4,2,8).    mode(4,3,9).
operation(5,2,2).   mode(5,4,8).
operation(6,2,3).   mode(6,1,6).    mode(6,2,5).

operation(7,3,1).   mode(7,4,6).
operation(8,3,2).   mode(8,1,11).   mode(8,2,10).
operation(9,3,3).   mode(9,2,5).    mode(9,3,6).

operation(10,4,1).  mode(10,1,6).   mode(10,2,5).
operation(11,4,2).  mode(11,4,3).
operation(12,4,3).  mode(12,2,13).  mode(12,3,14).

time(0..50).
```

Instance Representation

```
operation(1,1,1).    mode(1,1,11).    mode(1,2,10).
operation(2,1,2).    mode(2,2,6).     mode(2,3,7).
operation(3,1,3).    mode(3,4,6).

operation(4,2,1).    mode(4,2,8).     mode(4,3,9).
operation(5,2,2).    mode(5,4,8).
operation(6,2,3).    mode(6,1,6).     mode(6,2,5).

operation(7,3,1).    mode(7,4,6).
operation(8,3,2).    mode(8,1,11).    mode(8,2,10).
operation(9,3,3).    mode(9,2,5).     mode(9,3,6).

operation(10,4,1).   mode(10,1,6).    mode(10,2,5).
operation(11,4,2).   mode(11,4,3).
operation(12,4,3).   mode(12,2,13).   mode(12,3,14).

time(0..50).
```

Vanilla ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 != J2.

% Choose the operation start times
{start(X,S) : time(S)} = 1 :- operation(X,J,N).

% Derive the operation end times
end(X,S+P) :- start(X,S), process(X,_,P).

% Operations (of distinct jobs) must be processed without overlaps
:- ordered(X,Y), start(X,S1), start(Y,S2), end(X,E1), S1 <= S2, S2 < E1.

% Successor operation must not start before end of its predecessor
:- operation(X,J,N), operation(Y,J,N+1), start(Y,S), end(X,E), S < E.

% Derive and minimize the makespan
makespan(K) :- K = #max{E : end(X,E),
                            operation(X,J,N), not operation(X+1,J,N+1)}.
:~ makespan(K). [K]
```

Vanilla ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 != J2.

% Choose the operation start times
{start(X,S) : time(S)} = 1 :- operation(X,J,N).

% Derive the operation end times
end(X,S+P) :- start(X,S), process(X,_,P).

% Operations (of distinct jobs) must be processed without overlaps
:- ordered(X,Y), start(X,S1), start(Y,S2), end(X,E1), S1 <= S2, S2 < E1.

% Successor operation must not start before end of its predecessor
:- operation(X,J,N), operation(Y,J,N+1), start(Y,S), end(X,E), S < E.

% Derive and minimize the makespan
makespan(K) :- K = #max{E : end(X,E),
                            operation(X,J,N), not operation(X+1,J,N+1)}.
:~ makespan(K). [K]
```

Vanilla ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 != J2.

% Choose the operation start times
{start(X,S) : time(S)} = 1 :- operation(X,J,N).

% Derive the operation end times
end(X,S+P) :- start(X,S), process(X,_,P).

% Operations (of distinct jobs) must be processed without overlaps
:- ordered(X,Y), start(X,S1), start(Y,S2), end(X,E1), S1 <= S2, S2 < E1.

% Successor operation must not start before end of its predecessor
:- operation(X,J,N), operation(Y,J,N+1), start(Y,S), end(X,E), S < E.

% Derive and minimize the makespan
makespan(K) :- K = #max{E : end(X,E),
                            operation(X,J,N), not operation(X+1,J,N+1)}.
:~ makespan(K). [K]
```

Vanilla ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 != J2.

% Choose the operation start times
{start(X,S) : time(S)} = 1 :- operation(X,J,N).

% Derive the operation end times
end(X,S+P) :- start(X,S), process(X,_,P).

% Operations (of distinct jobs) must be processed without overlaps
:- ordered(X,Y), start(X,S1), start(Y,S2), end(X,E1), S1 <= S2, S2 < E1.

% Successor operation must not start before end of its predecessor
:- operation(X,J,N), operation(Y,J,N+1), start(Y,S), end(X,E), S < E.

% Derive and minimize the makespan
makespan(K) :- K = #max{E : end(X,E),
                            operation(X,J,N), not operation(X+1,J,N+1)}.
:~ makespan(K). [K]
```

Let's Run!

clingo instance.lp vanilla.lp --stats

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... start(12,41)
...
Answer: 28
process(1,2,10) process(2,2,6) ... start(12,9)
Optimization: 23
OPTIMUM FOUND

Time          : 8.981s (Solving: 3.69s ...)
Variables     : 2099
Constraints   : 1980906
```

☞ Highly problematic for ground instantiation size:

```
:- ordered(X,Y), start(X,S1), start(Y,S2), end(X,E1), S1 <= S2, S2 < E1.
```

Let's Run!

clingo instance.lp vanilla.lp --stats

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... start(12,41)
...
Answer: 28
process(1,2,10) process(2,2,6) ... start(12,9)
Optimization: 23
OPTIMUM FOUND

Time          : 8.981s (Solving: 3.69s ...)
Variables     : 2099
Constraints   : 1980906
```

☞ Highly problematic for ground instantiation size:

```
:- ordered(X,Y), start(X,S1), start(Y,S2), end(X,E1), S1 <= S2, S2 < E1.
```

Compact ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 != J2.

% Choose the operation start times
{start(X,S) : time(S)} = 1 :- operation(X,J,N).

% Derive the operation end times
end(X,S+P) :- start(X,S), process(X,_,P).

% Operations (of distinct jobs) must be processed without overlaps
:- ordered(X,Y), start(X,S1), start(Y,S2), end(X,E1), S1 <= S2, S2 < E1.

% Successor operation must not start before end of its predecessor
:- operation(X,J,N), operation(Y,J,N+1), start(Y,S), end(X,E), S < E.

% Derive and minimize the makespan
makespan(K) :- K = #max{E : end(X,E),
                            operation(X,J,N), not operation(X+1,J,N+1)}.
:~ makespan(K). [K]
```

Compact ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
start(X,0) :- operation(X,J,1).
start(Y,E) :- end(X,E), order(X,Y).
start(Y,E) :- end(X,E), operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

Compact ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
start(X,0) :- operation(X,J,1).
start(Y,E) :- end(X,E), order(X,Y).
start(Y,E) :- end(X,E), operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

Compact ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
start(X,0) :- operation(X,J,1).
start(Y,E) :- end(X,E), order(X,Y).
start(Y,E) :- end(X,E), operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

Compact ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
start(X,0) :- operation(X,J,1).
start(Y,E) :- end(X,E), order(X,Y).
start(Y,E) :- end(X,E), operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

Compact ASP Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
start(X,0) :- operation(X,J,1).
start(Y,E) :- end(X,E), order(X,Y).
start(Y,E) :- end(X,E), operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

Let's Run Again!

clingo instance.lp compact.lp --stats & time(0..5000)

```
Solving...
Answer: 1
process(1,1,11) process(2,2,6) ... start(12,8)
...
Answer: 12
process(1,2,10) process(2,2,6) ... start(12,9)
Optimization: 23
OPTIMUM FOUND

Time          : 0.026s (Solving: 0.01s ...)
Variables     : 3816
Constraints   : 12713
```

☞ Many time points highly problematic for ground instantiation size

Let's Run Again!

clingo instance.lp compact.lp --stats & time(0..5000)

```
Solving...
Answer: 1
process(1,2,10) process(2,2,6) ... start(12,8)
...
Answer: 5
process(1,2,10) process(2,2,6) ... start(12,9)
Optimization: 23
OPTIMUM FOUND

Time          : 3.235s (Solving: 1.46s ...)
Variables     : 483966
Constraints   : 1685813
```

☞ Many time points highly problematic for ground instantiation size

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
start(X,0) :- operation(X,J,1).
start(Y,E) :- end(X,E), order(X,Y).
start(Y,E) :- end(X,E), operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
start(X) >= 0      :- operation(X,J,1).
start(Y) >= end(X) :- order(X,Y).
start(Y) >= end(X) :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X,S+P) :- start(X,S), process(X,_,P), time(S).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
end(X) >= start(X) + P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
:- start(X,S), not time(S).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
start(X) <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                 time(S), not time(S+1).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Derive processing interval and minimize the makespan
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Derive the makespan
makespan >= end(X) :- operation(X,J,N), not operation(X+1,J,N+1).
```

ASP modulo DL Encoding

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Derive the makespan
&diff{end(X) - makespan} <= 0 :- operation(X,J,N), not operation(X+1,J,N+1).
```

Let's Run clingo-dl!

```
clingo-dl instance.lp difference.lp --stats
--minimize-variable=makespan & time(0..5000)
```

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... dl(start(12),9)
...
Answer: 11
process(1,2,10) process(2,3,7) ... dl(start(12),10)
DL Optimization: 23
UNSATISFIABLE

Time          : 0.013s (Solving: 0.00s ...)
Variables     : 141
Constraints   : 256
```

☞ Time range makes no difference for ground instantiation size

Let's Run `clingo-dl`!

```
clingo-dl instance.lp difference.lp --stats
--minimize-variable=makespan & time(0..5000)
```

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... dl(start(12),9)
...
Answer: 10
process(1,2,10) process(2,3,7) ... dl(start(12),10)
DL Optimization: 23
UNSATISFIABLE

Time          : 0.031s (Solving: 0.00s ...)
Variables     : 140
Constraints   : 199
```

☞ Time range makes no difference for ground instantiation size

And The Winner is ...

- Time points don't increase ground instantiation size with DL
- Makespan minimization refers to a single DL variable's value

☞ Perfect for: `clingo-dl --minimize-variable=makespan`

- Turning from makespan minimization

```
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

- Native support of (lexicographic) multi-criteria optimization:

```
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1@2,K]
```

- The optimization features provided for DL variables are not yet on a par with functionalities readily available in plain ASP

☞ Research and implementation progress needed to take burden of scripting custom DL optimization algorithms from the user

And The Winner is . . .

- Time points don't increase ground instantiation size with DL
- Makespan minimization refers to a single DL variable's value

☞ Perfect for: `clingo-dl --minimize-variable=makespan`

- Turning from makespan minimization

```
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1,K]
```

- Native support of (lexicographic) multi-criteria optimization:

```
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1@2,K]
```

- The optimization features provided for DL variables are not yet on a par with functionalities readily available in plain ASP

☞ Research and implementation progress needed to take burden of scripting custom DL optimization algorithms from the user

And The Winner is . . .

- Time points don't increase ground instantiation size with DL
- Makespan minimization refers to a single DL variable's value

☞ Perfect for: `clingo-dl --minimize-variable=makespan`

- Turning from makespan to total processing time minimization

```
:~ end(X,K), operation(X,J,N), not operation(X+1,J,N+1). [K,J]
```

  is easy to do in plain ASP

- Native support of (lexicographic) multi-criteria optimization:

```
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1@2,K]
```

- The optimization features provided for DL variables are not yet on a par with functionalities readily available in plain ASP

☞ Research and implementation progress needed to take burden of scripting custom DL optimization algorithms from the user

And The Winner is ...

- Time points don't increase ground instantiation size with DL
- Makespan minimization refers to a single DL variable's value

☞ Perfect for: `clingo-dl --minimize-variable=makespan`

- Turning from makespan to total processing time minimization

```
:~ end(X,K), operation(X,J,N), not operation(X+1,J,N+1). [K@1,J]
```

  is easy to do in plain ASP

- Native support of (lexicographic) multi-criteria optimization:

```
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1@2,K]
```

- The optimization features provided for DL variables are not yet on a par with functionalities readily available in plain ASP

☞ Research and implementation progress needed to take burden of scripting custom DL optimization algorithms from the user

And The Winner is . . .

- Time points don't increase ground instantiation size with DL
- Makespan minimization refers to a single DL variable's value

☞ Perfect for: `clingo-dl --minimize-variable=makespan`

- Turning from makespan to total processing time minimization

```
:~ end(X,K), operation(X,J,N), not operation(X+1,J,N+1). [K@1,J]
```

  is easy to do in plain ASP

- Native support of (lexicographic) multi-criteria optimization:

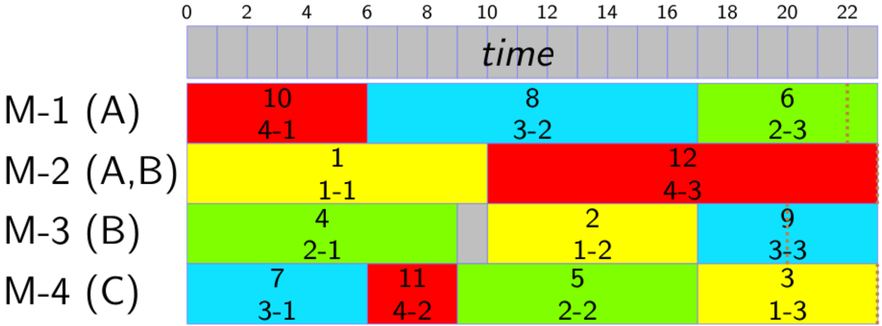
```
makespan(K) :- end(X,K), operation(X,J,N), not operation(X+1,J,N+1).
makespan(K) :- makespan(K+1), 0 < K.
:~ makespan(K). [1@2,K]
```

- The optimization features provided for DL variables are not yet on a par with functionalities readily available in plain ASP

☞ Research and implementation progress needed to take burden of scripting custom DL optimization algorithms from the user

Let's Add Job Dealines

```
deadline(1,23).   deadline(2,22).
deadline(3,21).   deadline(4,23).
```



☞ Job 2 is delayed by one and Job 3 by two time units

Minimize Number of Delayed Jobs

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Derive the makespan
&diff{end(X) - makespan} <= 0 :- operation(X,J,N), not operation(X+1,J,N+1).
```

Minimize Number of Delayed Jobs

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Minimize number of delayed jobs
:~ operation(X,J,N), not operation(X+1,J,N+1),
   deadline(J,D), not end(X) <= D. [1,J]
```

Minimize Number of Delayed Jobs

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Minimize number of delayed jobs
:~ operation(X,J,N), not operation(X+1,J,N+1),
   deadline(J,D), not end(X) <= D. [1,J]
```

Minimize Number of Delayed Jobs

```
% Choose one machine per operation
{process(X,M,P) : mode(X,M,P)} = 1 :- operation(X,J,N).

% Operations (of distinct jobs) on same machine have to be ordered
ordered(X,Y) :- operation(X,J1,N1), process(X,M,P1),
                operation(Y,J2,N2), process(Y,M,P2), J1 < J2.

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Minimize number of delayed jobs
:~ operation(X,J,N), not operation(X+1,J,N+1),
   deadline(J,D), not &diff{end(X) - 0} <= D. [1,J]
```

Let's Run with Delayed Job Minimization!

```
clingo-dl instance.lp deadline.lp differences.lp
--stats & time(0..1000)
```

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... dl(start(12),9)
    ...
Answer: 4
process(1,1,11) process(2,2,6) ... dl(start(12),10)
Optimization: 1
OPTIMUM FOUND

Time          : 0.002s (Solving: 0.00s ...)
Conflicts     : 37
Constraints   : 256
```

☞ Time range makes no difference for optimization performance

Let's Run with Delayed Job Minimization!

```
clingo-dl instance.lp deadline.lp differences.lp
--stats & time(0..1000)
```

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... dl(start(12),9)
    ...
Answer: 4
process(1,1,11) process(2,2,6) ... dl(start(12),10)
Optimization: 1
OPTIMUM FOUND

Time          : 0.005s (Solving: 0.00s ...)
Conflicts     : 43
Constraints   : 256
```

☞ Time range makes no difference for optimization performance

Minimize Sum of Job Delays

```
[...]

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Minimize number of delayed jobs
:~ operation(X,J,N), not operation(X+1,J,N+1),
   deadline(J,D), not &diff{end(X) - 0} <= D. [1,J]
```

Minimize Sum of Job Delays

```
[...]

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Minimize sum of job delays
delay(X,D,0..S+P-D-1) :- operation(X,J,N), not operation(X+1,J,N+1),
                         mode(X,M,P), deadline(J,D),
                         time(S), not time(S+1).

:~ delay(X,D,T), not end(X) <= D+T. [1,X,T]
```

Minimize Sum of Job Delays

```
[...]

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Minimize sum of job delays
delay(X,D,0..S+P-D-1) :- operation(X,J,N), not operation(X+1,J,N+1),
                         mode(X,M,P), deadline(J,D),
                         time(S), not time(S+1).

:~ delay(X,D,T), not end(X) <= D+T. [1,X,T]
```

Minimize Sum of Job Delays

```
[...]

% Choose an order of processing two operations on the same machine
{order(X,Y)} :- ordered(X,Y).
 order(Y,X)  :- ordered(X,Y), not order(X,Y).

% Derive lower bounds on operation start times
&diff{     0 - start(X)} <= 0 :- operation(X,J,1).
&diff{end(X) - start(Y)} <= 0 :- order(X,Y).
&diff{end(X) - start(Y)} <= 0 :- operation(X,J,N), operation(Y,J,N+1).

% Derive lower bounds on operation end times
&diff{start(X) - end(X)} <= -P :- process(X,_,P).

% Restrict lower bounds on start times to range given by time predicate
&diff{start(X) - 0} <= S :- operation(X,J,N), not operation(X+1,J,N+1),
                            time(S), not time(S+1).

% Minimize sum of job delays
delay(X,D,0..S+P-D-1) :- operation(X,J,N), not operation(X+1,J,N+1),
                         mode(X,M,P), deadline(J,D),
                         time(S), not time(S+1).

:~ delay(X,D,T), not &diff{end(X) - 0} <= D+T. [1,X,T]
```

Let's Run with Delay Sum Minimization!

clingo-dl instance.lp deadline.lp differencet.lp --stats & time(0..1000)

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... dl(start(12),9)
...
Answer: 54
process(1,2,10) process(2,2,6) ... dl(start(12),9)
Optimization: 2
OPTIMUM FOUND

Time           : 0.012s (Solving: 0.01s ...)
Conflicts      : 699
Constraints    : 256
```

☞ Time range may deteriorate optimization performance

Let's Run with Delay Sum Minimization!

clingo-dl instance.lp deadline.lp differencet.lp --stats & time(0..1000)

```
Solving...
Answer: 1
process(1,1,11) process(2,3,7) ... dl(start(12),9)
...
Answer: 3842
process(1,2,10) process(2,2,6) ... dl(start(12),9)
Optimization: 2
OPTIMUM FOUND

Time          : 345.868s (Solving: 345.84s ...)
Conflicts     : 937922
Constraints   : 256
```

☞ Time range may deteriorate optimization performance

Let's Run with Delay Sum Minimization!

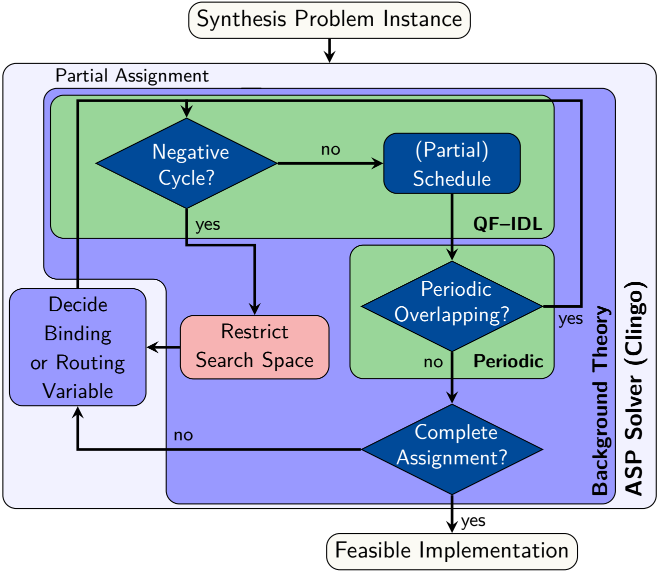
```
clingo-dl instance.lp deadline.lp differencet.lp
--stats --opt-heuristic=1 & time(0..1000)
```

```
Solving...
Answer: 1
process(1,2,10) process(2,2,6) ... dl(start(12),25)
...
Answer: 59
process(1,2,10) process(2,2,6) ... dl(start(12),9)
Optimization: 2
OPTIMUM FOUND

Time          : 0.198s (Solving: 0.17s ...)
Conflicts     : 667
Constraints   : 256
```

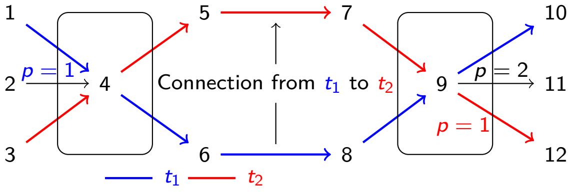
☞ Time range may deteriorate optimization performance

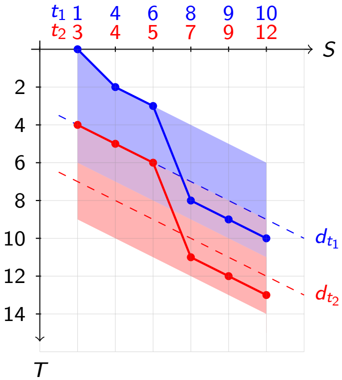
Symbolic System Synthesis [NWSH17]



Train Scheduling [AJO+19]

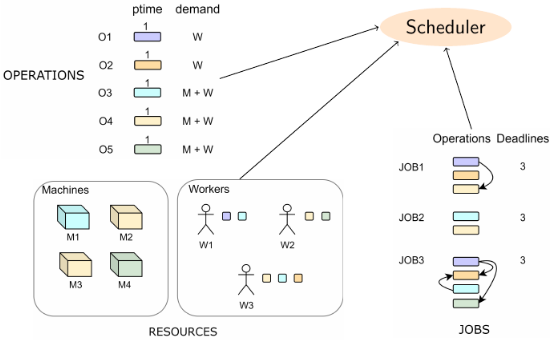
- Minimize sum of route penalties p and delays d at train stations



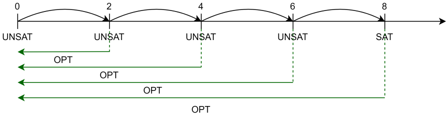


Fault Analysis Lab Scheduling [FSE21]

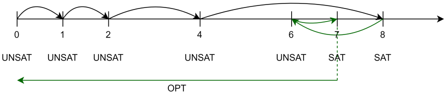
- Minimize sum of jobs' tardiness w.r.t. feasible maximum tardiness



- Linear search scheme:



- Exponential search scheme:



Conclusion

- ASP modulo DL supplies compact representation of quantities
- Particularly useful to express time, delays and non-overlapping

- Optimization by `clingo-dl` limited to single DL variable value
- Summation and/or lexicographic optimization need
  - Dedicated propagators [NWSH18] or
  - Reification in plain/multi-shot ASP [AJO+19, FSE21]

☞ `clingo-dl` is great for solving simplistic ASP modulo DL decision and optimization problems

☞ Richer optimization functionalities on DL variables' values would be the cherry on the cake

References I

[ADMR20] M. Alviano, C. Dodaro, J. Marques-Silva, and F. Ricca.
Optimum stable model search: Algorithms and implementation.
Journal of Logic and Computation, 30(4):863–897, 2020.

[AJO+19] D. Abels, J. Jordi, M. Ostrowski, T. Schaub, A. Toletti, and P. Wanko.
Train scheduling with hybrid ASP.
In M. Balduccini, Y. Lierler, and S. Woltran, editors, *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*, volume 11481 of *Lecture Notes in Artificial Intelligence*, pages 3–17. Springer-Verlag, 2019.

[BEP+14] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz.
Handbook on Scheduling: From Theory to Applications.
Springer-Verlag, 2014.

[BGJ+16] J. Bomanson, M. Gebser, T. Janhunen, B. Kaufmann, and T. Schaub.
Answer set programming modulo acyclicity.
Fundamenta Informaticae, 147(1):63–91, 2016.

[BSST09] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli.
Satisfiability modulo theories.
In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

[CB94] J. Crawford and A. Baker.
Experimental results on the application of satisfiability algorithms to scheduling problems.
In B. Hayes-Roth and R. Korf, editors, *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097. AAAI Press, 1994.

References II

[CDRS20] B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller.
Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators.
In C. Bessiere, editor, *Proceedings of the Twenty-ninth International Joint Conference on Artificial Intelligence (IJCAI'20)*, pages 1688–1694. ijcai.org, 2020.

[FSE21] G. Francescutto, K. Schekotihin, and M. El-Kholany.
Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid ASP.
In W. Faber, G. Friedrich, M. Gebser, and M. Morak, editors, *Proceedings of the Seventeenth European Conference on Logics in Artificial Intelligence (JELIA'21)*, volume 12678 of *Lecture Notes in Computer Science*, pages 313–328. Springer-Verlag, 2021.

[GKS12] M. Gebser, B. Kaufmann, and T. Schaub.
Conflict-driven answer set solving: From theory to practice.
Artificial Intelligence, 187-188:52–89, 2012.

[JKO+17] T. Janhunen, R. Kaminski, M. Ostrowski, T. Schaub, S. Schellhorn, and P. Wanko.
Clingo goes linear constraints over reals and integers.
Theory and Practice of Logic Programming, 17(5-6):872–888, 2017.

[KS23] R. Kaminski and T. Schaub.
On the foundations of grounding in answer set programming.
Theory and Practice of Logic Programming, 23(6):1138–1197, 2023.

[Lie23] Y. Lierler.
Constraint answer set programming: Integrational and translational (or SMT-based) approaches.
Theory and Practice of Logic Programming, 23(1):195–225, 2023.

References III

[NWSH17] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt.
Enhancing symbolic system synthesis through ASPmT with partial assignment evaluation.
In D. Atienza and G. Di Natale, editors, *Proceedings of the Twentieth Conference on Design, Automation and Test in Europe (DATE'17)*, pages 306–309. IEEE Computer Society Press, 2017.

[NWSH18] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt.
Exact multi-objective design space exploration using ASPmT.
In J. Madsen and A. Coskun, editors, *Proceedings of the Twenty-first Conference on Design, Automation and Test in Europe (DATE'18)*, pages 257–260. IEEE Computer Society Press, 2018.

[War98] J. Warners.
A linear-time transformation of linear inequalities into conjunctive normal form.
Information Processing Letters, 68(2):63–69, 1998.

[XGP+19] J. Xie, L. Gao, K. Peng, X. Li, and H. Li.
Review on flexible job shop scheduling.
IET Collaborative Intelligent Manufacturing, 1(3):67–77, 2019.