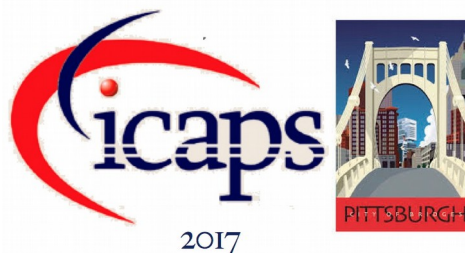




Introduction to CP Optimizer for Scheduling

Philippe Laborie
IBM, IBM Analytics
laborie@fr.ibm.com



Preamble: CP Optimizer?

- Historically developed since 2007 by ILOG, now IBM
- Descendant of ILOG Solver/Scheduler (1992-2007)
- Our team has 20+ years of experience in designing combinatorial optimization tools for **real-life industrial problems**, and particularly **scheduling** problems
- Industrial scheduling problems:
 - Large (e.g. 1.000.000 tasks)
 - Complex constraints (activities, resources)
 - Complex cost functions (not just makespan!)
 - Industrial context : tight deadlines, ill-defined problems, moving targets, bad data, need for performance, ...
- #1 objective of CP Optimizer: **lower the barrier to entry for efficiently solving industrial scheduling problems**

Preamble: Introductory problem

- A simplified semiconductor manufacturing scheduling problem



S. Knopp et al. Modeling Maximum Time Lags in Complex Job-Shops with Batching in Semiconductor Manufacturing. PMS 2016.

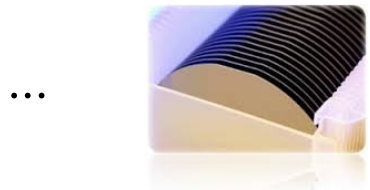
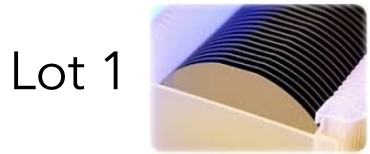
Preamble: Introductory problem

- Flexible job-shop scheduling with parallel batching

Preamble: Introductory problem

- Flexible job-shop scheduling with parallel batching

Wafer lots
(size, priority,
release date,
due date)

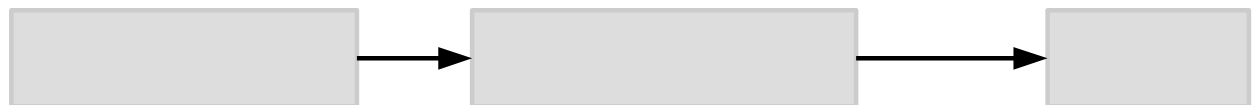
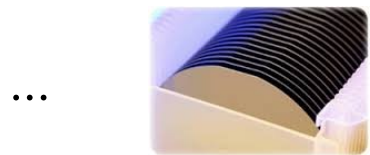
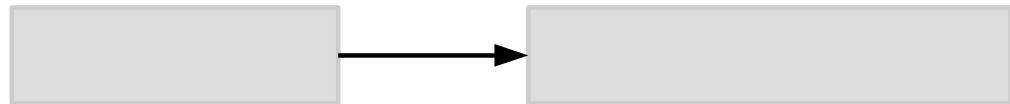
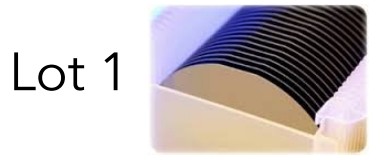


Preamble: Introductory problem

- Flexible job-shop scheduling with parallel batching

Wafer lots
(size, priority,
release date,
due date)

For each lot we are given a sequence of processing
steps

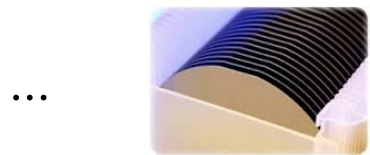
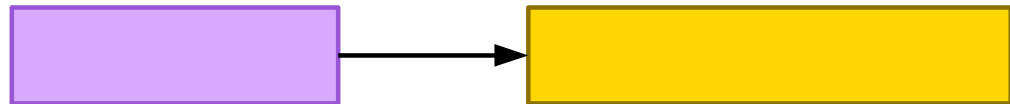
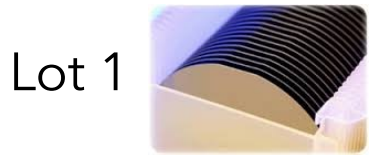


Preamble: Introductory problem

- Flexible job-shop scheduling with parallel batching

Wafer lots
(size, priority,
release date,
due date)

Each step has to be scheduled on a machine
Possible machines for a given step depend on its
family (here, its color)



Preamble: Introductory problem

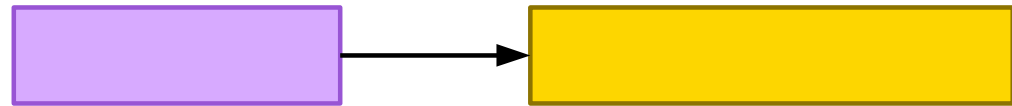
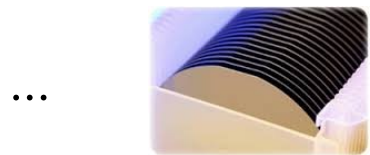
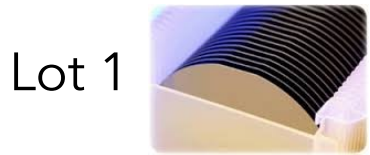
- Flexible job-shop scheduling with parallel batching

Wafer lots
(size, priority,
release date,
due date)

A machine specifies the step families it can process
and a family-dependent processing time

Example: M1 can process  and 

M2 can process  and 

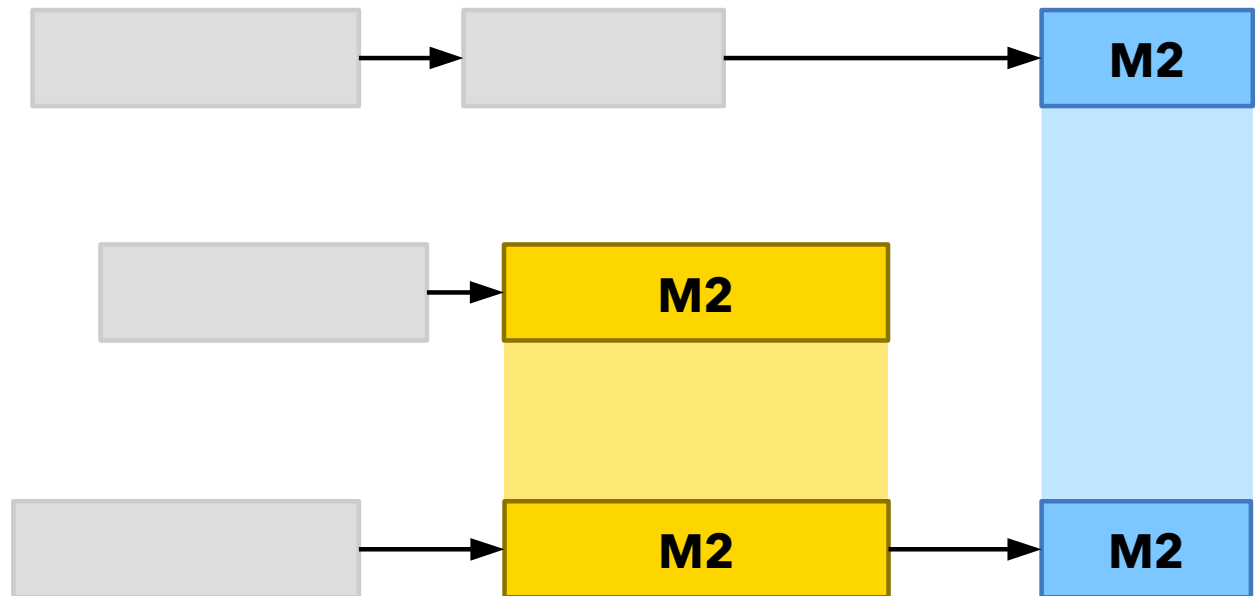


Preamble: Introductory problem

- Flexible job-shop scheduling with parallel batching

Wafer lots
(size, priority,
release date,
due date)

Steps of the same family can be processed together
on the same machine (**batch**)
Batched steps **start** and **end** at the same time
Batching **capacity**: max. number of wafers

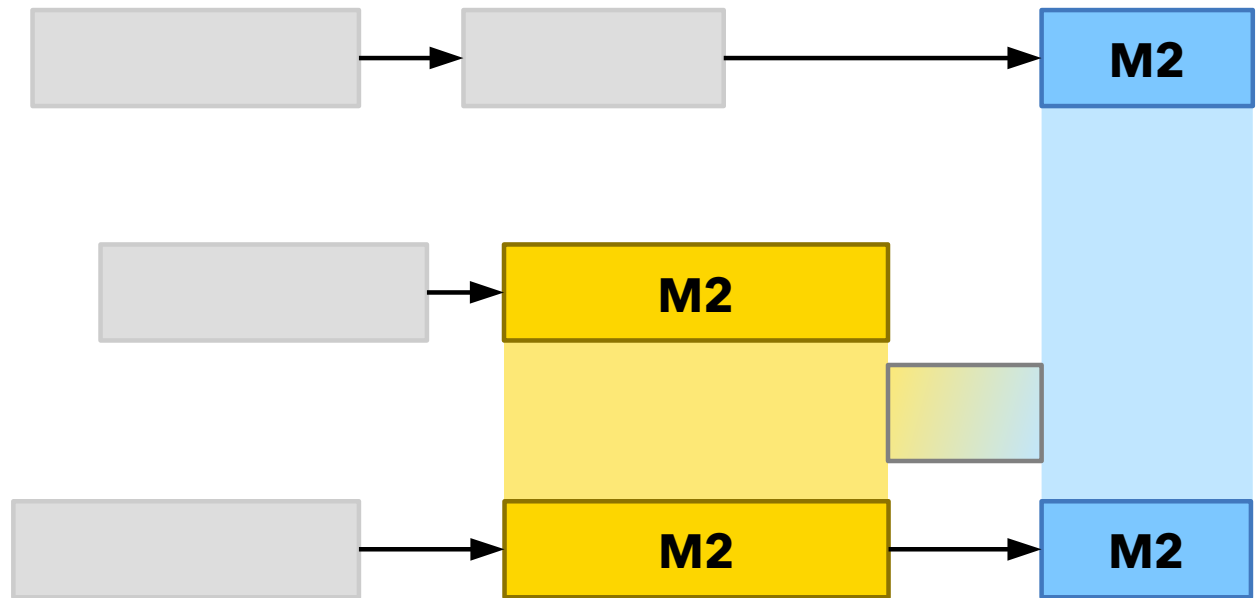


Preamble: Introductory problem

- Flexible job-shop scheduling with parallel batching

Wafer lots
(size, priority,
release date,
due date)

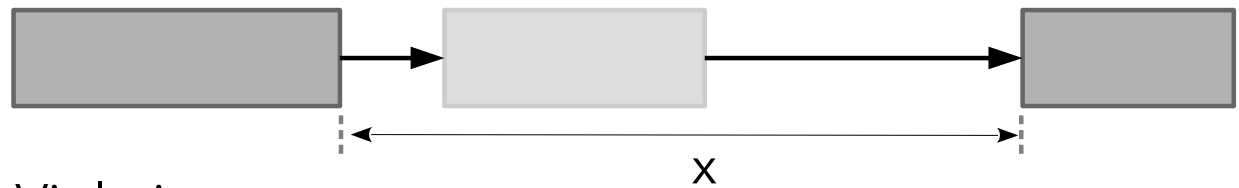
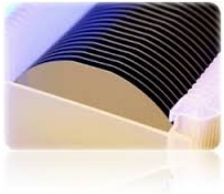
Sequence-dependent setup times on machines
Waiting time, e.g. required by temperature changes
Duration depends on family of adjacent steps



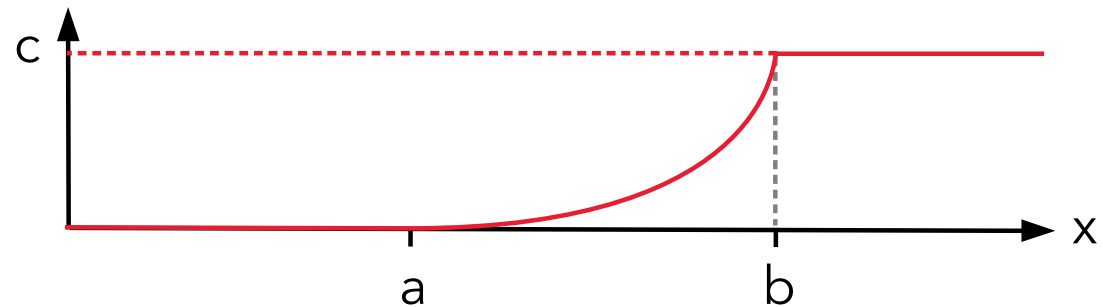
Preamble: Introductory problem

- Flexible job-shop scheduling with parallel batching

Physical and chemical properties imposes some **maximum time lags** between some pairs of lot steps
This is a soft constraint with a violation cost



Violation cost



$$V = \min(c, c \cdot \max(0, x-a)^2 / (b-a)^2)$$

Preamble: Introductory problem

- Lexicographical objective function:
 - Criterion 1: minimize total violation of maximum time lags
 - Criterion 2: minimize weighted tardiness cost of lots
- Problem size:
 - Up to 1000 lots
 - Up to 5000 lot-steps
 - Up to 150 machines
 - Up to 10 candidate machines per lot-step
 - Time unit is 1mn, schedule horizon is 48h (2880 units)

Preamble: Introductory problem

- Lexicographical objective function:
 - Criterion 1: minimize total violation of maximum time lags
 - Criterion 2: minimize weighted tardiness cost of lots
- Problem size:
 - Up to 1000 lots
 - Up to 5000 lot-steps
 - Up to 150 machines
 - Up to 10 candidate machines per lot-step
 - Time unit is 1mn, schedule horizon is 48h (2880 units)
- Try to think how you would solve this problem using your favorite tools/techniques (MILP, meta-heuristics, CP, ...)



Preamble: Introductory problem

- Lexicographical objective function:
 - Criterion 1: minimize total violation of maximum time lags
 - Criterion 2: minimize weighted tardiness cost of lots
- Problem size:
 - Up to 1000 lots
 - Up to 5000 lot-steps
 - Up to 150 machines
 - Up to 10 candidate machines per lot-step
 - Time unit is 1mn, schedule horizon is 48h (2880 units)
- Try to think how you would solve this problem using your favorite tools/techniques (MILP, meta-heuristics, CP, ...)
- After this tutorial, you will be able to solve it with a 50 lines long model achieving excellent performance

Overview

- Introduction
- Modeling
- Examples
- Performance
- Solving
- Tools
- Under the hood
- Conclusion

- Why this tutorial ?
 - CP Optimizer is not known enough in the scheduling community:
 - By people using Mathematical Programming (MILP)
 - By people using Meta-heuristics
 - By people using CP
 - By people working in AI Planning
 - Sometimes there is a misconception of what it is

Introduction

- Why this tutorial ?
 - CP Optimizer is not known enough in the scheduling community:
 - By people using Mathematical Programming (MILP)
 - By people using Meta-heuristics
 - By people using CP
 - By people working in AI Planning
 - Sometimes there is a misconception of what it is
- The messages I will try to convey:
 - CP Optimizer is **easy to try**
 - CP Optimizer is **easy to learn**
 - CP Optimizer is **easy to use**
 - CP Optimizer is **powerful** for solving scheduling problems
 - CP Optimizer is **free** for students, teachers and researchers
 - CP Optimizer is **fun** !

Introduction

- CP Optimizer comes together with CPLEX and OPL in IBM ILOG CPLEX Optimization Studio
- It is **free** for students, teachers and researchers

Researchers,
teachers,
university staff



http://ibm.biz/COS_Faculty

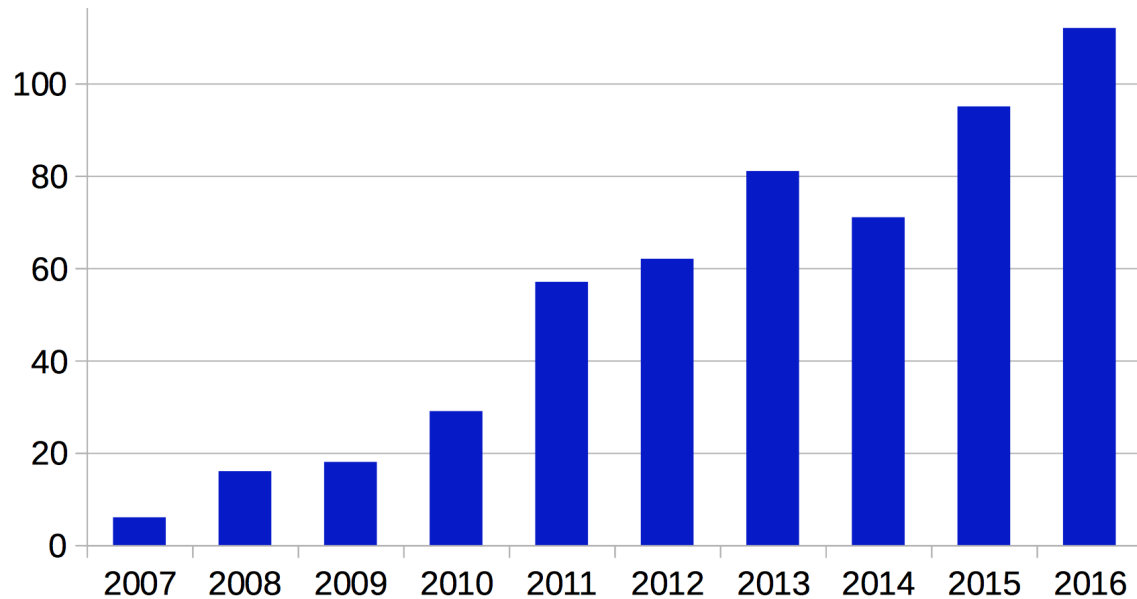
Students



http://ibm.biz/COS_Student

Introduction

- Why this tutorial ?
 - CP Optimizer is not known enough in the scheduling community (even if it is getting more and more attention)



Number of citations on Google Scholar for "CP Optimizer" per year

Different approaches for solving scheduling problems

- Mathematical Programming (MILP)
 - Advantages
 - Model & run: no need to worry (too much) about how the model is solved
 - Exact algorithm (optimality proof, gap)
 - Pain-points
 - Modeling scheduling problems is difficult
 - Maintaining/extending these complex models is difficult
 - All classical formulations lead to very large models (time-indexed, disjunctive, event-based) that often have loose LP relaxation and do not scale well

Typical real-world scheduling problems have 1000s of activities, not 10s or 100s !

Different approaches for solving scheduling problems

- User-defined Meta-heuristics (LS, GA, GRASP, SWO, ...)
 - Advantages
 - May produce excellent quality solutions (depending on how smart you are!)
 - Pain-points
 - No clear separation between model and resolution
 - Implementing/maintaining/extending the code is long, difficult and risky
 - Difficulty to handle complex scheduling constraints
 - No quality guarantee (optimality proof, gap)

In an industrial context, you need a fast assessment of the different approaches. You cannot always afford the time and risks of a PhD working on your specific problem!

Different approaches for solving scheduling problems

- Classical Constraint Programming (integer variables, global constraints like 'disjunctive' or 'cumulative')
 - Advantages
 - Modeling is easier than MILP for academic problems
 - Separation of model and resolution
 - Exact algorithms (optimality proof)
 - Pain-points
 - Some features may be hard to model (optional activities, multi-mode, calendars, batches, setup times, ...)
 - No efficient out of the box generic resolution algorithms for scheduling problems (especially for objective functions other than 'makespan', for large problems, ...)

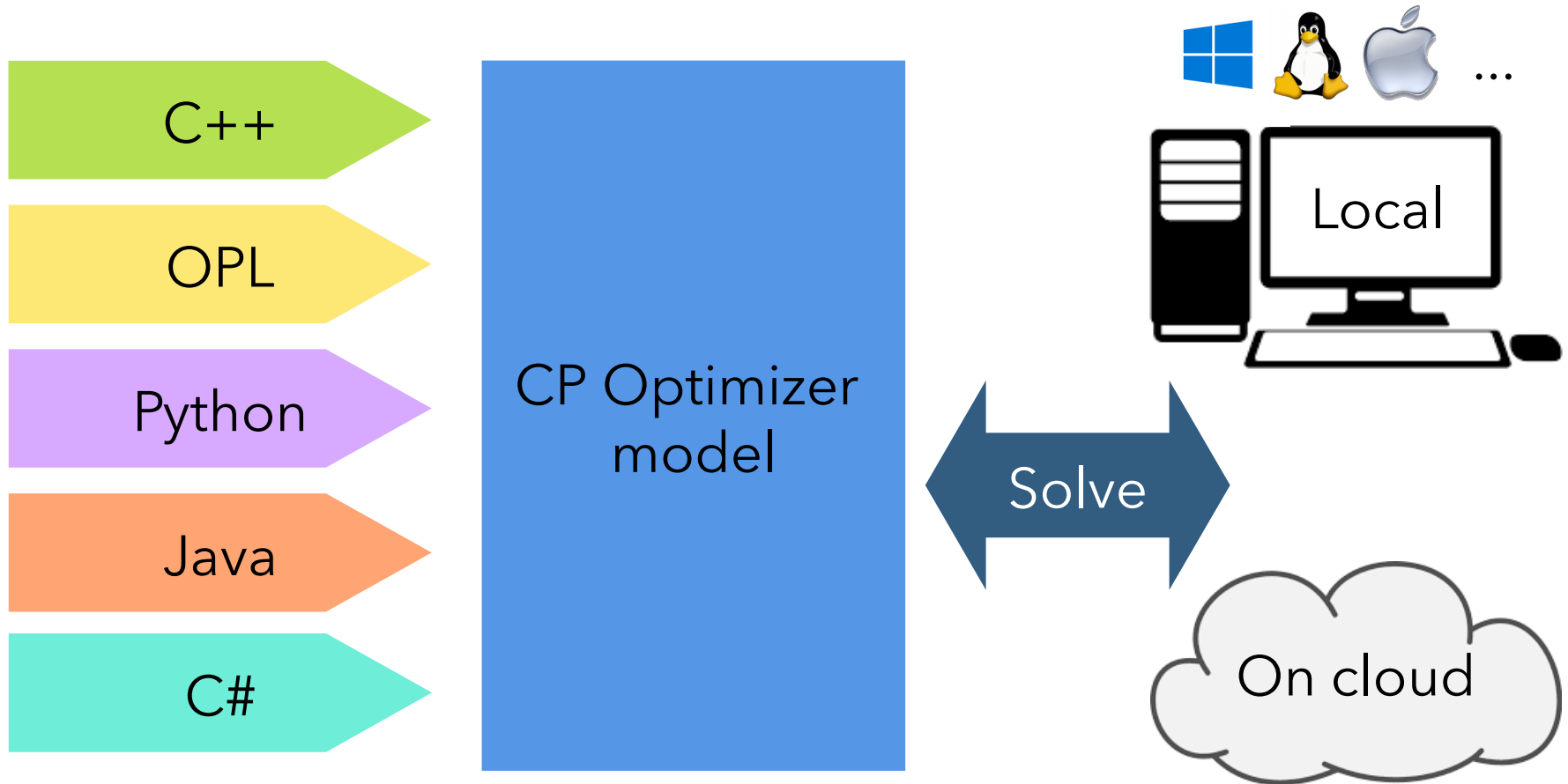
Real-world scheduling problems are complex, large and involve objective functions with weak propagation (sum)

The CP Optimizer approach

- Model & run
 - Declarative mathematical model
 - No need to worry about the resolution
- Introduction of adequate mathematical concepts for scheduling problems (intervals, functions)
 - Modeling is easy
 - Modeling is fast
 - Models are compact and maintainable
- Good out of the box performance for real world problems
- Exact algorithm using hybrid methods

Introduction

The CP Optimizer approach



Combinatorial optimization framework (MP, CP, ...)

- **Constants:** parameters defining an instance of problem
 - Examples: $A=0.02$
- **Variables:** x with domain D ($x \in D$)
 - Examples: $x \in [0,10]$, $y \in \{0,1\}$
- **Expressions:** arithmetical combination of variables
 - Examples: $x-y$, $\sum A_i x_i$, $x^{(-A*y)}$, $\max(x,y)$, $x \bmod y$, $\log(1+|x|)$
- **Constraints:** specify illegal combinations of variables value
 - Examples: $x \leq y$, $x+3.7*y \geq 2.1$, $\text{AllDifferent}(x,y,z)$
- **Objective function:** minimize or maximize a given numerical expression



Claim:

- Both MP and classical CP combinatorial optimization frameworks are not using the right abstractions to model scheduling problems
- Numerical decision variables alone (integer, floating point) make it hard to capture the essence and the structure of scheduling problems ... (even with a catalogue of more than 400 global constraints in CP)

Modeling

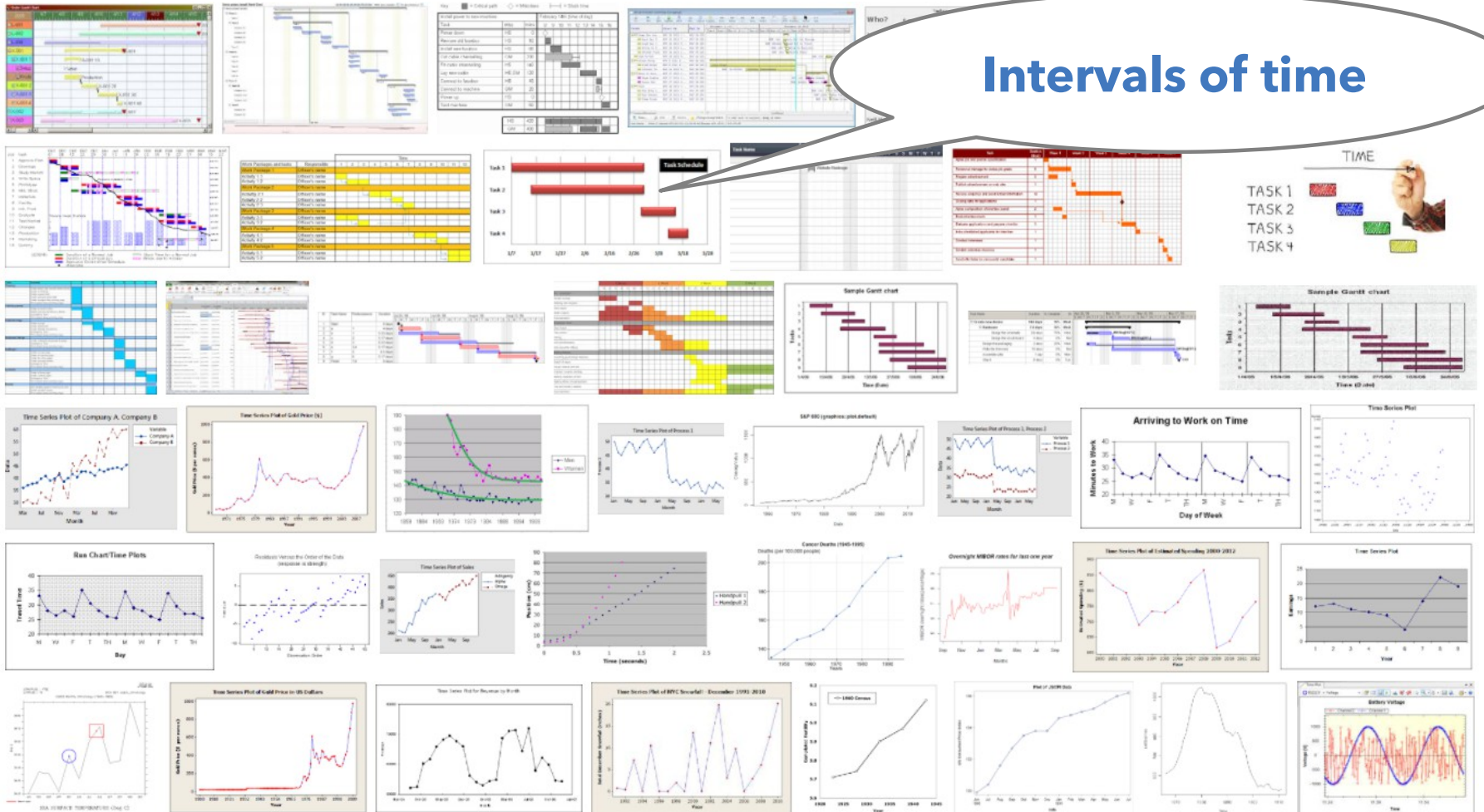
- Scheduling is about **time** !



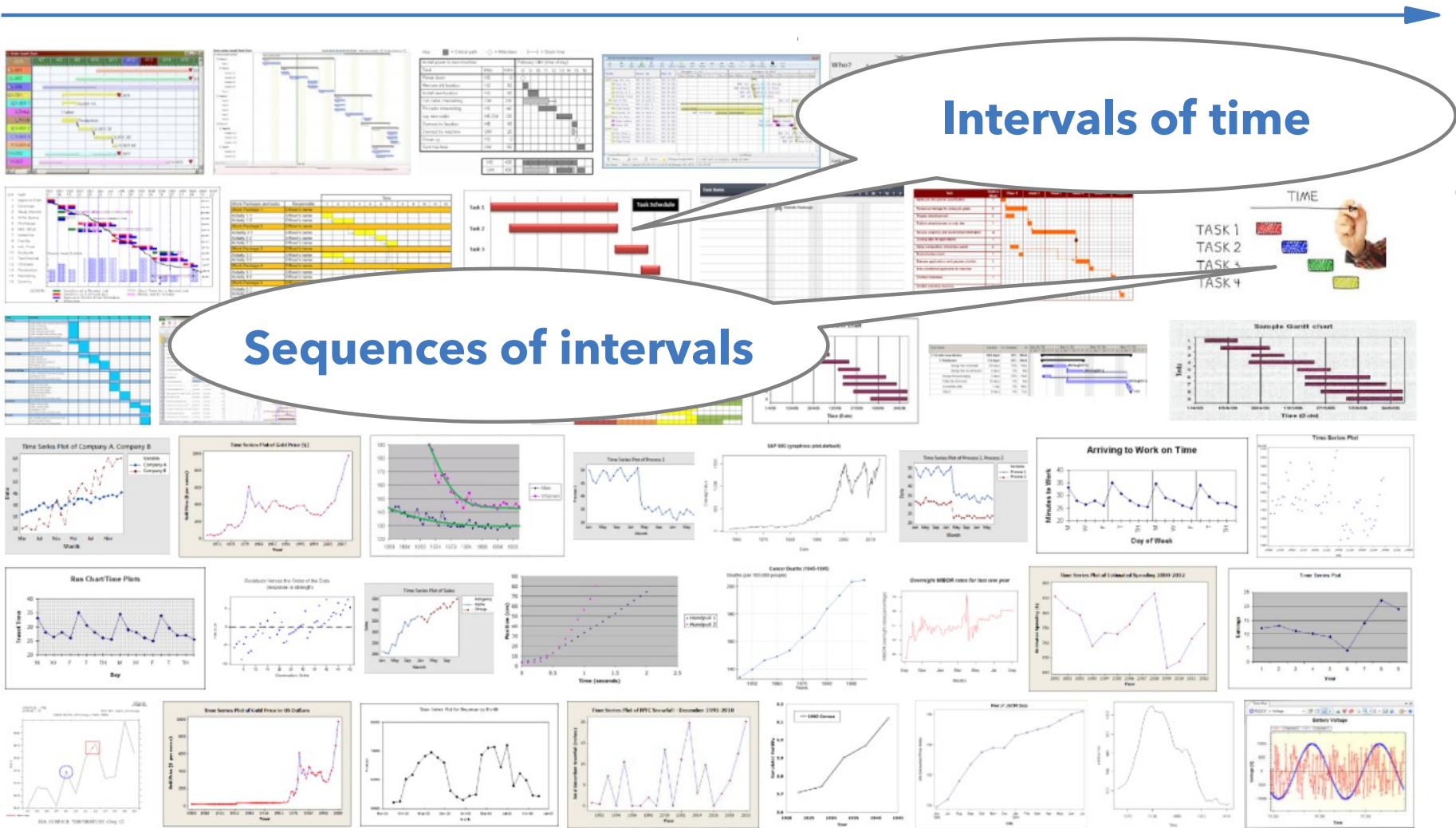
Modeling

- Scheduling is about **time** !

Intervals of time

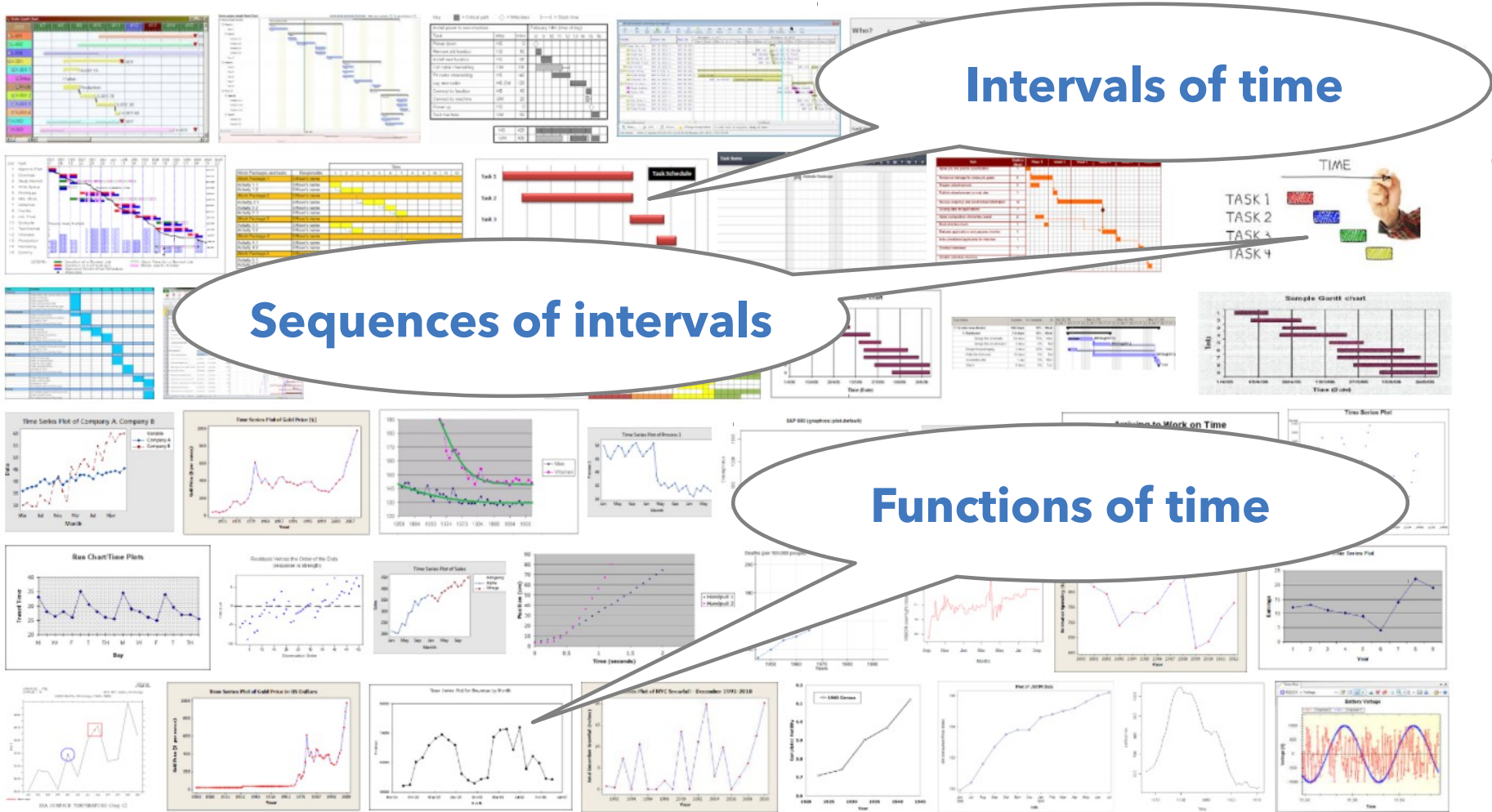


- Scheduling is about **time** !



Modeling

- Scheduling is about **time** !



- More details about the motivations and formal description of the modeling concepts that follow are available in:
- P. Laborie, J. Rogerie. *Reasoning with Conditional Time-Intervals. Proc. FLAIRS-2008, p555-560.*
- P. Laborie, J. Rogerie, P. Shaw, P. Vilím. *Reasoning with Conditional Time-Intervals. Part II: An Algebraical Model for Resources. Proc. FLAIRS-2009, p201-206.*

- CP Optimizer introduces a few new types of:

Constant



Variable



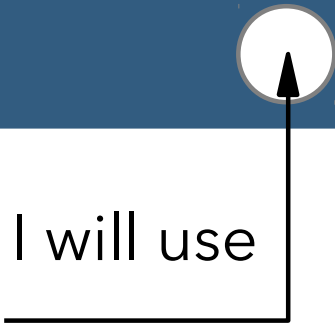
Expression



Constraint



- ... resulting in a new combinatorial optimization framework for modeling scheduling problems

- In this section presenting the modeling concepts, I will use the following categorization that will be put here 

Constant



Variable



Expression

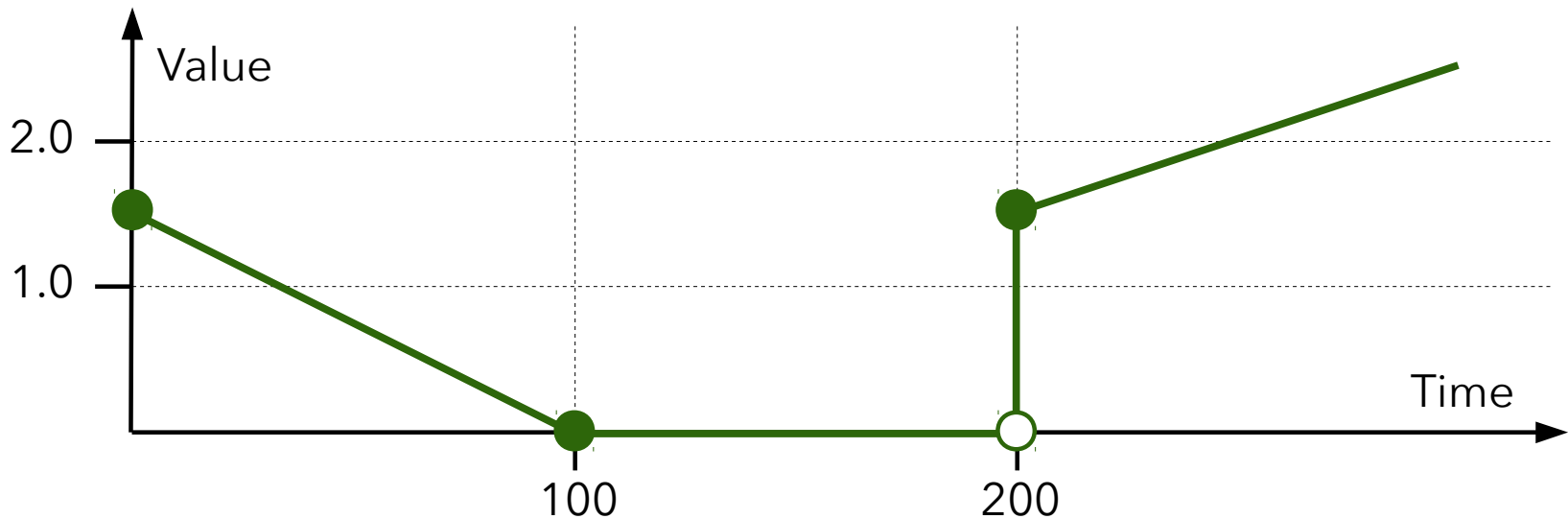


Constraint



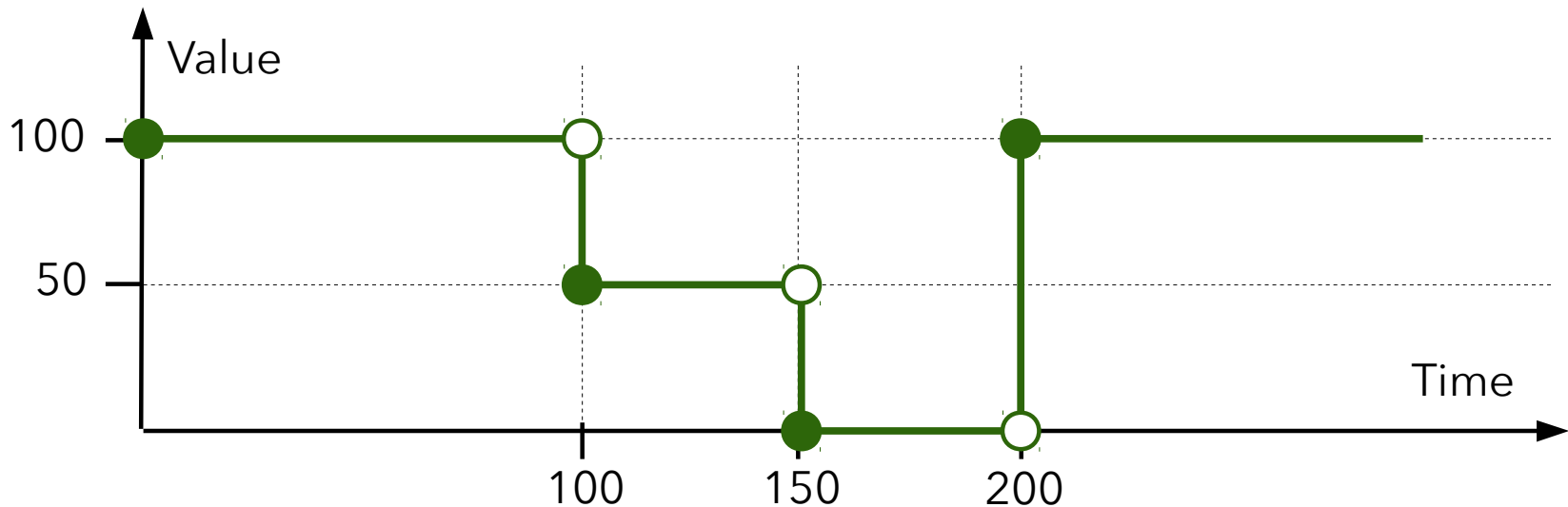
Piecewise-Linear functions

- What for?
 - Modeling earliness/tardiness costs, temporal preference, ...
- Example:
 - pwlFunction $F =$
piecewise $\{-0.015 \rightarrow 100; 0 \rightarrow 200; 1.5 \rightarrow 200; 0.01\} (100, 0);$



Stepwise functions

- What for?
 - Modeling resource calendars (breaks, intensity)
- Example:
 - `stepFunction F = stepwise {100->100; 50->150; 0->200; 100};`



Transition distance matrices

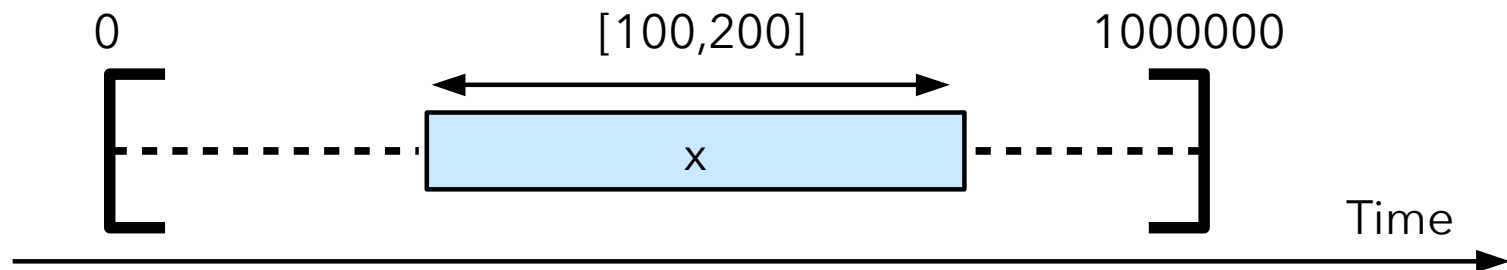
- What for?
 - Modeling transition times, travel times, etc.
- Example:

```
tuple Dist { int source; int target; int distance; }  
{Dist} DistMatrix = { <0,0,0>, <0,1,30>, <0,2,45>, ... }
```

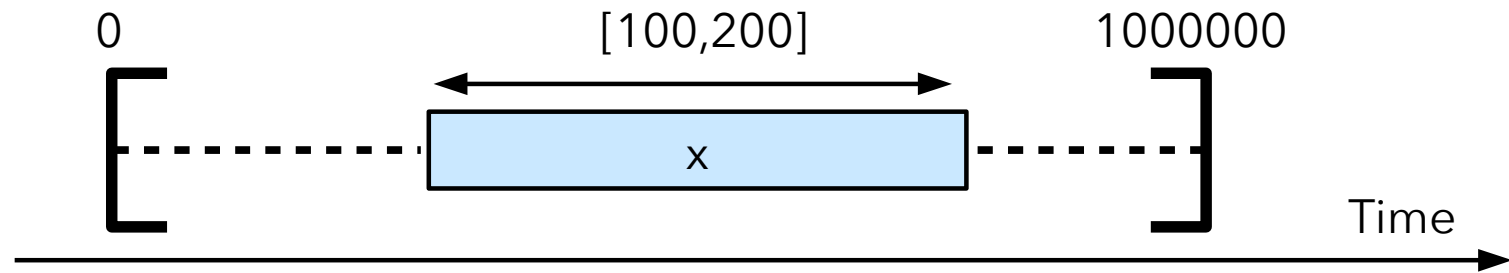
	0	1	2
0	0	30	45
1	25	5	60
2	40	55	0

Interval variables

- What for?
 - Modeling an interval of time during which a particular property holds (an activity executes, a resource is idle, a tank must remain empty, ...)
- Example:
 - `dvar interval x in 0..1000000 size 100..200;`



```
dvar interval x in 0..1000000 size 100..200;
```



- Properties:
 - The value of an interval variable is an integer interval [start,end)
 - Domain of possible values: [0,100), [1,101), [2,102),... [999900,1000000), [0,101),[1,102),...
 - Domain of interval variables is represented **compactly** inside CP Optimizer (a few bounds: smin, smax, emin, emax, szmin, szmax)

Optional interval variable

- Interval variables can be defined as being **optional** that is, it is part of the decisions of the problem to decide whether the interval will be present or absent in the solution
- What for?
 - Modeling optional activities, alternative execution modes for activities, and ... most of the discrete decisions in a schedule
- Example:
 - `dvar interval x optional in 0..1000000 size in 100..200`
- An optional interval variable has an additional possible value in its domain (absence value)

Optional interval variable

- An **optional** interval variable has an additional possible value in its domain (absence value)
- Domain of values for an optional interval variable x :

$$\text{Dom}(x) \subseteq \{\perp\} \cup \{[s,e) \mid s,e \in \mathbb{Z}, s \leq e\}$$

Absent interval



Interval of integers
(when interval is **present**)



- Constraints and expressions on interval variables specify how they handle the case of absent intervals (in general it is very intuitive)

Optional interval variable

- An **optional** interval variable has an additional possible value in its domain (absence value)
- Domain of values for an optional interval variable x :

$$\text{Dom}(x) \subseteq \{\perp\} \cup \{[s,e) \mid s,e \in \mathbb{Z}, s \leq e\}$$

Absent interval



Interval of integers
(when interval is **present**)



- Optionality is a fundamental and powerful notion that you must learn to leverage in your models (more on this later ...)

Interval variable **size**, **length** and **intensity function**

- What for ?
 - Modeling cases where the “intensity” of work is not the same during the whole interval and the interval requires some quantity of work to be done before completion
 - Activities that are suspended during some time periods (e.g. week-end, vacations)

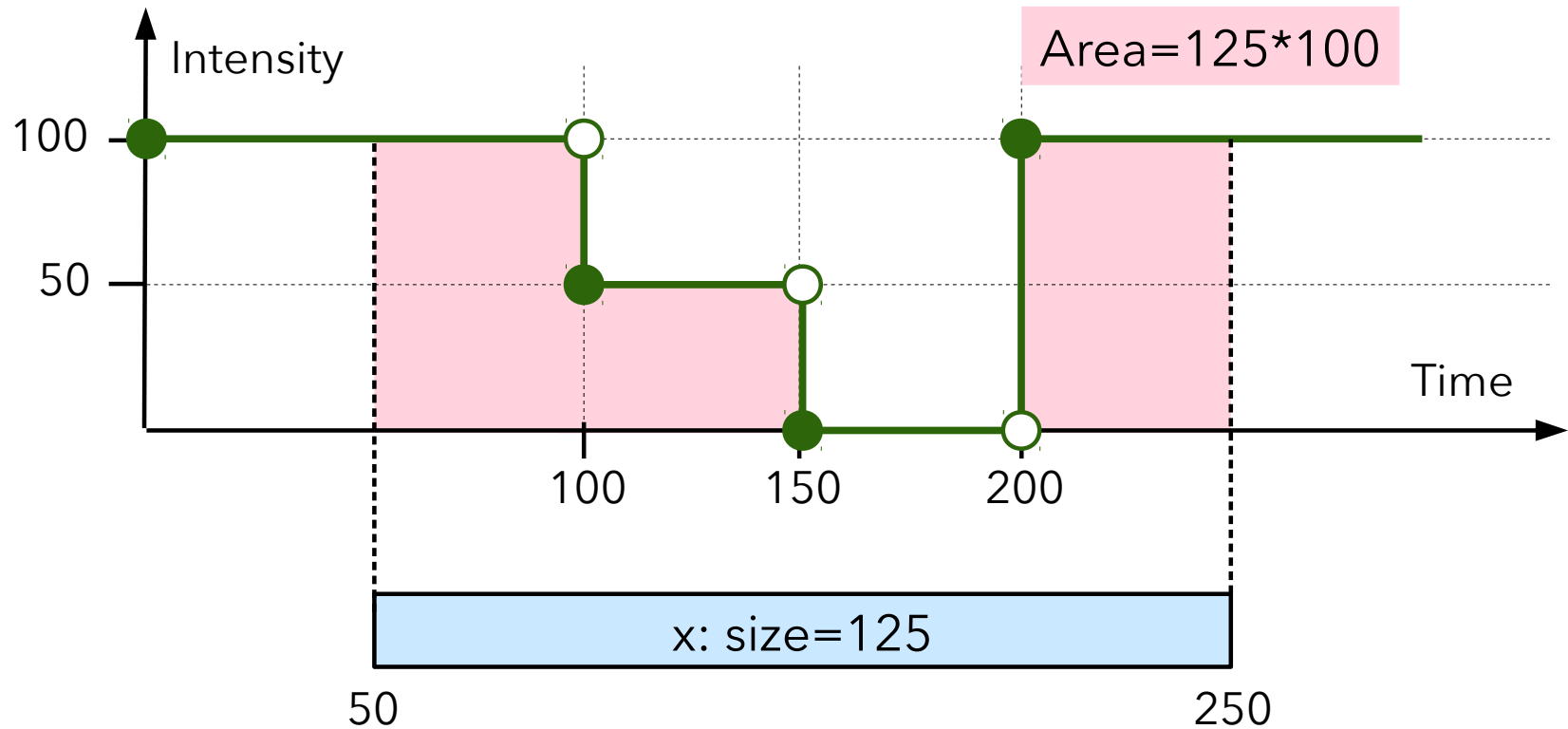
Interval variable **size**, **length** and **intensity function**

- The **length** of a (present) interval variable x is $e(x)-s(x)$
- The **size** of an interval variable is the “ideal” length of the interval variable
- By default, $\text{size}=\text{length}$ but an integer stepwise function F can be specified when creating an interval variable to change this relation (notion of intensity function)
- The **intensity function** F specifies the instantaneous ratio (in %) between size and length

$$(x \neq \perp) \Rightarrow 100 * \text{size}(x) \leq \sum_{t=s(x)}^{e(x)-1} F(t) < 100 * (\text{size}(x) + 1)$$

Interval variable **size**, **length** and **intensity function**

- Example: dvar interval x size 125 intensity F



$$(x \neq \perp) \Rightarrow 100 * \text{size}(x) \leq \sum_{t=s(x)}^{e(x)-1} F(t) < 100 * (\text{size}(x) + 1)$$

Interval variable **size**, **length** and **intensity function**

- When no intensity function is specified, it is assumed to be the constant full intensity function ($F=100\%$), so in this case $\text{size}(x)=\text{length}(x)=e(x)-s(x)$



Some unary constraints on interval variables are available to forbid an interval (if present) to start, end or overlap a set of fixed time windows

`forbidStart(x, F)`

`forbidEnd(x, F)`

`forbidExtent(x, F)`

- If interval x is present, it cannot start (resp. end, overlap) at a time t such that $F(t)=0$



- F can be the intensity function of the interval variable
- Suppose $F(t)=0$ on week-ends, 100 otherwise

```
// Activity x needs 10 work-days and  
// is suspended during week ends  
dvar interval x size 10 intensity F
```

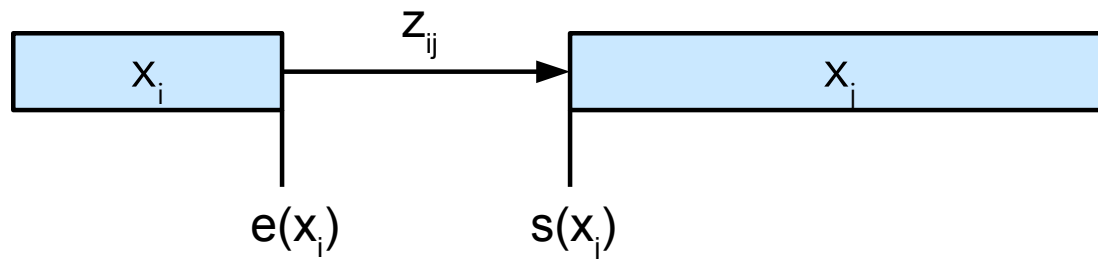
```
// Activity x cannot start during week-ends  
forbidStart(x,F)
```

```
// Activity x cannot end during week-ends  
forbidEnd(x,F)
```

- Same expressivity as Simple Temporal Networks (Dechter, Meiri, and Pearl, 1991) (STN)
- **But** temporal constraint definition $t_i + z_{ij} \leq t_j$ is reified by optional statuses
- Example:

`endBeforeStart(xi, xj, zij)` means:

$$(x_i \neq \perp) \wedge (x_j \neq \perp) \Rightarrow e(x_i) + z_{ij} \leq s(x_j)$$



Constraint name	Semantics $(x_i \neq \perp) \wedge (x_j \neq \perp) \Rightarrow$	Pictogram
endBeforeStart	$e(x_i) + z_{ij} \leq s(x_j)$	
startBeforeStart	$s(x_i) + z_{ij} \leq s(x_j)$	
endBeforeEnd	$e(x_i) + z_{ij} \leq e(x_j)$	
startBeforeEnd	$s(x_i) + z_{ij} \leq e(x_j)$	
endAtStart	$e(x_i) + z_{ij} = s(x_j)$	
startAtStart	$s(x_i) + z_{ij} = s(x_j)$	
endAtEnd	$e(x_i) + z_{ij} = e(x_j)$	
startAtEnd	$s(x_i) + z_{ij} = e(x_j)$	



- Precedence constraints are aggregated in a Precedence Network (more on this in section “Under the hood”)
 - Dedicated constraint propagation algorithms
 - Exploitation of the structure by the automatic search

- Unary presence constraint

presenceOf(x) means: $(x \neq \perp)$

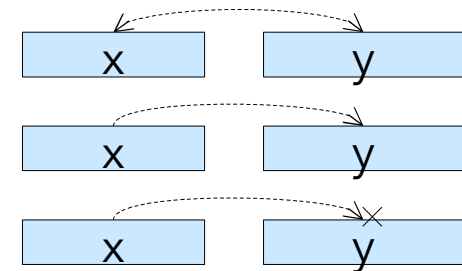
- Logical binary constraints between presence status:

Examples:

$\text{presenceOf}(x) == \text{presenceOf}(y)$

$\text{presenceOf}(x) \Rightarrow \text{presenceOf}(y)$

$\text{presenceOf}(x) \Rightarrow \neg \text{presenceOf}(y)$



Same expressivity as 2-SAT

- Of course, other combinations are also possible

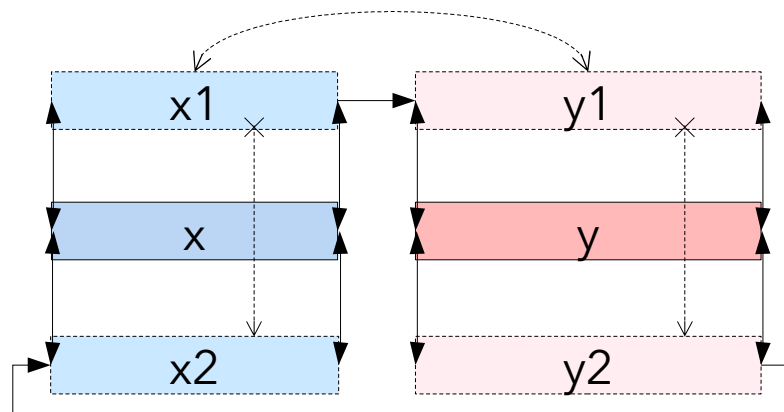
$\text{presenceOf}(x) \ \&\& \ \text{presenceOf}(y) \Rightarrow$
 $\text{presenceOf}(u) \ || \ \text{presenceOf}(v)$



- Logical binary constraints are aggregated in a Logical Network (more on this in section “Under the hood”)
 - Dedicated constraint propagation algorithms
 - Exploitation of the structure by the automatic search

Modeling: A parenthesis on Complexity

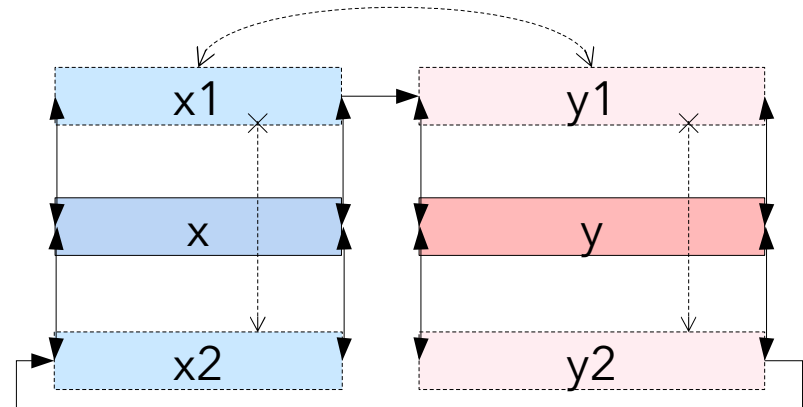
- Precedence constraints (STN) \Rightarrow Polynomial
- Logical binary constraints (2-SAT) \Rightarrow Polynomial
- Precedence + Logical binary \Rightarrow NP-Hard
- Proof is a consequence of the fact temporal disjunction between 2 interval variables x, y can be expressed with precedence and logical binary constraints only:



Modeling: A parenthesis on Complexity

- Where the first CP Optimizer model we see is one you should never write for solving a real problem ...

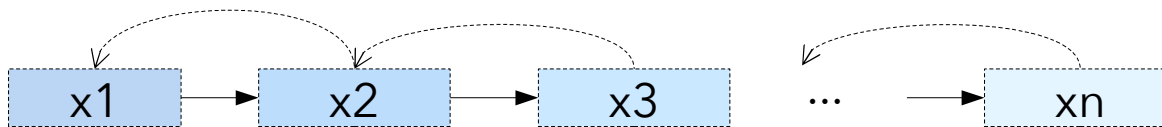
```
1  using CP;
2
3  // Decision variables
4  dvar interval x size 10;
5  dvar interval y size 20;
6  dvar interval x1 optional;
7  dvar interval x2 optional;
8  dvar interval y1 optional;
9  dvar interval y2 optional;
10
11 constraints {
12   // Precedence constraints
13   startAtStart(x, x1);
14   startAtStart(x, x2);
15   endAtEnd(x, x1);
16   endAtEnd(x, x2);
17   startAtStart(y, y1);
18   startAtStart(y, y2);
19   endAtEnd(y, y1);
20   endAtEnd(y, y2);
21   endBeforeStart(x1, y1);
22   endBeforeStart(y2, x2);
23   // Logical binary constraints
24   presenceOf(x1) == presenceOf(y1);
25   !presenceOf(x1) => presenceOf(x2);
26   !presenceOf(y1) => presenceOf(y2);
27 }
```



Modeling: A useful pattern

- This CP Optimizer pattern is much more useful ...

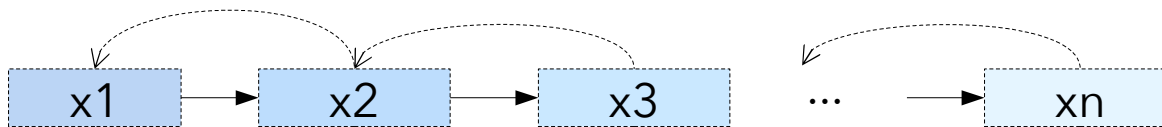
```
1 using CP;
2
3 // Decision variables
4 dvar interval x[i in 1..n] optional;
5
6 constraints {
7   forall(i in 1..n-1) {
8     // Precedence constraints
9     endBeforeStart(x[i], x[i+1]);
10    // Logical binary constraints
11    presenceOf(x[i+1]) => presenceOf(x[i]);
12  }
13 }
```



Modeling: A useful pattern

- This CP Optimizer pattern is much more useful ...

```
1 using CP;
2
3 // Decision variables
4 dvar interval x[i in 1..n] optional;
5
6 constraints {
7   forall(i in 1..n-1) {
8     // Precedence constraints
9     endBeforeStart(x[i], x[i+1]);
10    // Logical binary constraints
11    presenceOf(x[i+1]) => presenceOf(x[i]);
12  }
13 }
```



- Chain of optional interval variables:
 - Only the first ones will be present
 - Useful for modeling a chain of k intervals, k being unknown

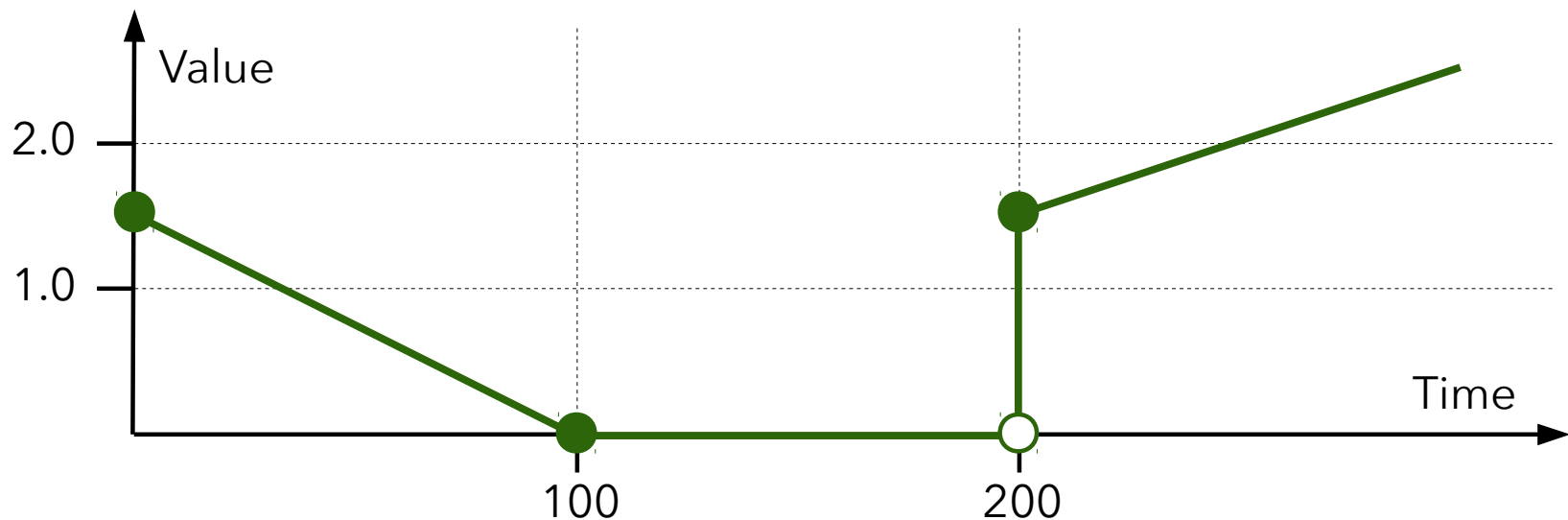
- What for ?
 - Build objective functions based on interval variables
 - Post constraints on interval variables (when there is not a more direct way)
- Integer expressions:
 - startOf(x, V) has value: V if $x = \perp$, s(x) otherwise
 - endOf(x, V) has value: V if $x = \perp$, e(x) otherwise
 - sizeOf(x, V) has value: V if $x = \perp$, size(x) otherwise
 - lengthOf(x, V) has value: V if $x = \perp$, e(x)-s(x) otherwise
- Parameter V is optional (default value is 0)
- Typical example of a makespan objective function:
minimize max(i in 1..n) endOf(x[i])

- What for ?
 - Build objective functions based on interval variables
 - Post constraints on interval variables (when there is not a more direct way)
- Numerical expressions given a piecewise linear function F :
 - $\text{startEval}(x, F, V)$ has value: V if $x = \perp$, $F(s(x))$ otherwise
 - $\text{endEval}(x, F, V)$ has value: V if $x = \perp$, $F(e(x))$ otherwise
 - $\text{sizeEval}(x, F, V)$ has value: V if $x = \perp$, $F(\text{size}(x))$ otherwise
 - $\text{lengthEval}(x, F, V)$ has value: V if $x = \perp$, $F(e(x) - s(x))$ otherwise
- Parameter V is optional (default value is 0.0)

- Typical example of a temporal preference/cost objective function:

$$\text{minimize } \sum(i \text{ in } 1..n) \text{ endEval}(x[i], F[i], V[i])$$

- $V[i]$ is the non-execution cost of activity $x[i]$
- $F[i]$ is the piecewise linear function of time t representing the cost when activity $x[i]$ finishes at t (if executed)



- Integer expressions:

$\text{overlapLength}(x, y, V)$

V

$|x \cap y|$

has value:

if $(x = \perp) \vee (y = \perp)$

otherwise

$\text{overlapLength}(x, S, E, V)$

V

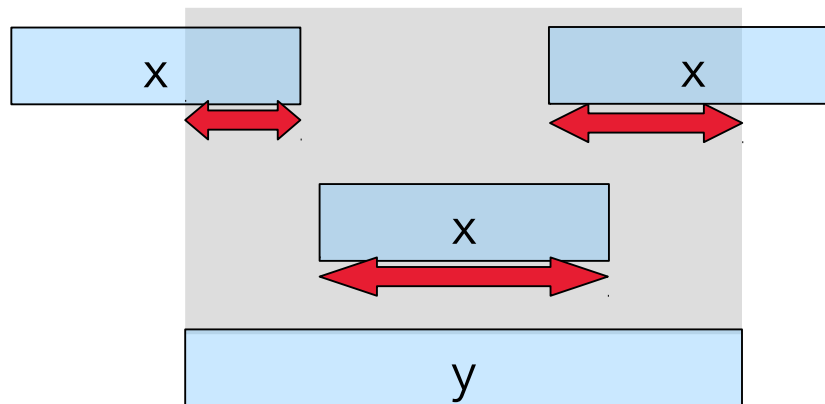
$|x \cap [S, E)|$

has value:

if $(x = \perp)$

otherwise

- Parameter V is optional (default value is 0)

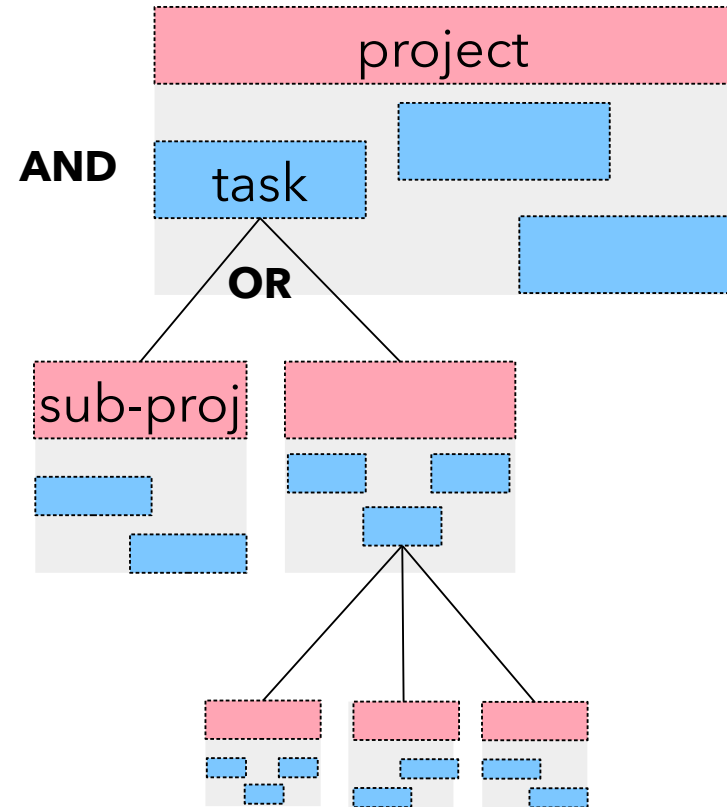
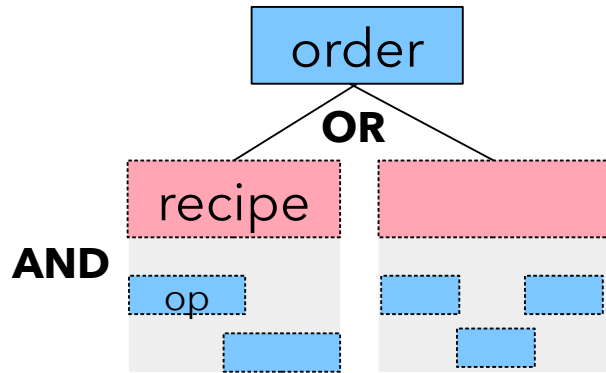


- These expressions can be mixed with other numerical expressions in arbitrarily complex expression trees:
 - General expressions:
 $x*y, k*x, x+y, x+k, x-y, \text{abs}(x), \text{min}(x,y), \text{max}(x,y), \dots$
 - Integer expressions:
 $x \text{ div } y, x \text{ mod } y,$
 - Floating point expressions:
 $\text{ceil}(x), \text{floor}(x), \text{frac}(x),$
 $x/y, \text{sqrt}(x), \text{exp}(x), \text{log}(x), \text{pow}(x,y), \dots$
 - Array expressions:
 $A[x]$ where x is an integer expression indexing an array A of integer/floating point values/expressions

- What for?
 - Modeling hierarchical problems that can naturally be described in terms of And-Or graphs
- Manufacturing use-case:
 - A production order has earliness-tardiness cost and can be executed following several **alternative** recipes. A recipe consists of a **set of** operations with temporal constraints. Some operations require setups, transportation using secondary resources, etc. Some operations (e.g. testing) or the whole production order can be left **unperformed** but this will incur additional cost.

- What for?
 - Modeling hierarchical problems that can naturally be described in terms of And-Or graphs
- Project scheduling use-case:
 - A project is **decomposed** into several tasks with temporal constraints. A task is either a leaf task or a sub-project that can be **decomposed** further on. There may be **alternative** ways to decompose a task. A task uses some skilled actors. Some tasks have due-date and tardiness cost. Tasks can be left **unperformed** or externalized but this will incur additional cost. Some tasks of different sub-project have meeting points that can be **skipped** under cost compensation.

- Hierarchical decomposition

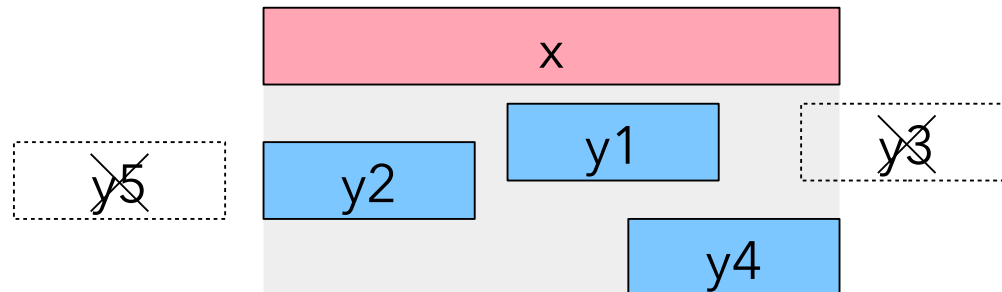


- Note some similarities with HTN planning

- Span constraint (AND node)

$\text{span}(x, [y_1, \dots, y_n])$

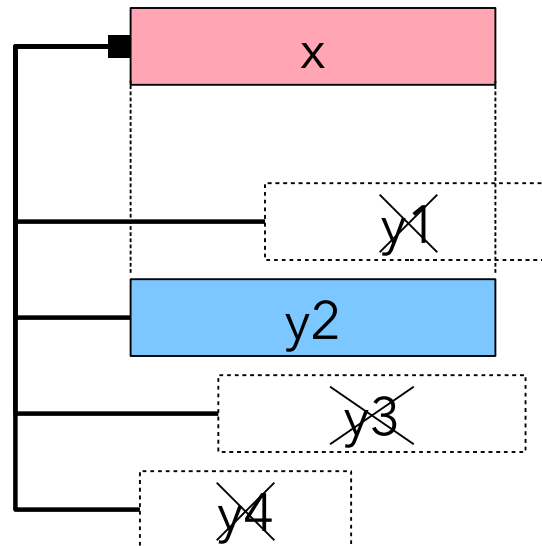
- If x is present it spans all present intervals from $\{y_1, \dots, y_n\}$ that is, at least one of y_i supports the start (resp. end) of x
- If x is absent, then all y_i are absent too



- Alternative constraint (OR node)

`alternative(x, [y1,...,yn])`

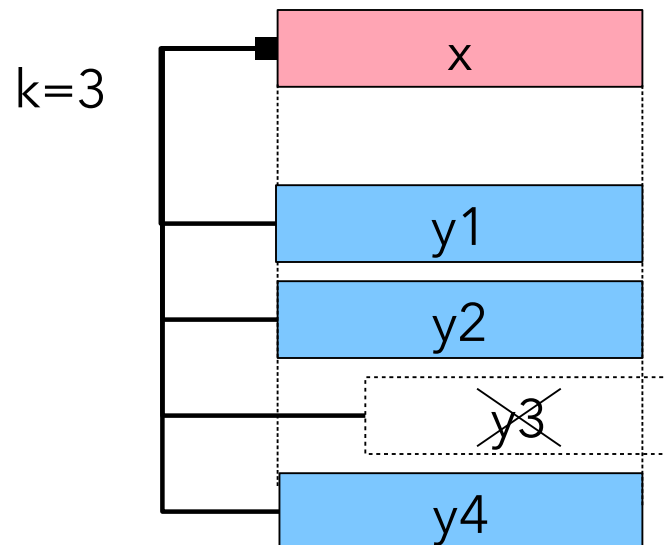
- If x is present, then exactly one of the $\{y1, \dots, yn\}$ is present and synchronized with x (same start and end value)
- If x is absent, then all y_i are absent too



- Generalized alternative constraint

$\text{alternative}(x, [y_1, \dots, y_n], k)$ k : integer expression

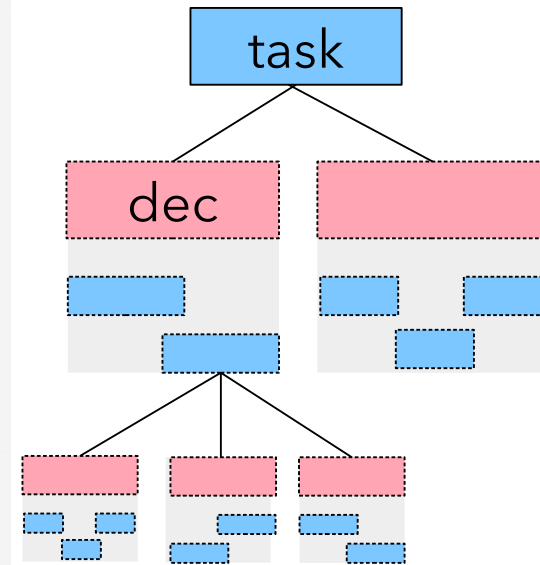
- If x is present, then exactly k of the $\{y_1, \dots, y_n\}$ are present and synchronized with x (same start and end value)
- If x is absent, then all y_i are absent too



Modeling: Structural constraints

- Work-breakdown structure with alternative tasks

```
1 using CP;
2 tuple Dec { int task; {int} subtasks; };
3 int n = ...;
4 int compulsory[1..n] = ...;
5 {Dec} Decs = ...;
6 int nbDecs[i in 1..n] = card( {d | d in Decs : d.task==i} );
7 int nbParents[i in 1..n] = card( {d | d in Decs : i in d.subtasks} );
8
9 dvar interval task[i in 1..n] optional;
10 dvar interval dec[d in Decs] optional;
11
12 constraints {
13   forall(i in 1..n) {
14     if (nbParents[i]==0 && 0<compulsory[i])
15       presenceOf(task[i]);
16     if (nbDecs[i]>0) {
17       alternative(task[i], all(d in Decs: d.task==i) dec[d]);
18       forall(d in Decs: d.task==i)
19         span(dec[d], all(j in d.subtasks) task[j]);
20     }
21   }
22   forall(d in Decs, j in d.subtasks: 0<compulsory[j])
23     presenceOf(dec[d]) => presenceOf(task[j]);
24 }
```



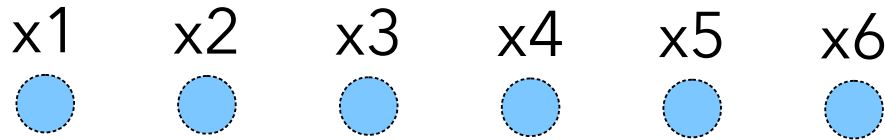
Modeling: Interval variables sequencing

- What for?
 - Modeling constraints that enforce (among other things) a total temporal ordering of a set of interval variables
- Examples:
 - A machine that can only perform one task at a time (job-shop)
 - A vehicle that can only visit one location at a time (TSP)

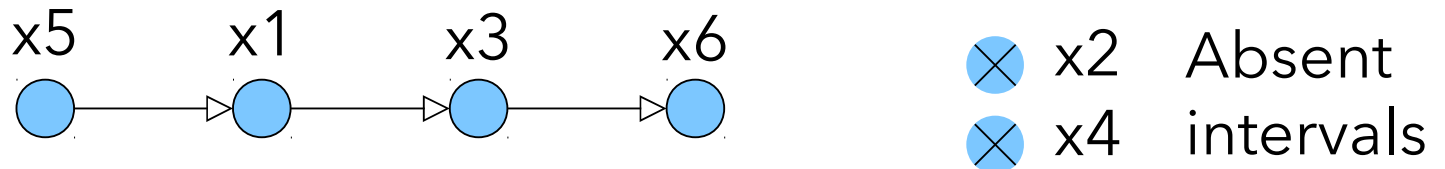
Sequence Variable

- A **sequence variable** p is defined on a set of interval variables $\{x_1, \dots, x_n\}$

```
dvar interval x[i in 1..n] ...;  
dvar sequence p in x;
```



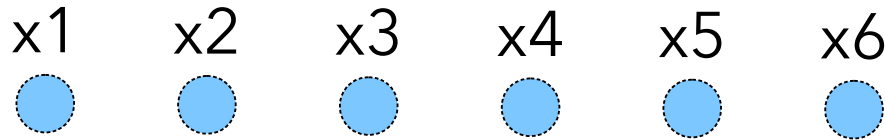
- A **value** of p is a permutation of the present intervals in x



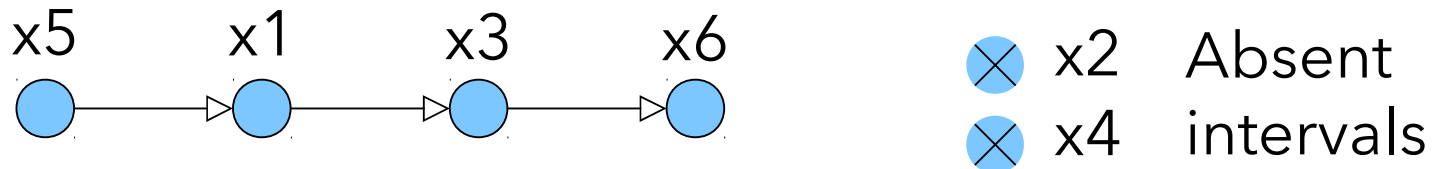
Sequence Variable

- A **sequence variable** p is defined on a set of interval variables $\{x_1, \dots, x_n\}$

```
dvar interval x[i in 1..n] ...;  
dvar sequence p in x;
```



- A **value** of p is a permutation of the present intervals in x



So far, the order in the permutation p does not imply any temporal ordering

Sequencing constraints

- **Sequencing constraints** are unary constraints on a sequence variable p
 - $\text{first}(p,x)$: if x is present it is the first one in the permutation
 - $\text{last}(p,x)$: if x is present it is the last one in the permutation
 - $\text{prev}(p,x,y)$: if both x and y are present, x appears right before y in the permutation
 - $\text{before}(p,x,y)$: if both x and y are present, x appears somewhere before y in the permutation

Sequencing constraints

```
dvar interval x[i in 1..3] optional;  
dvar sequence p in x;  
  
constraints {  
    presenceOf(x[2])  
}
```

Domain of variable p

```
{  
    (x2),  
    (x1 → x2),  
    (x2 → x1),  
    (x2 → x3),  
    (x3 → x2),  
    (x1 → x2 → x3),  
    (x1 → x3 → x2),  
    (x2 → x1 → x3),  
    (x2 → x3 → x1),  
    (x3 → x1 → x2),  
    (x3 → x2 → x1)  
}
```

Sequencing constraints

```
dvar interval x[i in 1..3] optional;  
dvar sequence p in x;
```

```
constraints {  
  presenceOf(x[2])  
  first(p, x[1]);  
}
```

Domain of variable p

```
{  
  (x2),  
  (x1 → x2),  
  (x2 → x1),  
  (x2 → x3),  
  (x3 → x2),  
  (x1 → x2 → x3),  
  (x1 → x3 → x2),  
  (x2 → x1 → x3),  
  (x2 → x3 → x1),  
  (x3 → x1 → x2),  
  (x3 → x2 → x1)  
}
```

Sequencing constraints

```
dvar interval x[i in 1..3] optional;  
dvar sequence p in x;  
  
constraints {  
  presenceOf(x[2])  
  first(p, x[1]);  
  before(p, x[2], x[3]);  
}
```

Domain of variable p

```
{  
  (x2),  
  (x1 → x2),  
  (x2 → x1),  
  (x2 → x3),  
  (x3 → x2),  
  (x1 → x2 → x3),  
  (x1 → x3 → x2),  
  (x2 → x1 → x3),  
  (x2 → x3 → x1),  
  (x3 → x1 → x2),  
  (x3 → x2 → x1)  
}
```

Sequencing constraints

```
dvar interval x[i in 1..3] optional;  
dvar sequence p in x;
```

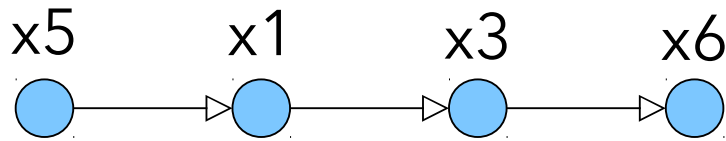
```
constraints {  
  presenceOf(x[2])  
  first(p, x[1]);  
  before(p, x[2], x[3]);  
  prev(p, x[1], x[3]);  
}
```



Domain of variable p

```
{  
  (x2),  
  (x1 → x2),  
  (x2 → x1),  
  (x2 → x3),  
  (x3 → x2),  
  (x1 → x2 → x3),  
  (x1 → x3 → x2),  
  (x2 → x1 → x3),  
  (x2 → x3 → x1),  
  (x3 → x1 → x2),  
  (x3 → x2 → x1)  
}
```


No-Overlap constraint

- Let p be a sequence variable. No-overlap constraint $\text{noOverlap}(p)$ means that p represents a chain of non overlapping interval variables



 x_2 Absent
 x_4 intervals

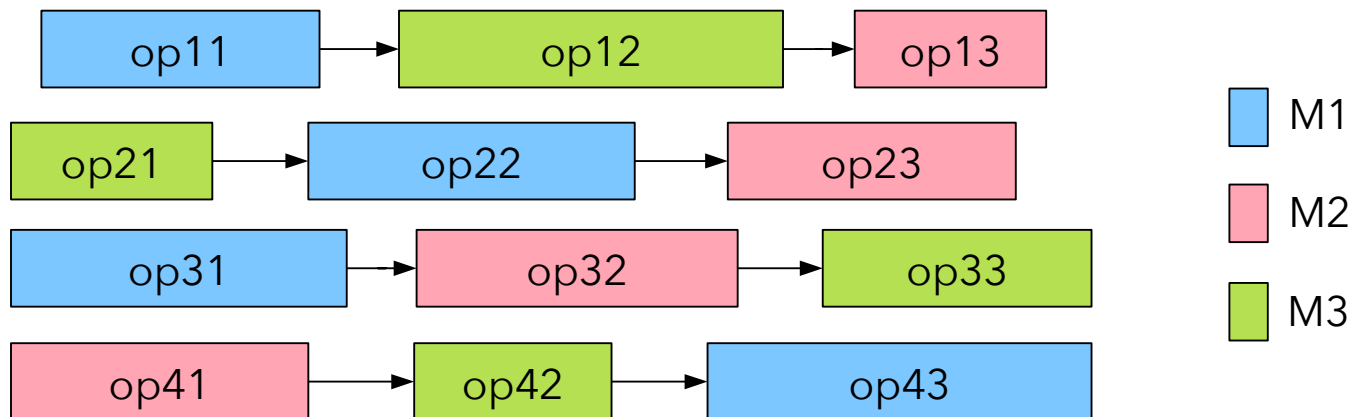


- If both x and y are present and x is before y in sequence p , then x is constrained to end before the start of y

Modeling: Interval variables sequencing

Example: Job-shop scheduling problem

```
1 dvar interval op[j in Jobs][p in Pos] size Ops[j][p].pt;  
2 dvar sequence mchs[m in Mchs] in  
3   all(j in Jobs, p in Pos: Ops[j][p].mch == m) op[j][p];  
4  
5 minimize max(j in Jobs) endOf(op[j][nbPos]);  
6 subject to {  
7   forall(m in Mchs)  
8     noOverlap(mchs[m]);  
9   forall(j in Jobs, p in 2..nbPos)  
10    endBeforeStart(op[j][p-1], op[j][p]);  
11 }
```



Modeling: A parenthesis on performance of this model

- Job-shop scheduling is a heavily studied problem: more than 60.000 references on Google Scholar (using MIP, CP and many meta-heuristics)
- Still, CP Optimizer was able to improve some lower and upper bounds on famous open instances (CPAIOR-2015)

Job Shop

Instance	LB	UB
tail11	1357	1357
tail12	1367	1367
tail13	1342	1342
tail15	1339	1339
tail16	1360	1360
tail18	1377	1396
tail19	1332	1332
tail20	1348	1348
tail21	1642	1642
tail22	1561	1600
tail23	1518	1557

Instance	LB	UB
tail24	1644	1644
tail25	1558	1595
tail26	1591	1643
tail27	1652	1680
tail28	1603	1603
tail29	1573	1625
tail30	1519	1584
tail33	1788	1791
tail40	1651	1669
tail41	1906	2005
tail42	1884	1937

Instance	LB	UB
tail44	1948	1979
tail46	1957	2004
tail47	1807	1889
tail49	1931	1961
tail50	1833	1923
abz07	656	656
abz08	648	667
abz09	678	678
swv03	1398	1398
swv04	1464	1464
swv05	1424	1424

Instance	LB	UB
swv06	1630	1671
swv07	1513	1595
swv08	1671	1752
swv09	1633	1655
swv10	1663	1743
yam1	854	884
yam2	870	904
yam3	859	892
yam4	929	968

Modeling: A parenthesis on performance of this model

- A detailed comparison with different MIP models was performed in [1]:

Problem	Disjunctive		Disjunctive (multi)		Disjunctive (multi+tune)		CP	
	Time (geo/arith)	Opt	Time (geo/arith)	Opt	Time (geo/arith)	Opt	Time (geo/arith)	Opt
3 × 3	0.01 / 0.01	10	0.02 / 0.02	10	0.02 / 0.02	10	0.00 / 0.00	10
4 × 3	0.01 / 0.01	10	0.03 / 0.03	10	0.13 / 0.13	10	0.00 / 0.00	10
5 × 3	0.02 / 0.02	10	0.07 / 0.07	10	0.03 / 0.03	10	0.00 / 0.00	10
3 × 6	0.01 / 0.01	10	0.04 / 0.04	10	0.02 / 0.02	10	0.00 / 0.00	10
3 × 8	0.01 / 0.01	10	0.03 / 0.03	10	0.07 / 0.07	10	0.00 / 0.00	10
3 × 10	0.01 / 0.01	10	0.04 / 0.04	10	0.07 / 0.07	10	0.00 / 0.00	10
5 × 5	0.02 / 0.02	10	0.04 / 0.04	10	0.07 / 0.08	10	0.00 / 0.00	10
8 × 8	0.60 / 0.61	10	0.30 / 0.30	10	0.32 / 0.32	10	0.15 / 0.15	10
10 × 10	3.19 / 3.48	10	1.65 / 1.70	10	1.72 / 1.76	10	0.67 / 0.68	10
12 × 12	99.43 / 212.76	10	37.51 / 71.74	10	33.19 / 47.67	10	3.58 / 3.88	10
15 × 15	1568.89 / 2272.75	5	865.29 / 1595.79	8	926.83 / 1920.19	7	39.57 / 47.71	10
20 × 15	-	-	-	-	-	-	1037.81 / 1924.35	6

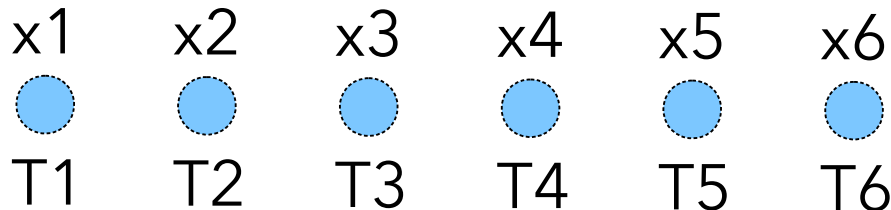
Table 4: Comparison of the best MIP model with multi-threading and parameter tuning (using CPLEX) and the CP model. Bold numbers indicate the best model for the given problem size. The symbol ‘-’ means that none of the 10 problem instances were solved to optimality within 3600 seconds.

[1] W-Y. Ku, J. C. Beck. *Mixed Integer Programming models for job shop scheduling: A computational analysis*. Computers & Operations Research, Vol. 73, pp165-173, 2016.

No-Overlap constraint with **transition distance matrix**

- At creation time of a sequence variable, each interval variable can be associated an integer type

```
int T[i in 1..n] = ...;  
dvar interval x[i in 1..n] ...;  
dvar sequence p  
    in      all(i in 1..n) x[i]  
    types  all(i in 1..n) T[i];
```



- Types will be used to index transition matrices

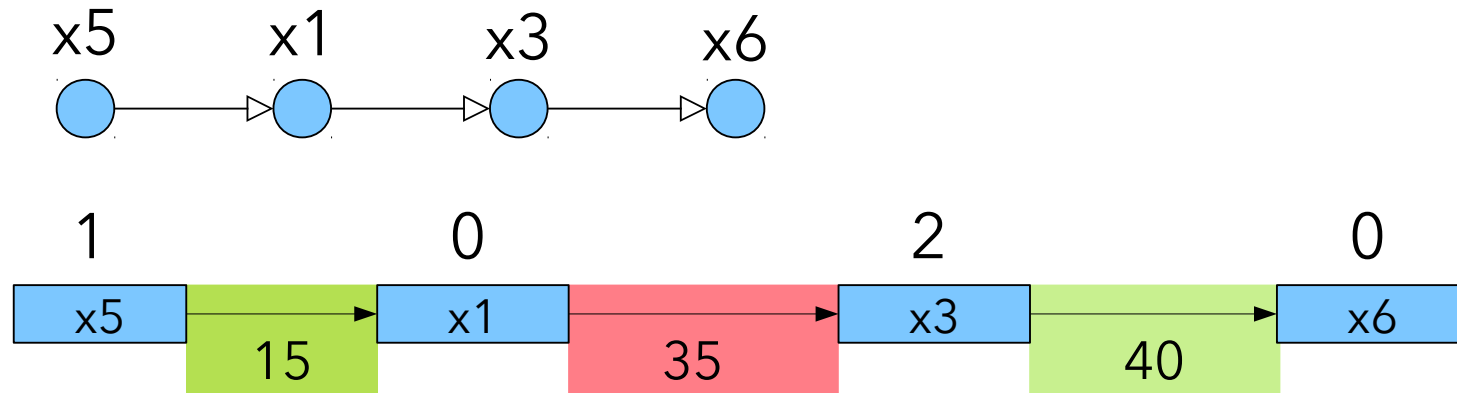
No-Overlap constraint with **transition distance matrix**

- Let p be a sequence variable and M a transition matrix. No-overlap constraint `noOverlap(p,M,direct)` means that p represents a chain of non overlapping interval variables with transition distance M in between consecutive intervals
- Boolean parameter `direct` tells if the transition distance is only applied to direct successors (`direct=true`) or also to indirect ones (`direct=false`, default)

No-Overlap constraint with **transition distance matrix**

- `noOverlap(p,M,true)`

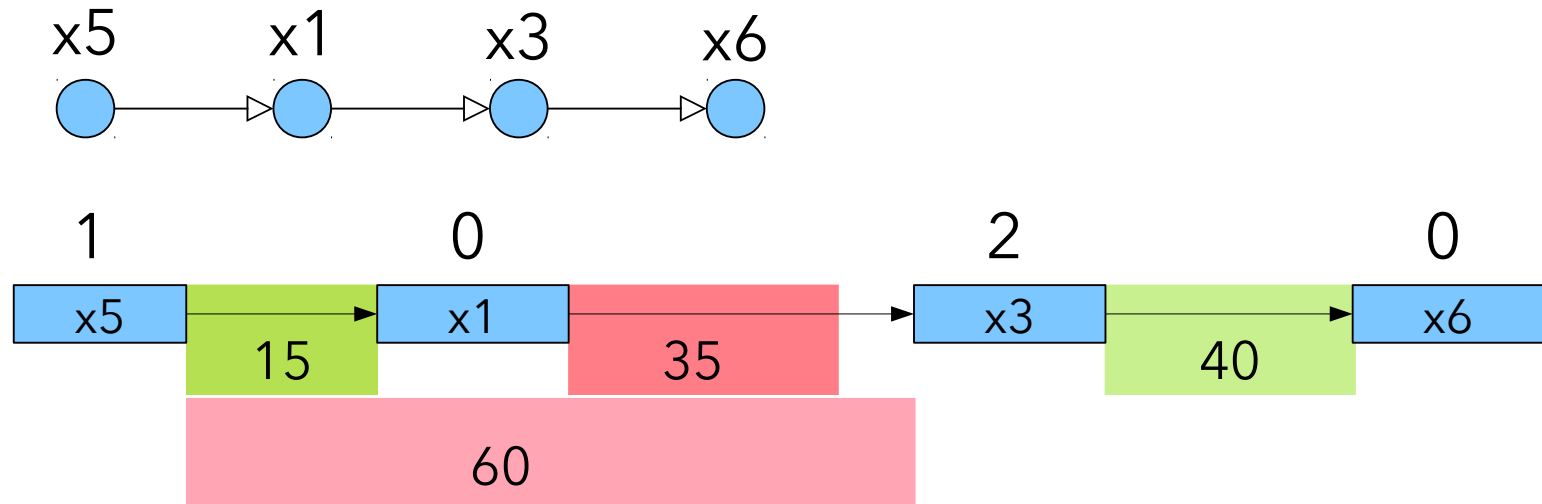
	0	1	2
0	0	30	35
1	15	5	60
2	40	55	0



No-Overlap constraint with **transition distance matrix**

- `noOverlap(p,M,false)`

	0	1	2
0	0	30	35
1	15	5	60
2	40	55	0





No-Overlap constraint with **transition distance matrix**

- Note that if M satisfies the triangle inequality, $\text{noOverlap}(p, M, \text{false})$ and $\text{noOverlap}(p, M, \text{true})$ are equivalent

Modeling: Interval variables sequencing

Example: Vehicle Routing Problem with Time Windows

```
1 using CP;
2 int n = ...; // Number of visits
3 int m = ...; // Number of vehicles
4 int D[0..n+1][1..m] = ...; // Visit durations
5 tuple window { int start; int end; };
6 window W[1..n] = ...; // Visit time-windows
7 tuple triplet { int id1; int id2; int value; };
8 {triplet} M[1..m] = ...; // Travel times for each vehicle
9
10 dvar interval visit[i in 1..n] in W[i].start..W[i].end;
11 dvar interval vvisit[i in 0..n+1][j in 1..m] optional size D[i][j];
12 dvar sequence route[j in 1..m]
13   in all(i in 0..n+1) vvisit[i][j] types all(i in 0..n+1) i;
14
15 minimize sum(j in 1..m) endOf(vvisit[n+1][j]);
16 subject to {
17   forall(j in 1..m) {
18     noOverlap(route[j], M[j], true);
19     presenceOf(vvisit[0][j]);
20     presenceOf(vvisit[n+1][j]);
21     first(route[j], vvisit[0][j]); // Departure from depot
22     last(route[j], vvisit[n+1][j]); // Return to depot
23   }
24   forall(i in 1..n)
25     alternative(visit[i], all(j in 1..m) vvisit[i][j]);
26 }
```

Sequence expressions

- Integer expressions to get the type/start/end/length/size of the interval variable that is next/previous to an interval variable x in a sequence p
`typeOfNext(p, x, l, v)`
- Integer l is the value of the expression if x is present and is the last interval in sequence p
- Integer v is the value of the expression if interval x is absent (default: 0)
- Similar integer expressions:
`startOfNext, endOfNext, sizeOfNext, lengthOfNext,`
`startOfPrev, endOfPrev, sizeOfPrev, lengthOfPrev,`
`typeOfPrev`

Modeling: Interval variables sequencing

What for ?

- A typical use-case is for expressing transition costs

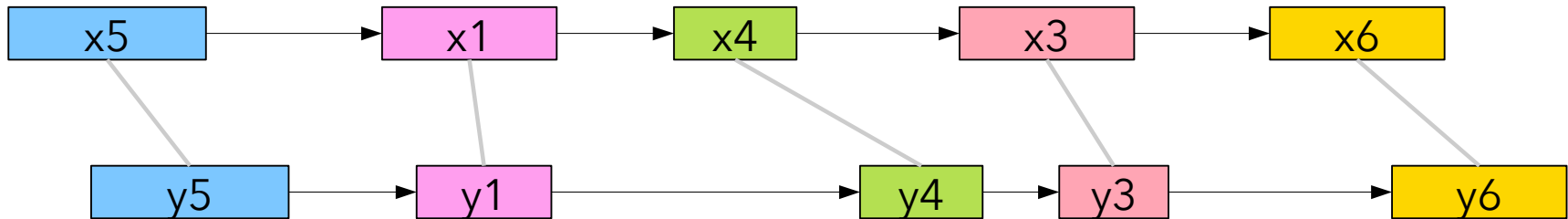
```
1 int n = ...; // Number of activities
2 int m = ...; // Number of types
3 int T[i in 1..n]= ...; // Type of activity act[i]
4 float C[0..m][0..m] = ...; // Transition costs
5 dvar interval o[i in 1..n];
6 dvar sequence seq in o types T;
7 minimize sum(i in 1..n) C[T[i]][typeOfNext(seq,o[i],0,0)];
8 subject to {
9     noOverlap(seq);
10 }
```

Same-sequence and Same-common-subsequence constraints

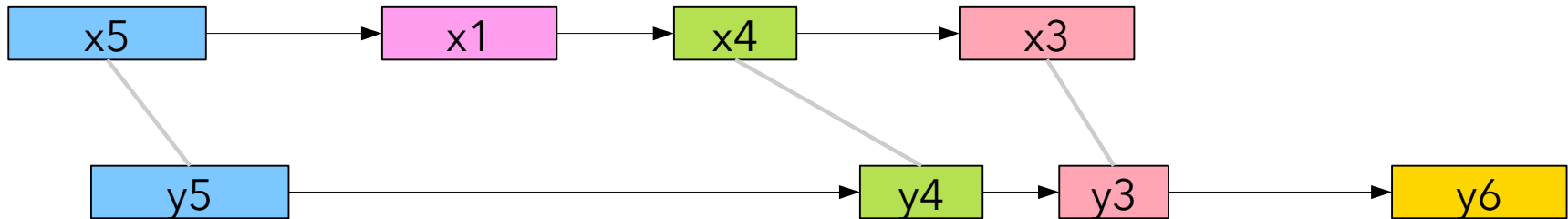
- What for ?
 - Modeling First-in/First-out and No-bypass constraints in some physical systems (trains on single-line railways, items on a conveyor belt)
 - Stochastic scheduling: scenario-based approaches for scheduling with uncertainties with the objective to build robust sequences

Same-sequence and Same-common-subsequence constraints

`sameSequence(px,py)`

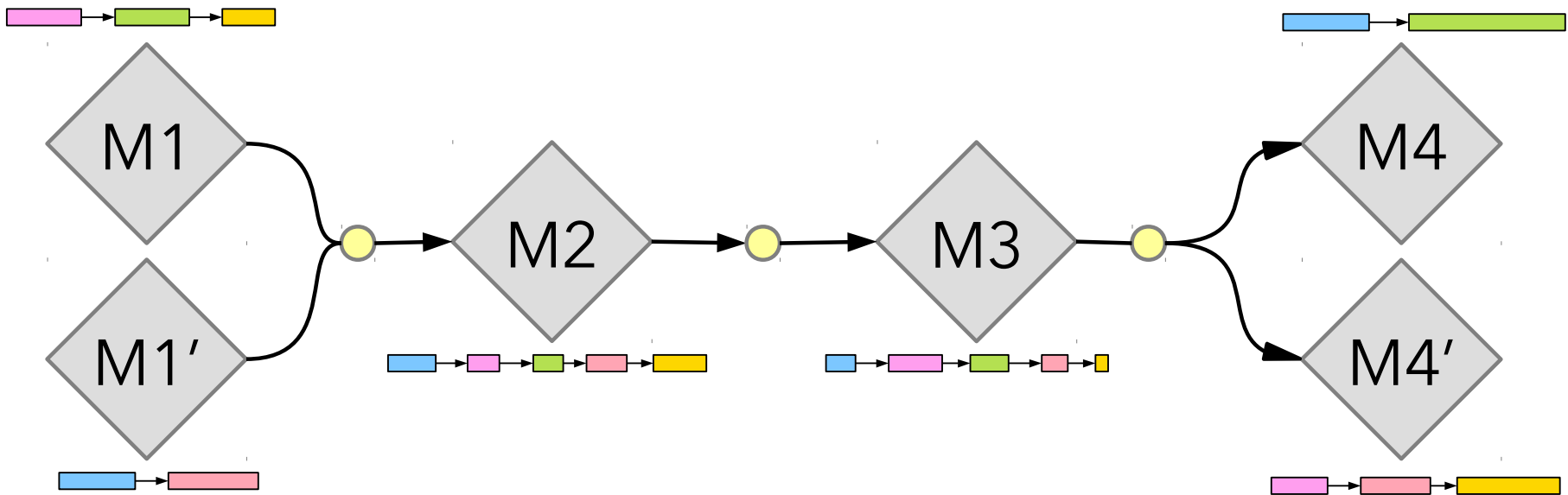


Same-sequence and Same-common-subsequence constraints
`sameCommonSubsequence(px,py)`



Same-sequence and Same-common-subsequence constraints

- No-bypass constraint (e.g. conveyor belts, train scheduling)



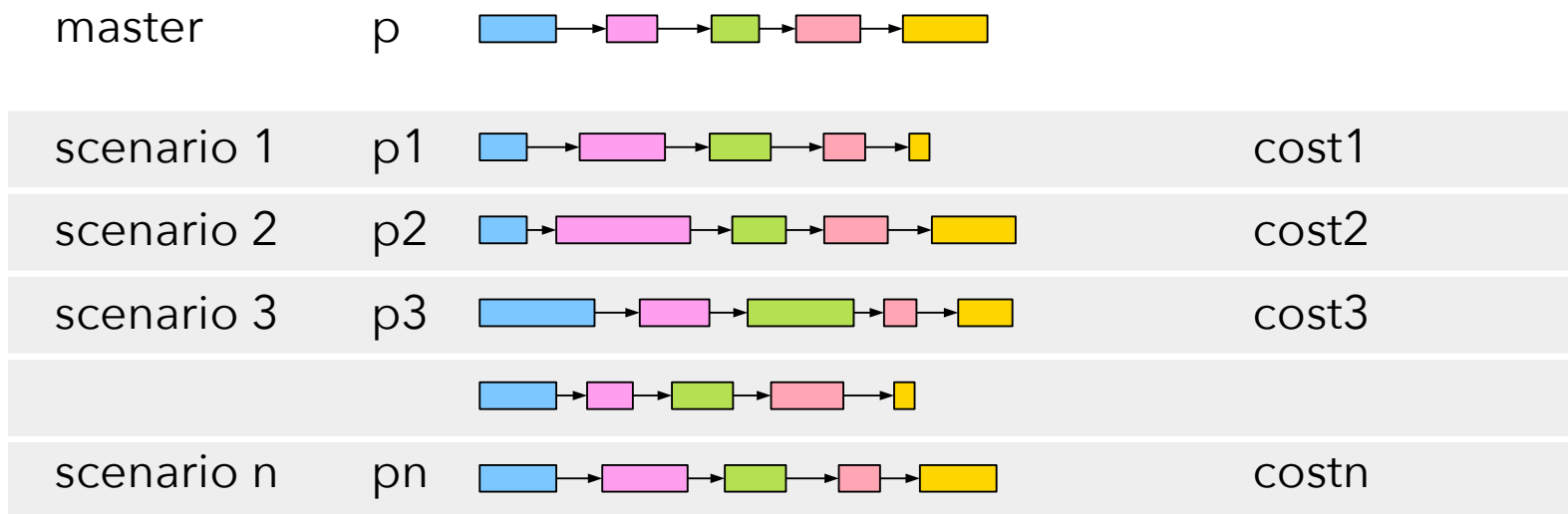
`sameSequence(p2,p3)`

`sameCommonSubsequence(p1,p2)`
`sameCommonSubsequence(p1',p2)`

`sameCommonSubsequence(p3,p4)`
`sameCommonSubsequence(p3,p4')`

Same-sequence and Same-common-subsequence constraints

- Stochastic scheduling: sequences as 1st stage variables in 2-stage stochastic programming



$$\forall i \in [1, n]: \text{sameSequence}(p, p_i)$$

$$\text{cost} = 1/n * \sum_{i \in [1, n]} \text{cost}_i$$

Modeling: Interval variables sequencing

- To summarize:

Variable: sequence variable p

Constraints:

`first(p,x)`

`last(p,x)`

`prev(p,x,y)`

`before(p,x,y)`

`noOverlap(p)`

`noOverlap(p,M)`

`sameSequence(p1,p2)`

`sameCommonSubsequence(p1,p2)`

Expressions:

`typeOfNext(p,x,l,v)`

`startOfNext(p,x,l,v)`

`endOfNext(p,x,l,v)`

`sizeOfNext(p,x,l,v)`

`lengthOfNext(p,x,l,v)`

Similar ones with Prev

Modeling: Interval variables sequencing

Why introducing new types of variables and expressions when a new constraint (**disjunctive**) could be enough ?

Think how you would model:

- Transition distance
 - Add new constraint variants with a transition matrix ?
- Transition costs
 - Add a cost matrix (same as transition distance?) and a total cost expression (sum)
- What if the costs are not aggregated with a sum ?
- What if you additionally need next/prev variables to express more complex constraints ?
- How to model the equivalent of a **sameSequence** ?

Modeling: Interval variables sequencing

Why introducing new types of variables and expressions when a new constraint (**disjunctive**) could be enough ?

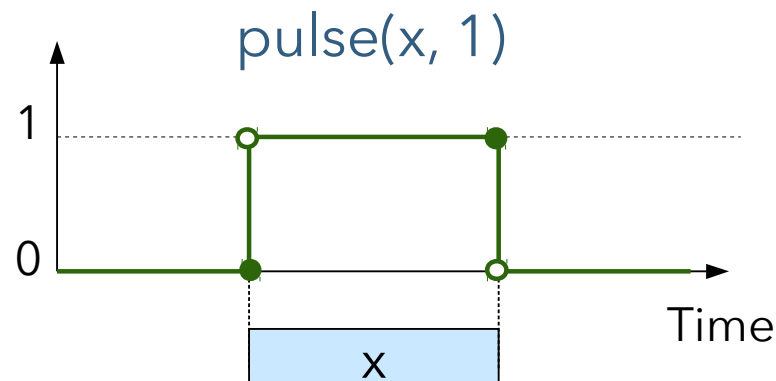
- The notion of a **sequence variable** provides a common structure on which independent aspects of complex sequencing can be posted
- This structure can be exploited by the automatic search

Modeling: Cumul functions

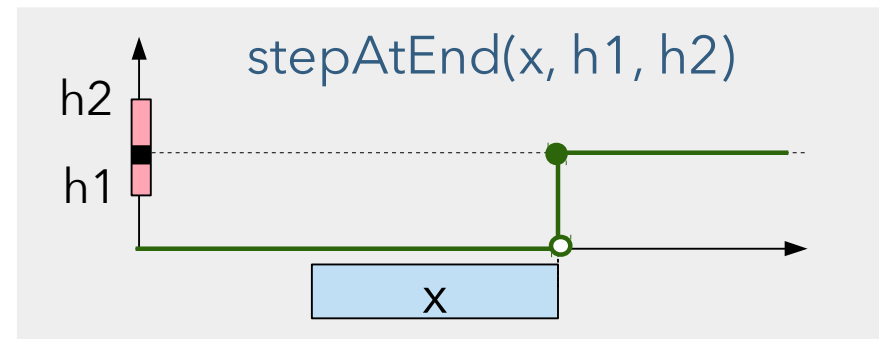
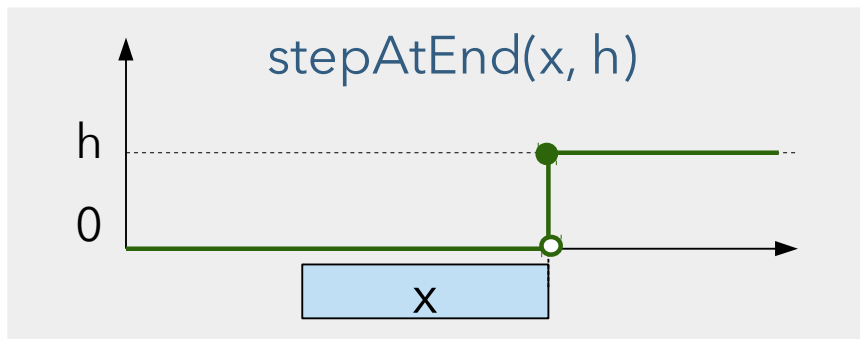
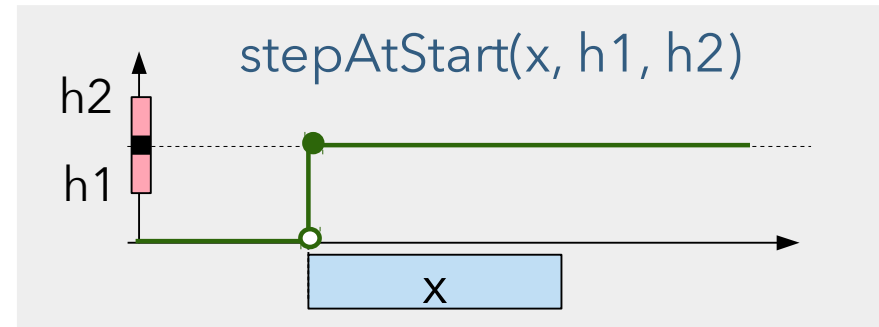
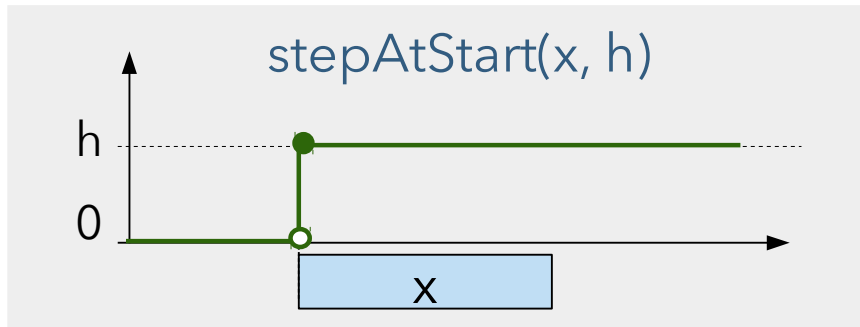
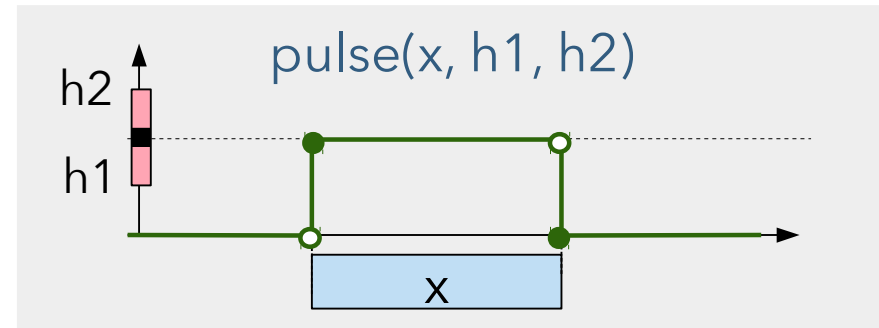
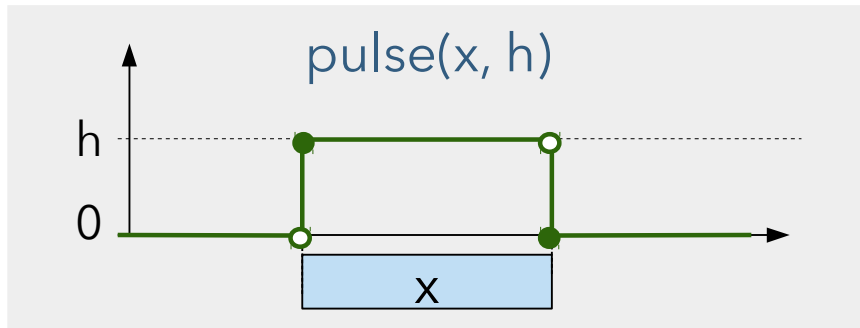
- What for?
 - The value of a **cumul function expression** represents the time evolution of a quantity (e.g. level of an inventory) that can be incrementally changed (increased or decreased) by interval variables
- Examples:
 - The number of workers of a given type
 - The level of an inventory

- The individual contribution of an interval variable x to a cumul function expression is called an **elementary cumul function**
- An **elementary cumul function** is a cumul function expression
- Example:

$$\text{cumulFunction } y = \text{sum}(i \text{ in } 1..n) \text{ pulse}(x[i], 1)$$

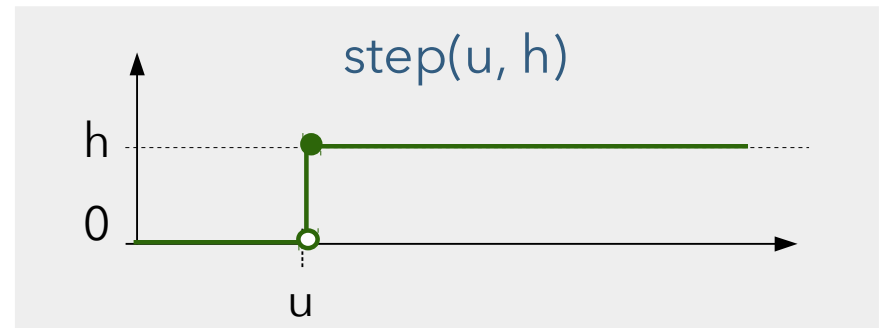
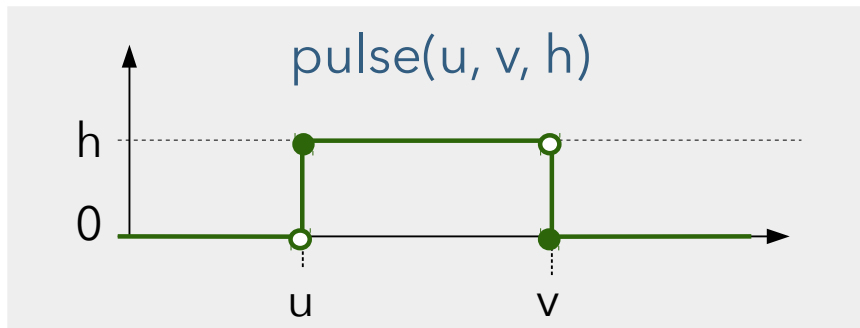


- Elementary cumul functions



Modeling: Cumul functions

- Elementary cumul functions
 - All these elementary cumul functions on an interval variable x are the null function (null contribution) when x is absent
 - There are also some constant elementary cumul functions:



- A cumul function expression f is the algebraic sum of elementary cumul functions f_i or their negation:

$$f = \sum_i \varepsilon_i \cdot f_i \quad \text{with } \varepsilon_i \in \{-1, +1\}$$

- Note the similarity:
 - On scalar expressions:
$$y = \text{sum}(i \text{ in } 1..n) C[i] * \text{presenceOf}(x[i])$$

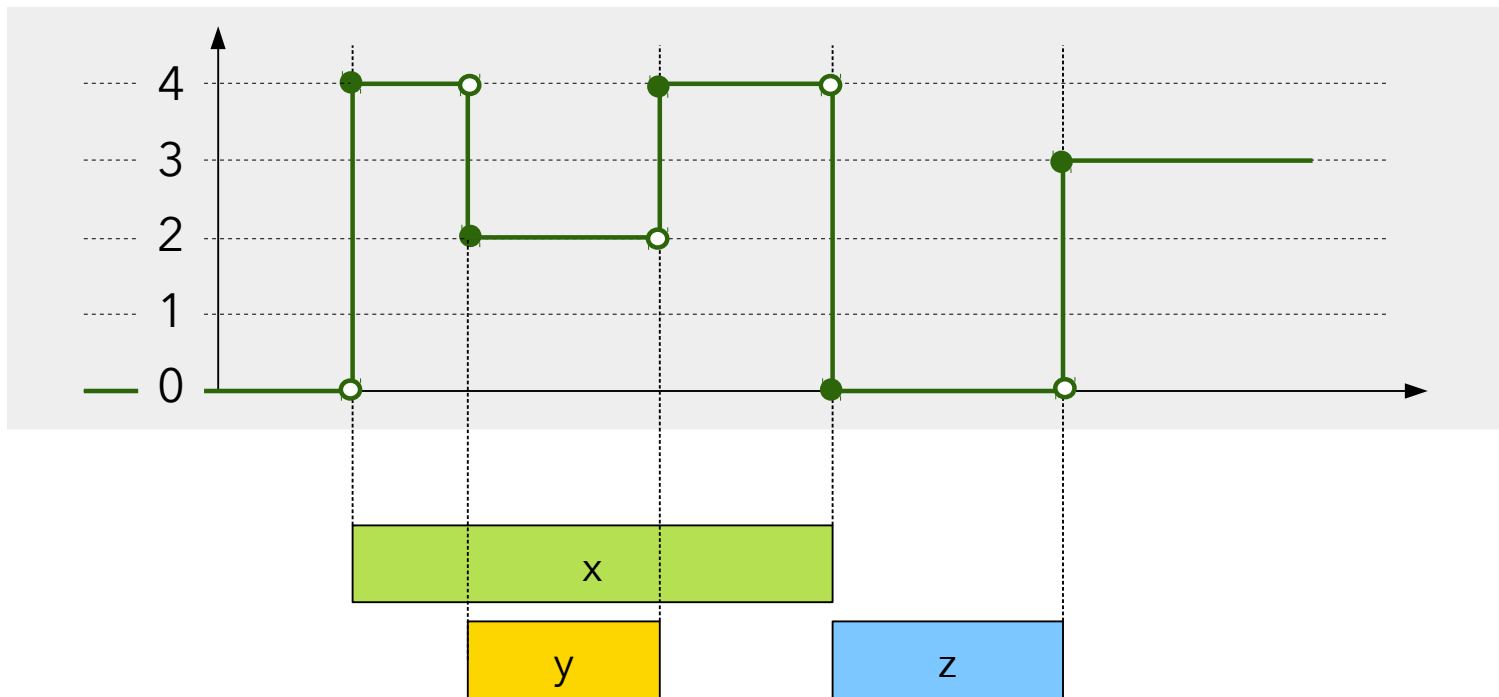
y is a numerical expression
 - We add the temporal dimension of scheduling and we naturally have similar expressions on functions:
$$y = \text{sum}(i \text{ in } 1..n) \text{pulse}(x[i], C[i])$$

y is a (cumul) function expression

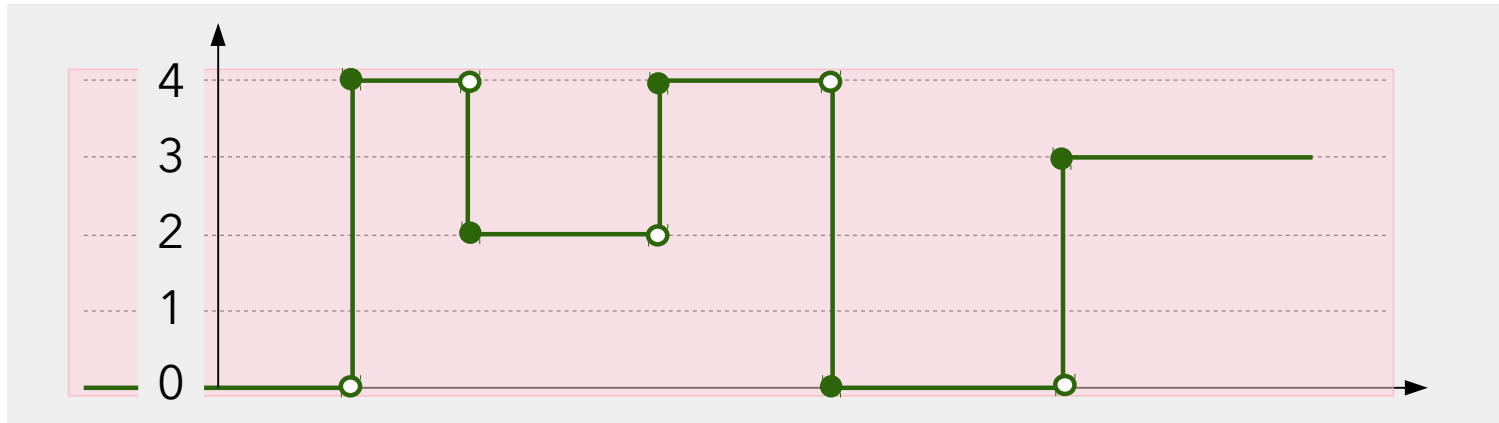
Example of cumul function:

$$\text{cumulFunction } f = \text{pulse}(x,4) - \text{pulse}(y,2) + \text{stepAtEnd}(z,3)$$

Value of the cumul function expression at a solution:

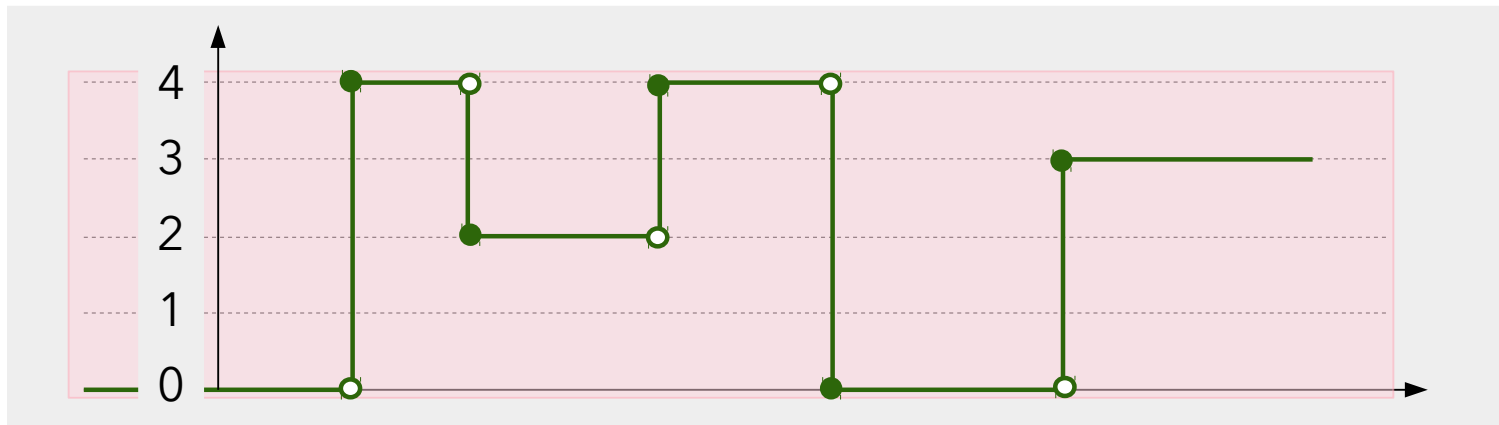


Constraints on cumul function expressions



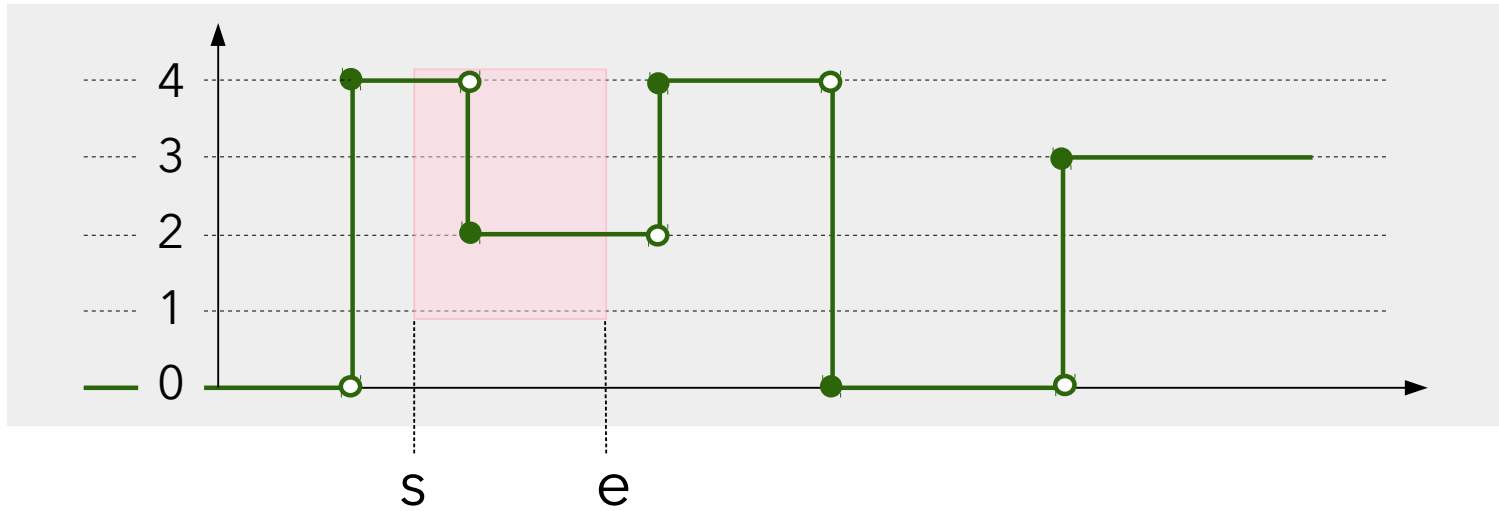
```
int h = ...  
cumulFunction f = ...  
f <= h
```

Constraints on cumul function expressions



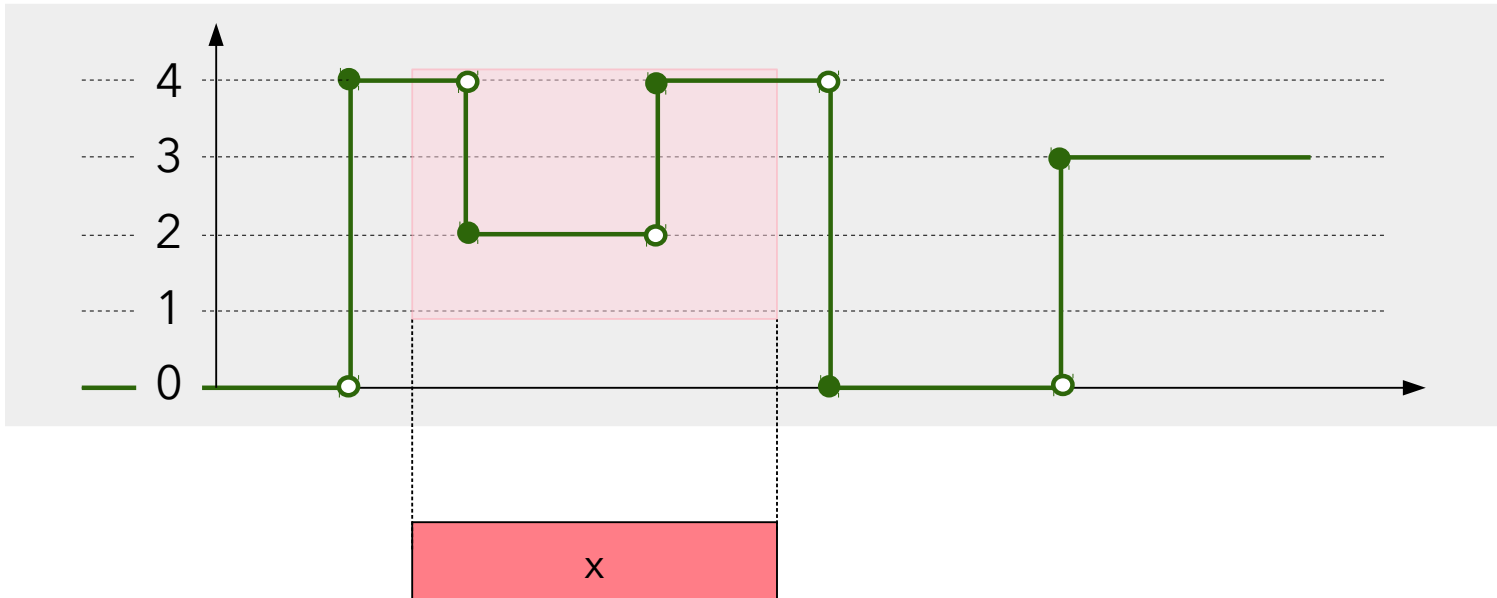
```
dvar int h = ...  
cumulFunction f = ...  
f <= h
```

Constraints on cumul function expressions



```
int s, e, hmin, hmax = ...  
cumulFunction f = ...  
alwaysIn(f, s, e, hmin, hmax)
```

Constraints on cumul function expressions

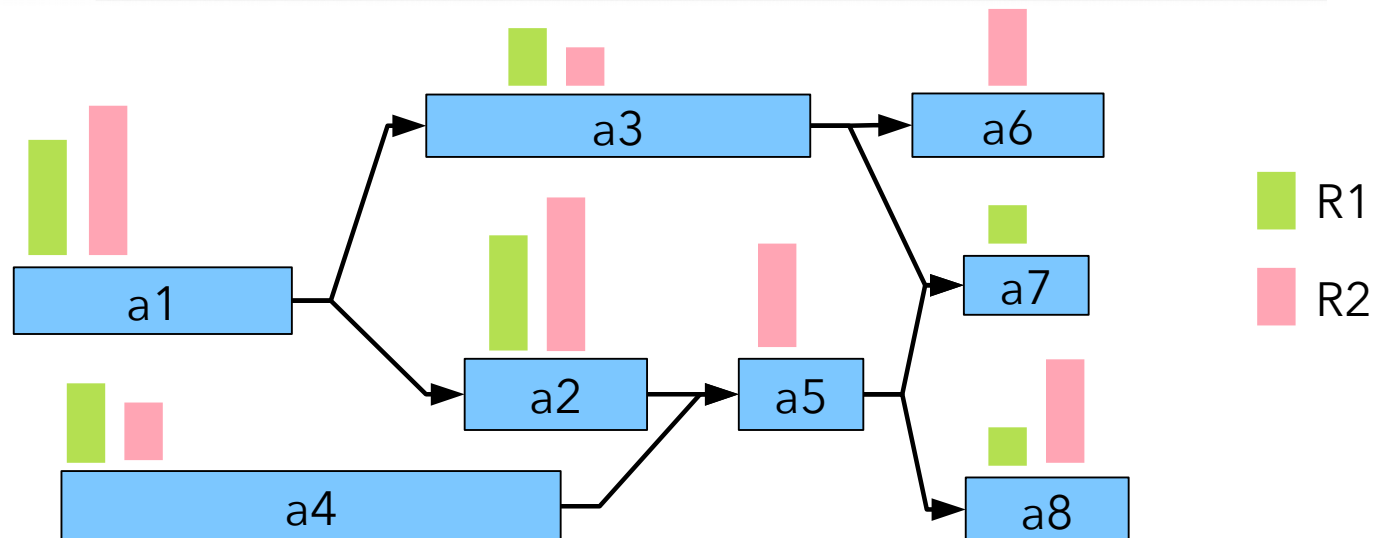


```
int hmin, hmax = ...  
cumulFunction f = ...  
alwaysIn(f, x, hmin, hmax)
```

Modeling: Cumul functions

Example: Resource-constrained project scheduling (RCPSP)

```
1  dvar interval a[i in Tasks] size i.pt;  
2  cumulFunction usage[r in Resources] =  
3    sum(i in Tasks: i.qty[r]>0) pulse(a[i],i.qty[r]);  
4  minimize max(i in Tasks) endOf(a[i]);  
5  subject to {  
6    forall(r in Resources)  
7      usage[r] <= Capacity[r];  
8    forall(i in Tasks, j in i.succs)  
9      endBeforeStart(a[i], a[<j>]);  
10 }
```



Modeling: Cumul functions

- Like Job-shop scheduling, RCPSP is a heavily studied problem
- Still, CP Optimizer was able to improve some lower and upper bounds on famous (though not very realistic) instances (CPAIOR-2015)

RCPSP

Instance	LB	UB
j60_9_5	82	85
j60_9_10	91	93
j60_13_5	93	97
j60_25_8	96	99
j60_29_3	115	121
j60_29_6	145	154
j60_29_10	112	119
j60_45_5	100	106
j60_45_6	133	144
j60_45_10	105	114
j90_5_4	101	102
j90_9_1	100	104
j90_9_9	107	116
j90_13_6	118	124

Instance	LB	UB
j90_13_7	117	124
j90_25_2	123	131
j90_25_3	115	123
j90_25_7	123	130
j90_25_8	133	140
j90_29_2	123	126
j90_29_4	141	149
j90_41_2	158	168
j90_41_4	144	154
j90_41_7	146	157
j90_41_10	147	150
j90_45_5	164	174
j120_8_5	101	104

Instance	LB	UB
j120_11_3	190	203
j120_11_4	183	196
j120_11_10	166	181
j120_12_3	133	136
j120_16_5	185	200
j120_16_9	190	205
j120_17_4	118	120
j120_17_9	130	134
j120_26_3	161	167
j120_31_6	184	192
j120_31_8	177	192
j120_31_10	203	227
j120_36_10	199	216

Instance	LB	UB
j120_37_3	136	139
j120_37_4	157	163
j120_37_7	152	161
j120_39_2	106	108
j120_46_5	140	149
j120_47_8	127	133
j120_48_2	112	113
j120_48_6	103	105
j120_51_1	187	206
j120_52_10	135	144
j120_57_2	152	161
j120_58_1	134	141
j120_59_2	104	106

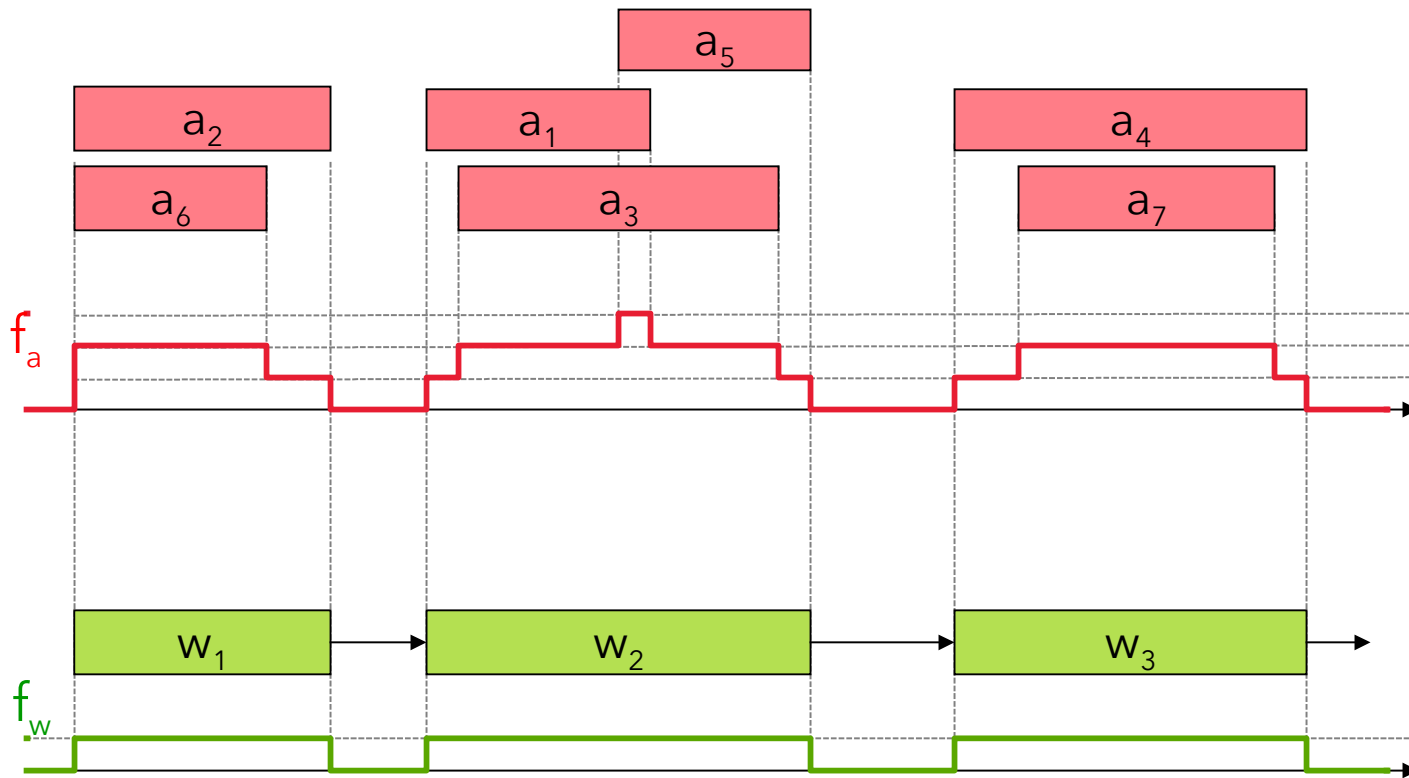
Modeling: Cumul functions

Example: Intervals when a cumulative resource is in use

```
1  int C = ...;
2  dvar interval a[i in 1..n] size ...;
3  dvar interval w[j in 1..m] optional size 1..H;
4  cumulFunction fa = sum(i in 1..n) pulse(a[i],1);
5  cumulFunction fw = sum(j in 1..m) pulse(w[j],1);
6  constraints {
7    fa <= C;
8    forall(j in 2..m) {
9      presenceOf(w[j]) => presenceOf(w[j-1]);
10     endBeforeStart(w[j-1],w[j],1);
11   }
12   forall(i in 1..n) {
13     alwaysIn(fw,a[i],1,1);
14   }
15   forall(j in 1..m) {
16     alwaysIn(fa,w[j],1,n);
17   }
18 }
```

Modeling: Cumul functions

Example: Intervals when a cumulative resource is in use



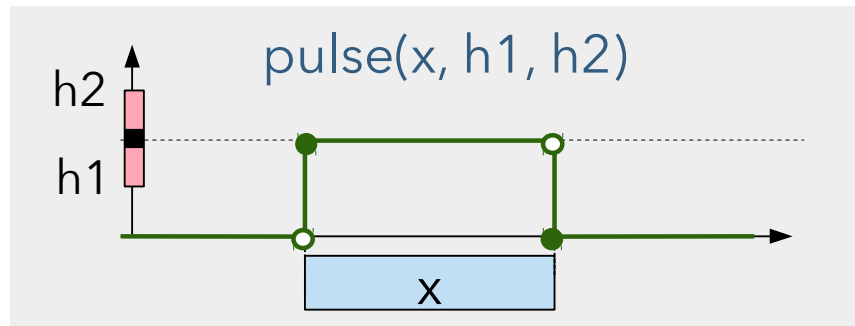
Cumul height expressions

- Integer expressions are available to get the contribution of a given interval variable x to a cumul function f at its start or end point (with default value v if x is absent)

`heightAtStart(f, x, v)`

`heightAtEnd(f, x, v)`

- This is useful to constrain the height of an elementary cumul function



Modeling: Cumul functions

- To summarize:

Expression: cumul function $f = \sum_i \varepsilon_i \cdot f_i$ with $\varepsilon_i \in \{-1, +1\}$

f_i are elementary cumul functions:

pulse(x,h), pulse(x,h1,h2),
stepAtStart(x,h), stepAtStart(x,h1,h2),
stepAtEnd(x,h), stepAtEnd(x,h1,h2),
pulse(u,v,h), step(u,h)

Constraints:

$f \leq h$
alwaysIn(f, s, e, hmin, hmax)
alwaysIn(f, x, hmin, hmax)

Expressions:

heightAtStart(f,x,v)
heightAtEnd(f,x,v)

Modeling: Cumul functions

Why introducing new types of expressions when a new constraint (**cumulative**) could be enough ?

Think how you would model:

- The mix of pulse and startAtStart/End with +/- sign
 - Add new constraint variants with types/sign of the elementary functions ?
- The different types of alwaysIn constraints
- The algebra on cumul functions
 - We currently have +, - but we could make it richer in the future

Modeling: Cumul functions

Why introducing new types of expressions when a new constraint (**cumulative**) could be enough ?

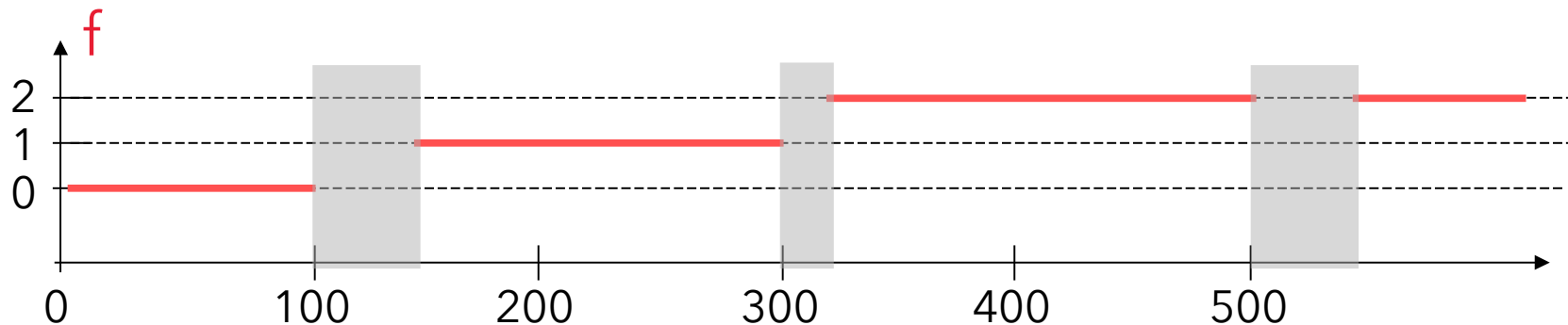
- The notion of a **cumul function** provides a common structure on which independent aspects of complex cumulative scheduling can be posted
- This structure can be exploited by the automatic search

Modeling: State function variables

- What for?
 - The value of a **state function variable** represents the time evolution of a value (e.g. state/configuration of a resource) that can be changed/required by interval variables
 - Two interval variables requiring incompatible states cannot overlap
 - Two interval variables requiring compatible states can (optionally) be batched together
- Examples:
 - Tool installed on a machine
 - Temperature of an oven
 - Type of raw material present in a tank

- The value of a **state function variable** is a set of non-overlapping **segments** over which the function maintains a constant non-negative integer state. In between those segments, the state of the function is not defined.
- For instance for an oven with 3 possible temperature levels (0,1,2) we could have the following time evolution:

[start = 0, end = 100):	state=0
[start = 140, end = 300):	state=1
[start = 320, end = 500):	state=2
[start = 540, end = 600):	state=2



- State function variable declaration

stateFunction f

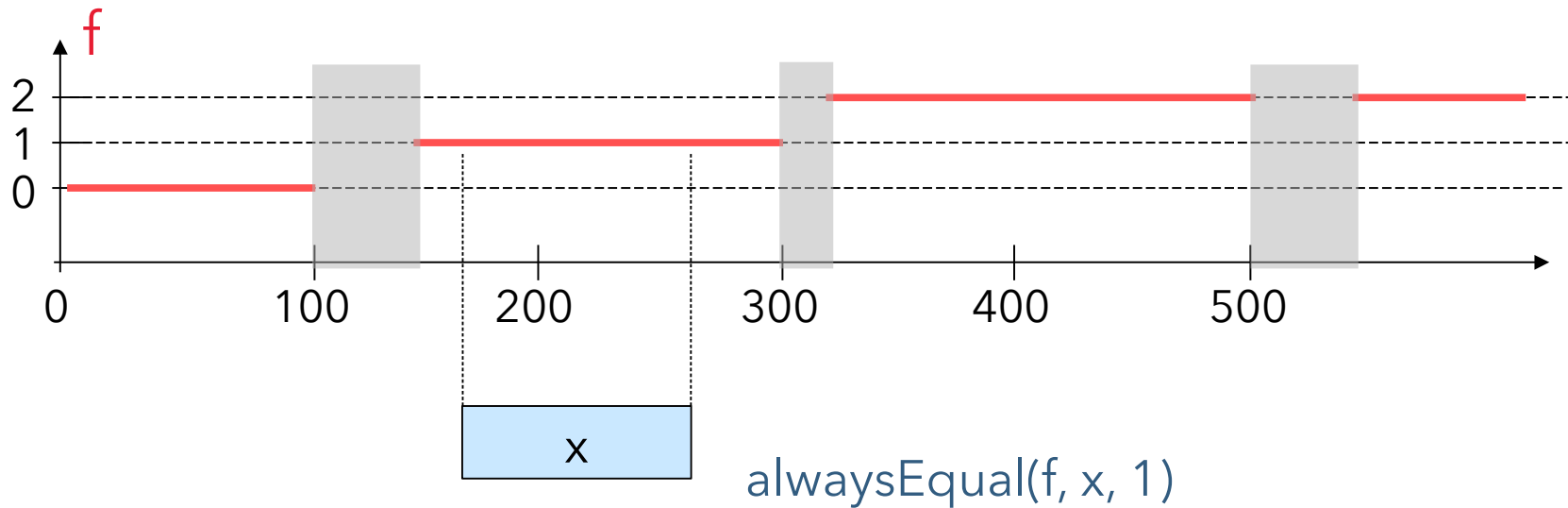
- State function with transition distance matrix M specifying distance between consecutive segments of value i and j

stateFunction f with M

- Note that, unlike cumul functions, state functions are **variables**, not expressions: they are not defined from a set of interval variables
- The value of the state function will be **decided** by the engine so as to satisfy all the constraints on the state function

- Constraints on state functions
 - Over an interval variable x (if present), state function f is always defined and its state is v :

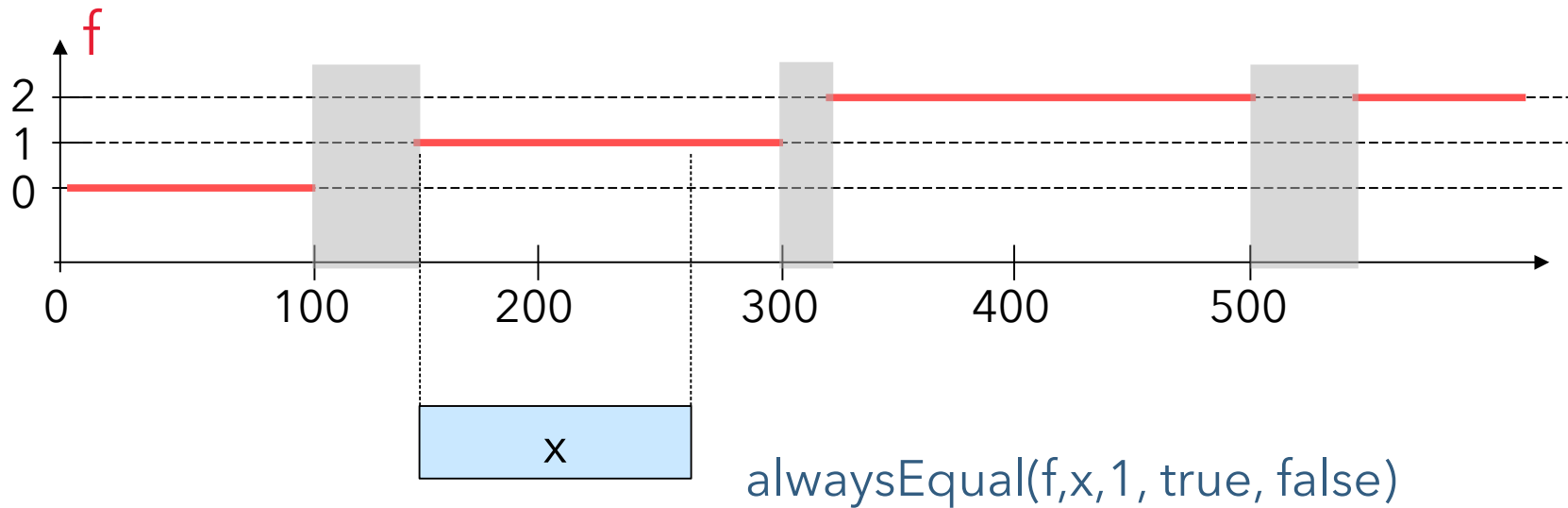
$\text{alwaysEqual}(f, x, v)$



- Constraints on state functions
 - Notion of a start and end alignment with respect to the segment of the state function:

$\text{alwaysEqual}(f, x, v, \text{true}, \text{false})$

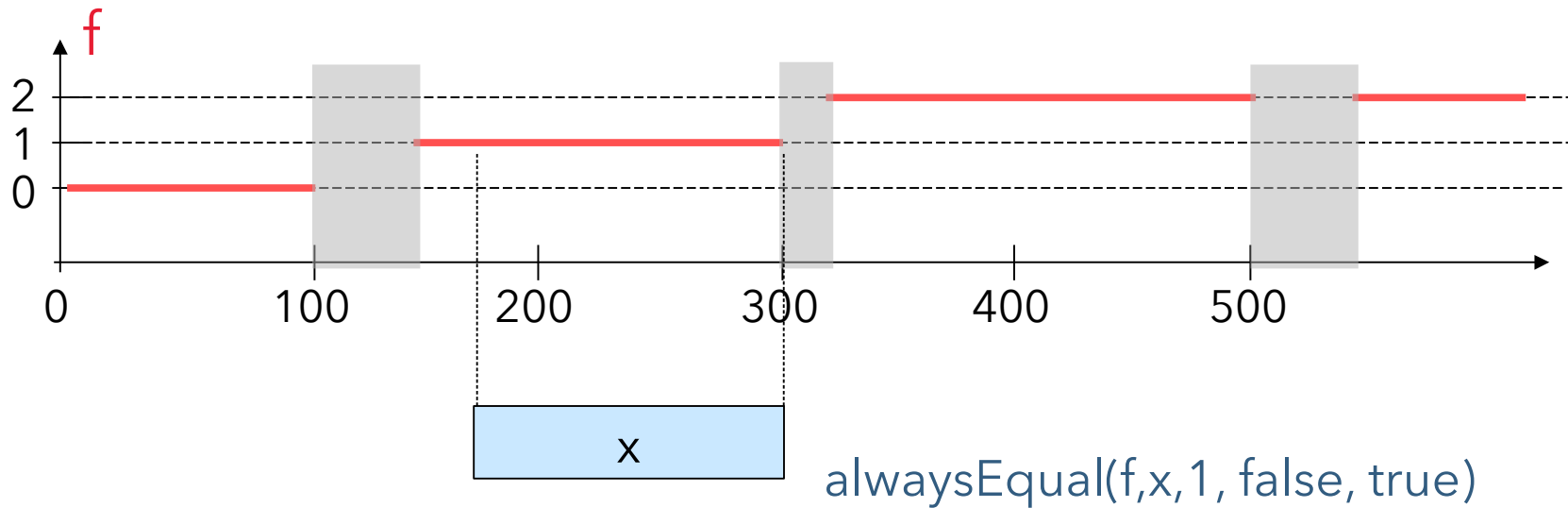
↑
start-aligned interval variable



- Constraints on state functions
 - Notion of a start and end alignment with respect to the segment of the state function:

$\text{alwaysEqual}(f, x, v, \text{false}, \text{true})$

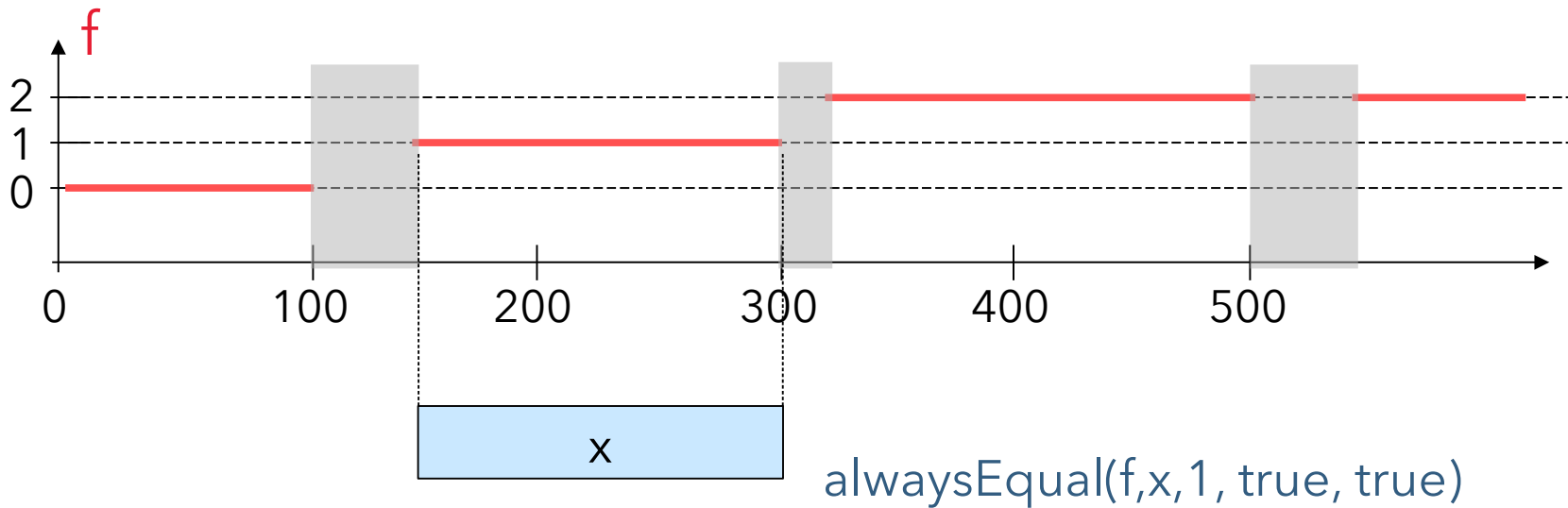
↑
end-aligned interval variable



- Constraints on state functions
 - Notion of a start and end alignment with respect to the segment of the state function:

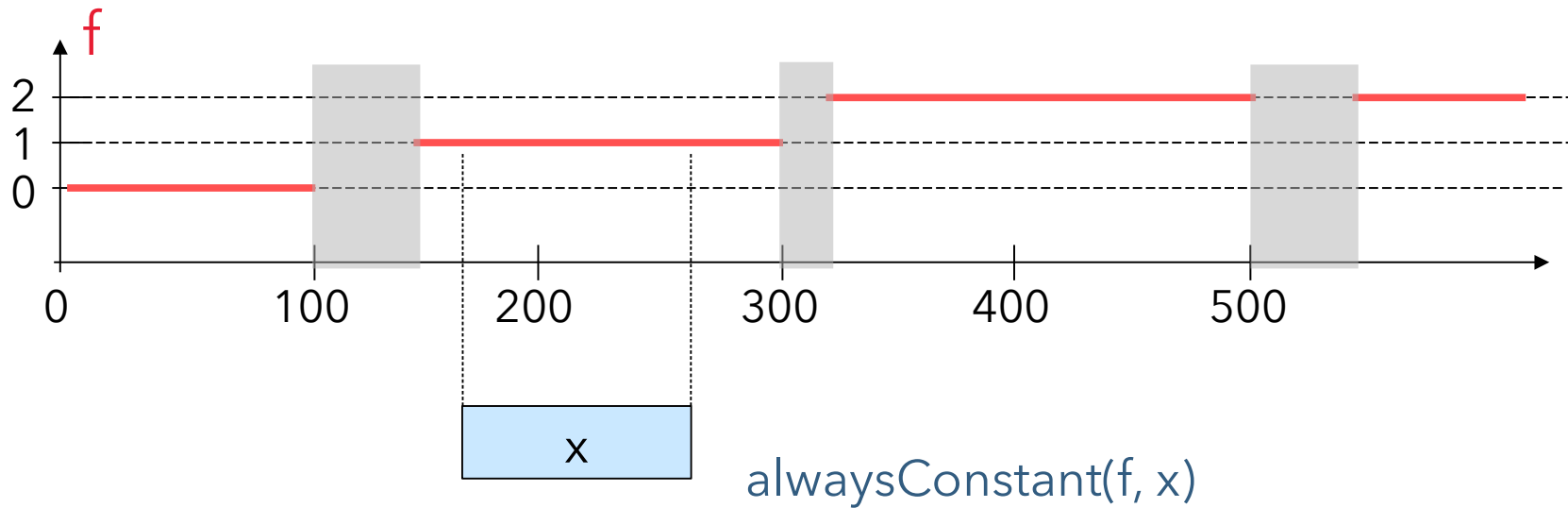
`alwaysEqual(f, x, v, true, true)`

↑ ↑
start and end-aligned interval variable



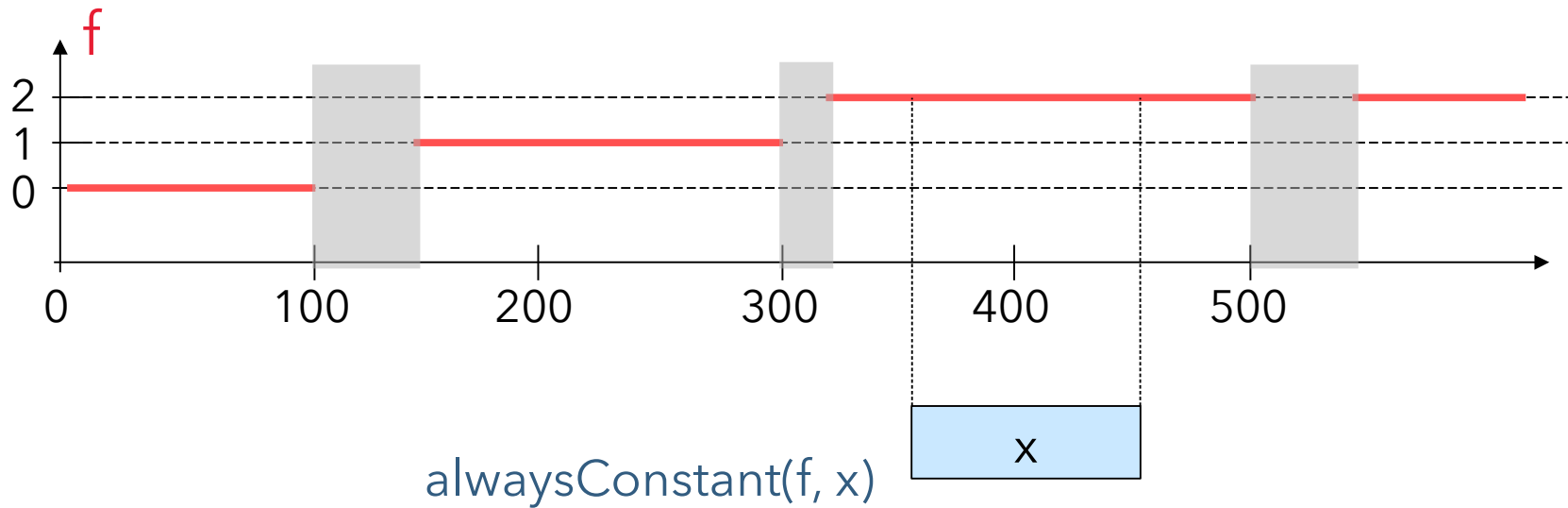
- Constraints on state functions
 - Over an interval variable x (if present), state function f is always defined and has a constant (not specified) state:

`alwaysConstant(f, x)`



- Constraints on state functions
 - Over an interval variable x (if present), state function f is always defined and has a constant (not specified) state:

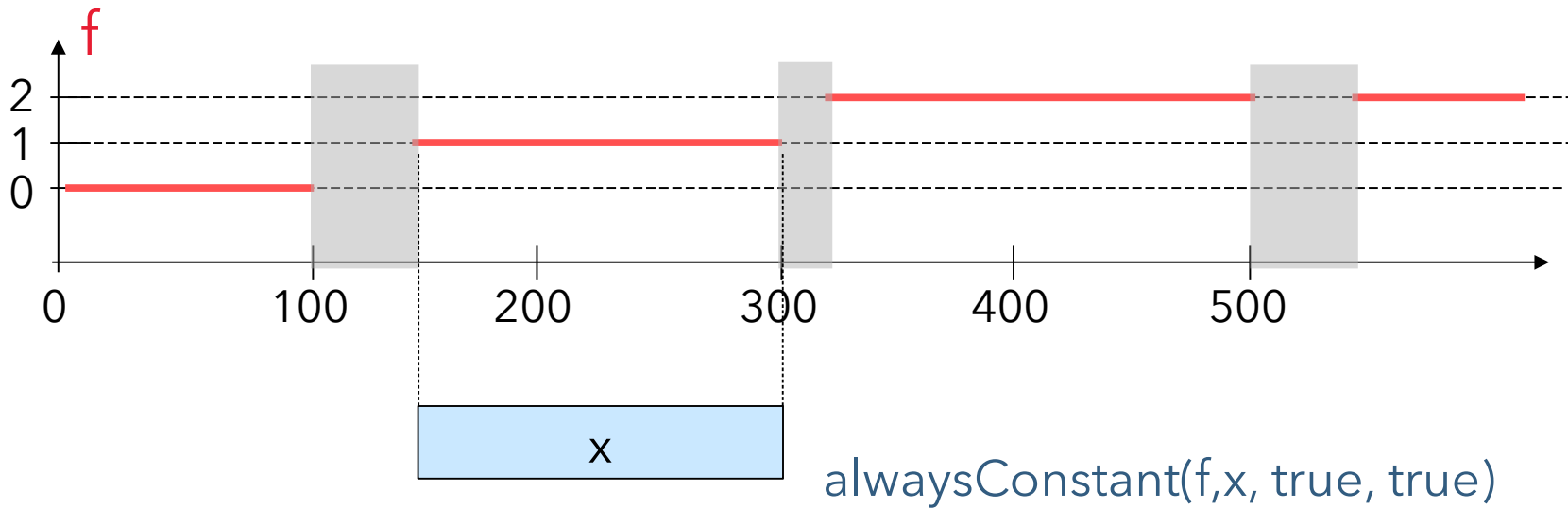
`alwaysConstant(f, x)`



- Constraints on state functions
 - Notion of a start and end alignment with respect to the segment of the state function:

`alwaysConstant(f, x, true, true)`

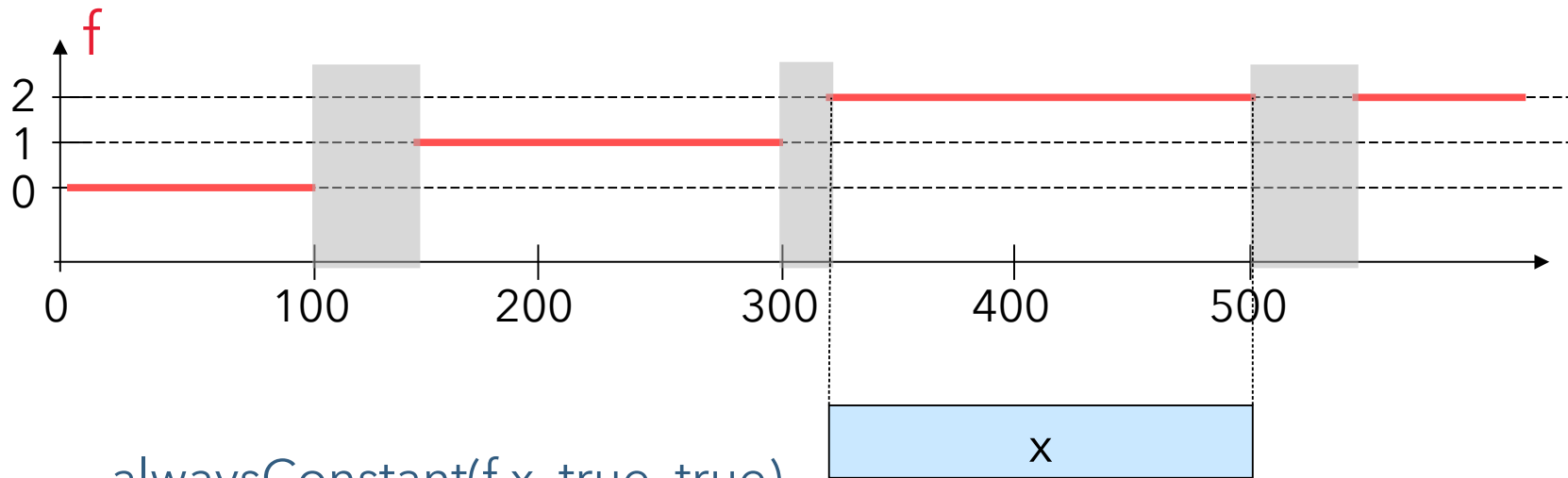
↑ ↑
start and end-aligned interval variable



- Constraints on state functions
 - Notion of a start and end alignment with respect to the segment of the state function:

`alwaysConstant(f, x, true, true)`

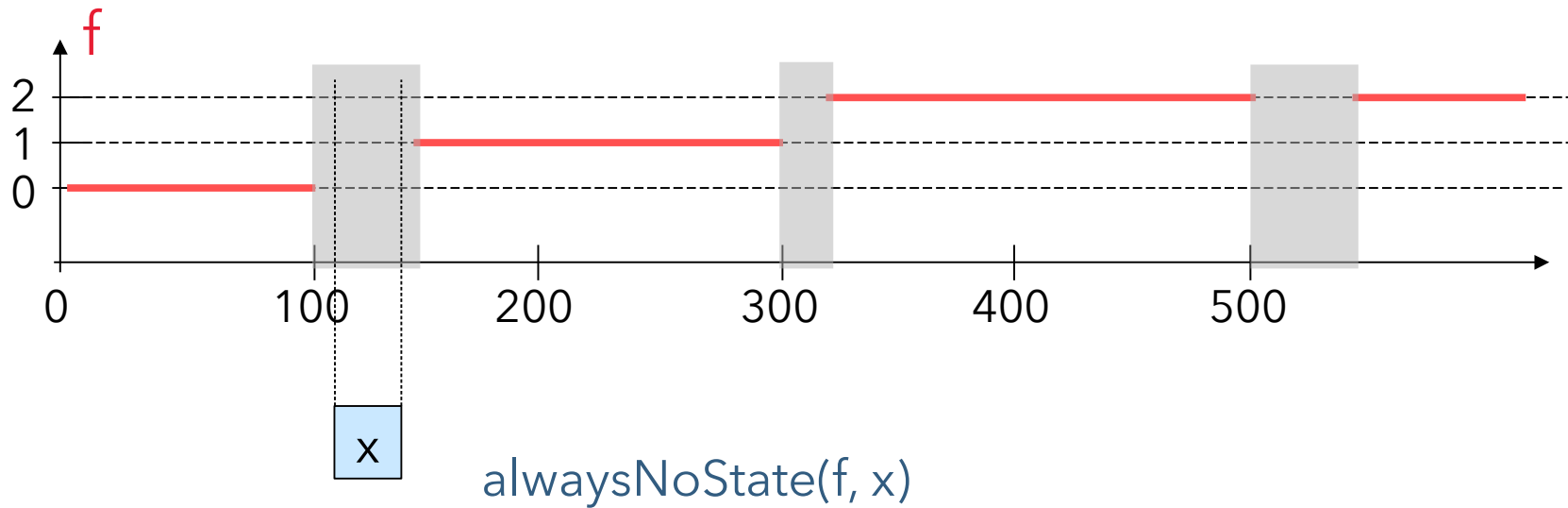
↑ ↑
start and end-aligned interval variable



`alwaysConstant(f,x, true, true)`

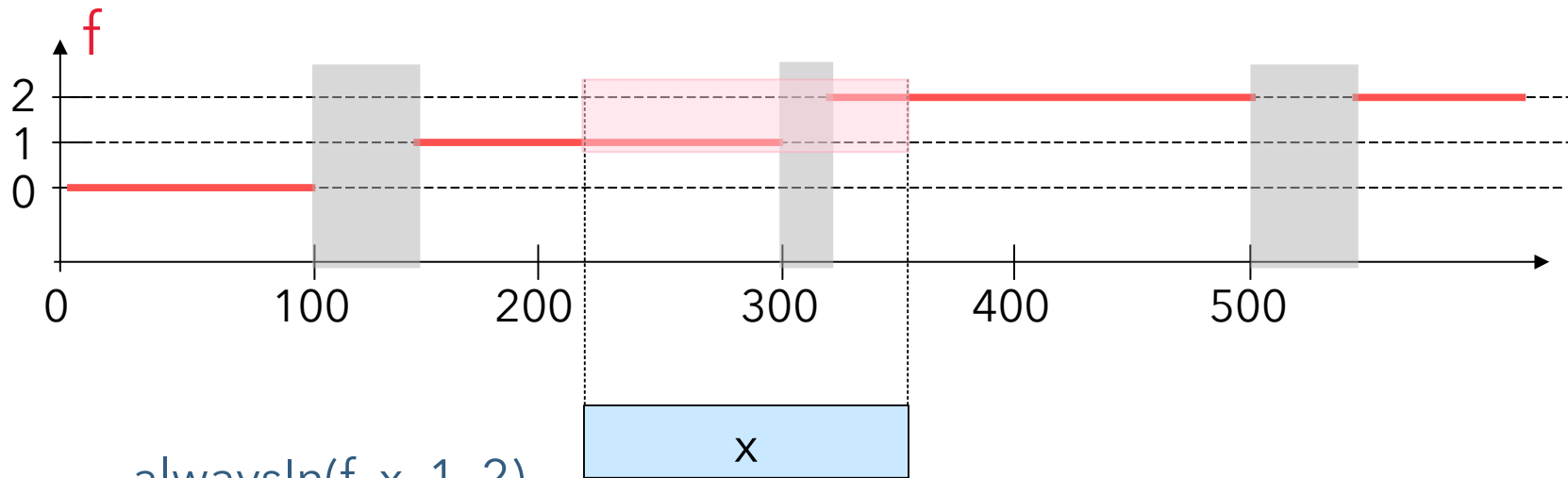
- Constraints on state functions
 - Over an interval variable x (if present), state function f is always not defined:

`alwaysNoState(f, x)`



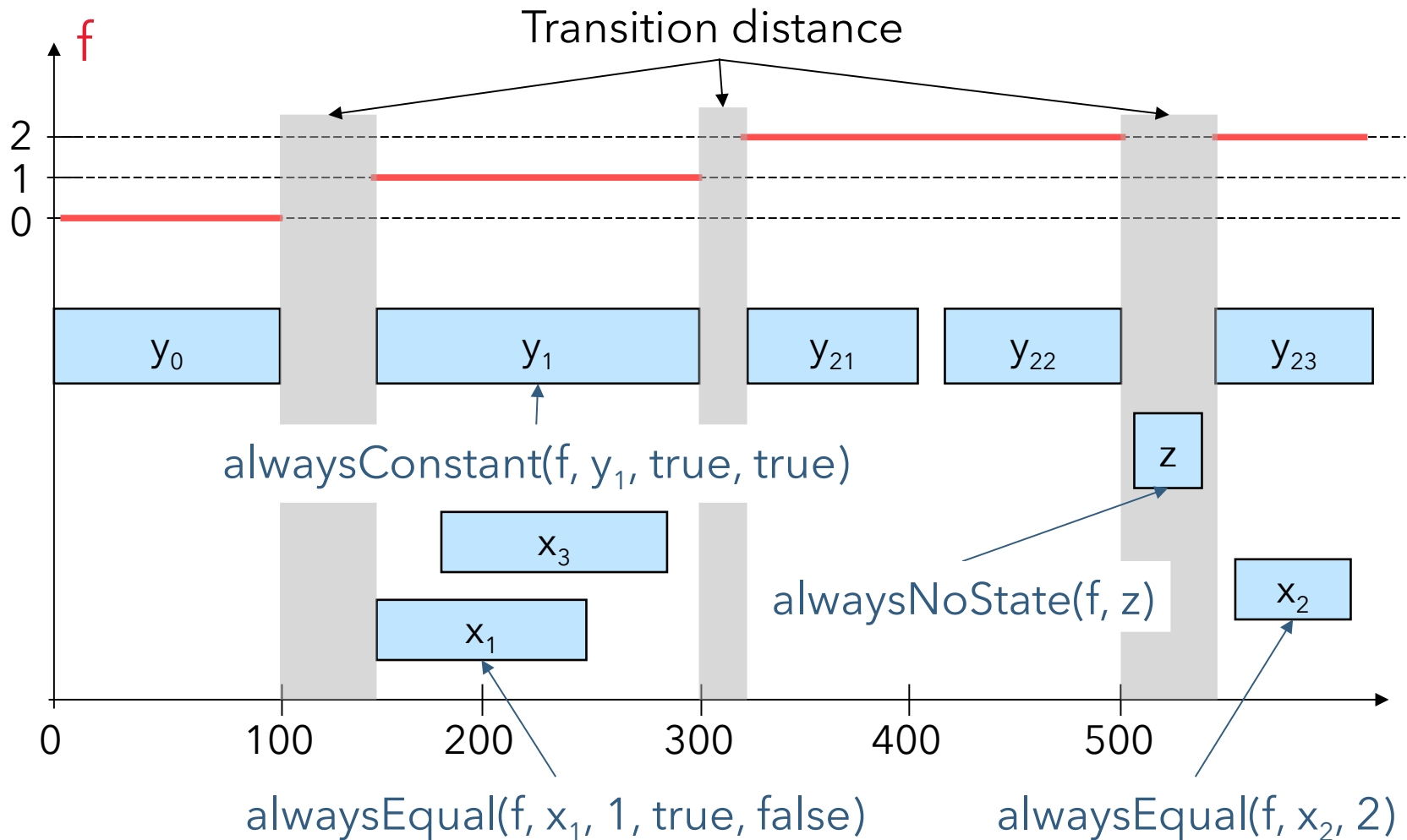
- Constraints on state functions
 - Over an interval variable x (if present), state function f , if defined is always in range $[vmin, vmax]$:

$\text{alwaysIn}(f, x, vmin, vmax)$



$\text{alwaysIn}(f, x, 1, 2)$

- Constraints on state functions



Modeling: State function variables

- Example: simplified semiconductor manufacturing machine

```
1 using CP;
2 int n = ...;
3 int capacity = ...;
4 int pt[1..n] = ...;
5 int nbwafers[1..n] = ...;
6 int family[1..n] = ...;
7 tuple triplet { int id1; int id2; int value; };
8 {triplet} M = ...; // Transition time between pairs of families
9
10 dvar interval op[i in 1..n] size pt[i];
11 stateFunction batch with M;
12 cumuFunction load = sum (i in 1..n) pulse(op[i], nbwafers[i]);
13
14 constraints {
15     load <= capacity;
16     forall(i in 1..n) {
17         alwaysEqual(batch, op[i], family[i], true, true);
18     }
19 }
```

Modeling: State function variables

- To summarize:

Variable: state function f [with M]

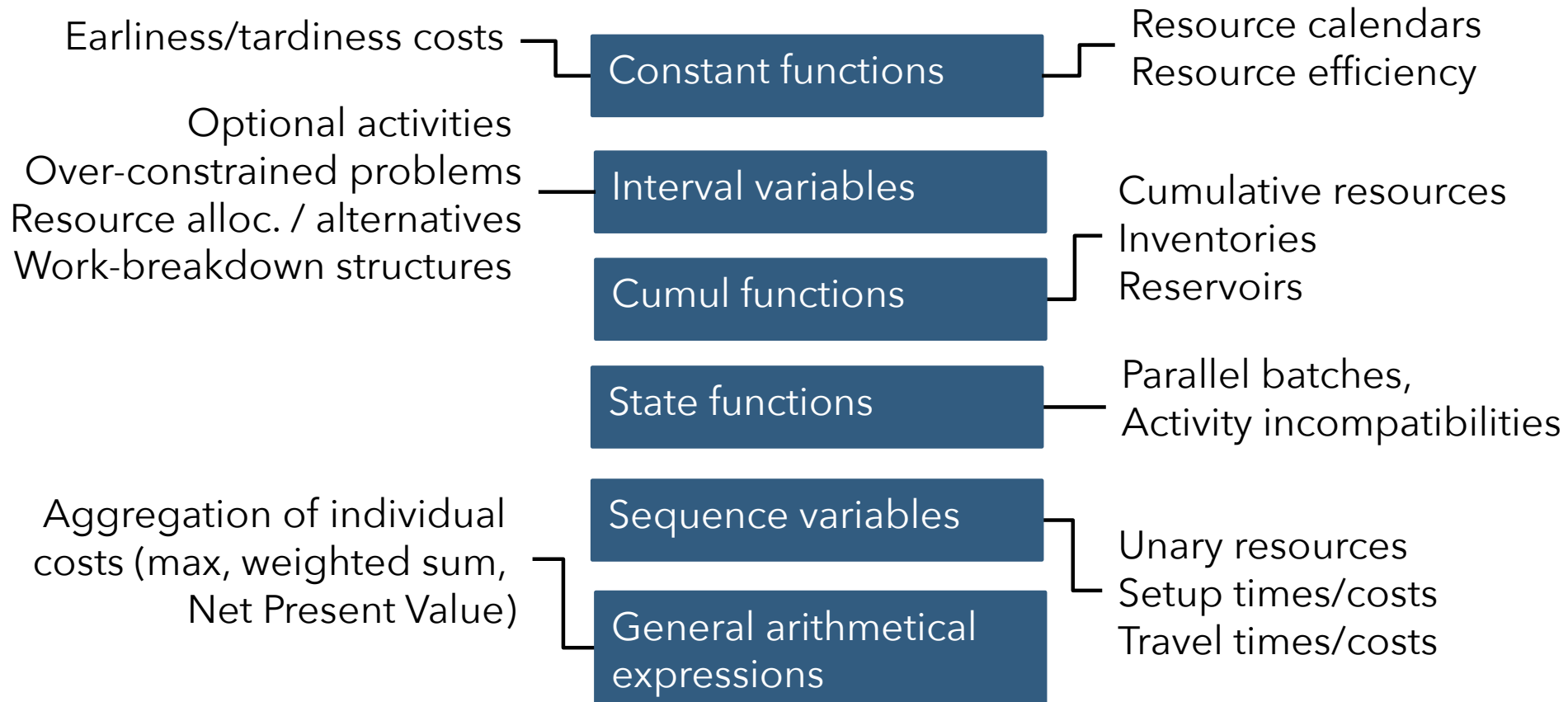
Constraints:

alwaysEqual(f, x, v, sal, eal)
alwaysConstant(f, x, sal, eal)
alwaysNoState(f, x)
alwaysIn($f, x, vmin, vmax$)

alwaysEqual(f, s, e, v, sal, eal)
alwaysConstant(f, s, e, sal, eal)
alwaysNoState(f, s, e)
alwaysIn($f, s, e, vmin, vmax$)

Modeling: Wrap-up

- CP Optimizer has mathematical concepts that naturally map to features invariably found in industrial scheduling problems



Modeling

- We have seen most of the scheduling concepts provided by CP Optimizer for modeling scheduling problems
- As in CP Optimizer “modeling is the only thing that matters” we could as well stop here ...



- But we will do a few additional laps ...

Overview

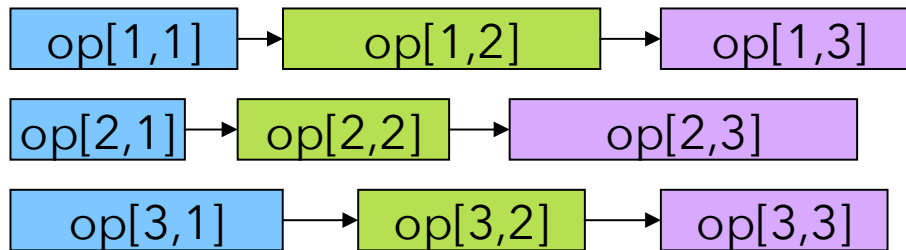
- Introduction
- Modeling
- Examples
- Performance
- Solving
- Tools
- Under the hood
- Conclusion

Examples

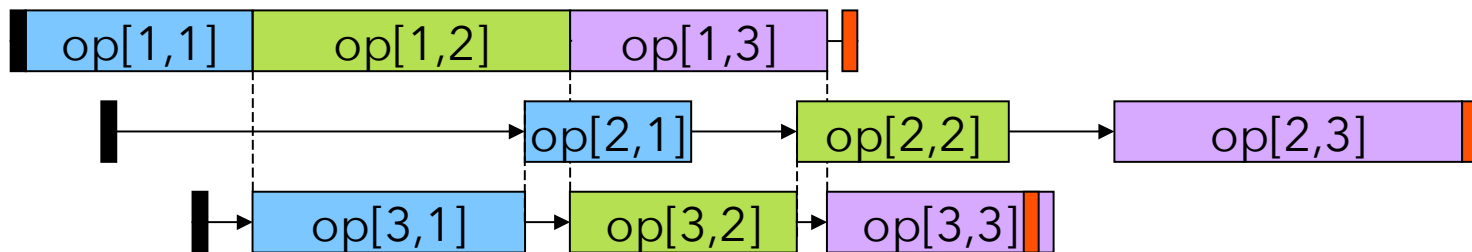
- Flow-shop with earliness and tardiness costs
- Multi-Mode RCPSP
- Satellite communication scheduling
- Semiconductor manufacturing scheduling
- GEO-CAPE Observation scheduling
 - Presentation at this ICAPS conference :
P. Laborie, B. Messaoudi. *New Results for the GEO-CAPE Observation Scheduling Problem*
 - Tomorrow, Session *Applications II*, 3:45-4:45 p.m.

Flow-shop with earliness and tardiness costs

- Classical Flow-Shop Scheduling problem:
 - n jobs, m machines



- Job release dates (■), due dates (■) and weight



Flow-shop with earliness and tardiness costs

```
1 using CP;
2 int n = ...;
3 int m = ...;
4 int rd[1..n] = ...;
5 int dd[1..n] = ...;
6 float w[1..n] = ...;
7 int pt[1..n][1..m] = ...;
8 float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);
9
10 dvar interval op[i in 1..n][j in 1..m] size pt[i][j];
11
12 dexpr int C[i in 1..n] = endOf(op[i][m]);
13
14 minimize sum(i in 1..n) w[i]*abs(C[i]-dd[i])/W;
15 subject to {
16     forall(i in 1..n) {
17         rd[i] <= startOf(op[i][1]);
18         forall(j in 1..m-1)
19             endBeforeStart(op[i][j],op[i][j+1]);
20     }
21     forall(j in 1..m)
22         noOverlap(all(i in 1..n) op[i][j]);
23 }
```

Flow-shop with earliness and tardiness costs

```
1 using CP;
2 int n = ...;
3 int m = ...;
4 int rd[1..n] = ...;
5 int dd[1..n] = ...;
6 float w[1..n] = ...;
7 int pt[1..n][1..m] = ...;
8 float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);
9
10 dvar interval op[i in 1..n][j in 1..m] size pt[i][j];
11
12 dexpr int C[i in 1..n] = endOf(op[i][m]);
13
14 minimize sum(i in 1..n) w[i]*abs(C[i]-dd[i])/W;
15 subject to {
16     forall(i in 1..n) {
17         rd[i] <= startOf(op[i][1]);
18         forall(j in 1..m-1)
19             endBeforeStart(op[i][j],op[i][j+1]);
20     }
21     forall(j in 1..m)
22         noOverlap(all(i in 1..n) op[i][j]);
23 }
```

Decision variables:

operations $op[i][j]$ modeled as interval variables of size $pt[i][j]$

Objective:

minimize weighted sum of deviation from due date of each job

Flow-shop with earliness and tardiness costs

```
1 using CP;
2 int n = ...;
3 int m = ...;
4 int rd[1..n] = ...;
5 int dd[1..n] = ...;
6 float w[1..n] = ...;
7 int pt[1..n][1..m] = ...;
8 float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);
9
10 dvar interval op[i in 1..n][j in 1..m] size pt[i][j];
11
12 dexpr int C[i in 1..n] = endOf(op[i][m]);
13
14 minimize sum(i in 1..n) w[i]*abs(C[i]-dd[i])/W;
15 subject to {
16     forall(i in 1..n) {
17         rd[i] <= startOf(op[i][1]);
18         forall(j in 1..m-1)
19             endBeforeStart(op[i][j],op[i][j+1]);
20     }
21     forall(j in 1..m)
22         noOverlap(all(i in 1..n) op[i][j]);
23 }
```

Decision variables:

operations $op[i][j]$ modeled as interval variables of size $pt[i][j]$

Constraints:

for each job i :

- release dates $rd[i]$
- precedence constraints

for each machine j :

- operations do not overlap on the machine

Flow-shop with earliness and tardiness costs

- Benchmark used in E. Danna and L. Perron. *Structured v.s. Unstructured Large Neighborhood Search: a Case Study on Job-shop Scheduling Problems with Earliness and Tardiness Costs*. In Proc. CP-2003.
- Comparison against state-of-the-art (CPAIOR-2009)

Problem	GA-best	S-LNS-best	<i>CPO</i>	Problem	GA-best	S-LNS-best	<i>CPO</i>
jb1	0.474	0.191	<i>0.191</i>	ljb1	0.279	0.215	<i>0.215</i>
jb2	0.499	0.137	<i>0.137</i>	ljb2	0.598	0.508	<i>0.509</i>
jb4	0.619	0.568	<i>0.568</i>	ljb7	0.246	0.110	<i>0.137</i>
jb9	0.369	0.333	<i>0.334</i>	ljb9	0.739	1.015	<i>0.744</i>
jb11	0.262	0.213	<i>0.213</i>	ljb10	0.512	0.525	<i>0.549</i>
jb12	0.246	0.190	<i>0.190</i>	ljb12	0.399	0.605	<i>0.518</i>

Table 1. Results for Flow-shop Scheduling with Earliness and Tardiness Costs

- Average improvement
 - Against GA: 25%
 - Against specific LNS: 1.7%

Flow-shop with earliness and tardiness costs

- Benchmark used in E. Danna and L. Perron. *Structured v.s. Unstructured Large Neighborhood Search: a Case Study on Job-shop Scheduling Problems with Earliness and Tardiness Costs*. In Proc. CP-2003.
- Comparison against state-of-the-art (CPAIOR-2009)

Problem	GA-best	S-LNS-best	<i>CPO</i>	Problem	GA-best	S-LNS-best	<i>CPO</i>
jb1	0.474	0.191	<i>0.191</i>	★jb1	0.279	0.215	<i>0.215</i> ★
jb2	0.499	0.137	<i>0.137</i>	★jb2	0.598	0.508	<i>0.509</i> 0.508
jb4	0.619	0.568	<i>0.568</i>	★jb7	0.246	0.110	<i>0.137</i> 0.102
jb9	0.369	0.333	<i>0.334</i>	★jb9	0.739	1.015	<i>0.744</i> 0.695
jb11	0.262	0.213	<i>0.213</i>	★jb10	0.512	0.525	<i>0.549</i> 0.437
jb12	0.246	0.190	<i>0.190</i>	★jb12	0.399	0.605	<i>0.518</i> 0.387

Table 1. Results for Flow-shop Scheduling with Earliness and Tardiness Costs

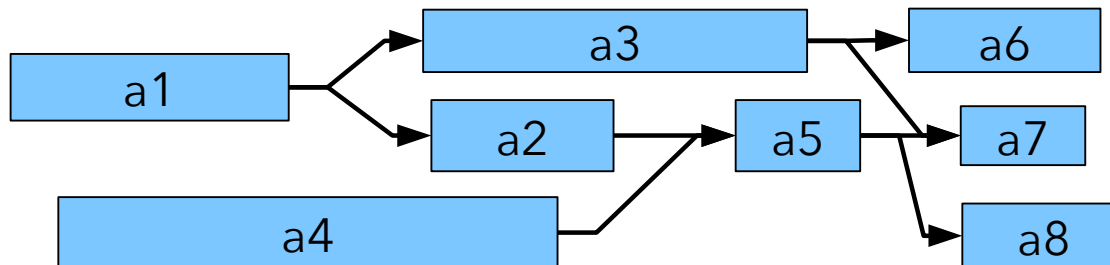
- Average improvement

- Against GA: 25% **31%**
- Against specific LNS: 1.7% **8.6%**

Updated results (V12.7.1)
★ : optimality proof

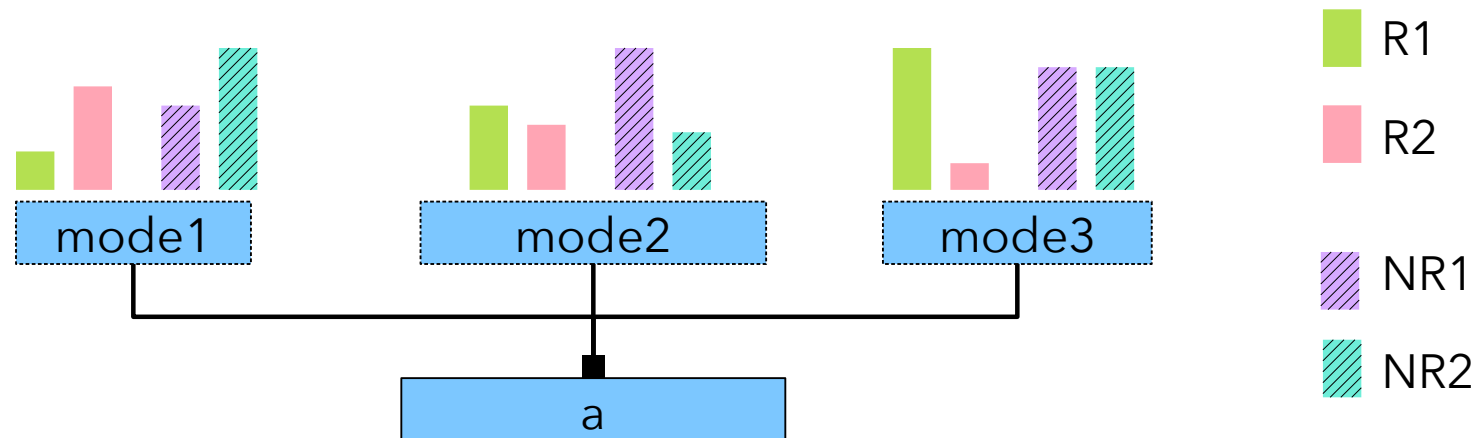
Multi-Mode RCPSP

- Extension of RCPSP:
 - A task uses both renewable and **non-renewable** resources
 - Each task can be performed according to several **alternative modes**
 - A mode specifies the task duration and some particular resource usage



Multi-Mode RCPSP

- Extension of RCPSP:
 - A task uses both renewable and **non-renewable** resources
 - Each task can be performed according to several **alternative modes**
 - A mode specifies the task duration and some particular resource usage



Multi-Mode RCPSP

```
1 using CP;
2 int NbTasks = ...;
3 int NrRs = ...;
4 int NbNs = ...;
5 int CapR[1..NrRs] = ...;
6 int CapN[1..NbNs] = ...;
7
8 tuple Task { key int id; {int} succs; }
9 {Task} Tasks = ...;
10
11 tuple Mode {
12     int tid;
13     int pt;
14     int dmdR [1..NrRs];
15     int dmdN [1..NbNs];
16 }
17 {Mode} Modes = ...;
18
19 dvar interval task[t in Tasks];
20 dvar interval mode[m in Modes] optional size m.pt;
21
22 cumulFunction rUsage[r in 1..NrRs] =
23     sum (m in Modes: m.dmdR[r]>0) pulse(mode[m], m.dmdR[r]);
24 dexpr int nUsage[r in 1..NbNs] =
25     sum (m in Modes: m.dmdN[r]>0) m.dmdN[r] * presenceOf(mode[m]);
26
27 minimize max(t in Tasks) endOf(task[t]);
28 subject to {
29     forall (t in Tasks)
30         alternative(task[t], all(m in Modes: m.tid==t.id) mode[m]);
31     forall (r in 1..NrRs)
32         rUsage[r] <= CapR[r];
33     forall (r in 1..NbNs)
34         nUsage[r] <= CapN[r];
35     forall (t1 in Tasks, t2id in t1.succs)
36         endBeforeStart(task[t1], task[<t2id>]);
37 }
```

Multi-Mode RCPSP

```
1 using CP;
2 int NbTasks = ...;
3 int NrRs = ...;
4 int NbNs = ...;
5 int CapR[1..NrRs] = ...;
6 int CapN[1..NbNs] = ...;
7
8 tuple Task { key int id; {int} succs; }
9 {Task} Tasks = ...;
10
11 tuple Mode {
12     int tid;
13     int pt;
14     int dmdR [1..NrRs];
15     int dmdN [1..NbNs];
16 }
17 {Mode} Modes = ...;
18
19 dvar interval task[t in Tasks];
20 dvar interval mode[m in Modes] optional size m.pt;
21
22 cumulFunction rUsage[r in 1..NrRs] =
23     sum (m in Modes: m.dmdR[r]>0) pulse(mode[m], m.dmdR[r]);
24 dexpr int nUsage[r in 1..NbNs] =
25     sum (m in Modes: m.dmdN[r]>0) m.dmdN[r] * presenceOf(mode[m]);
26
27 minimize max(t in Tasks) endOf(task[t]);
28 subject to {
29     forall (t in Tasks)
30         alternative(task[t], all(m in Modes: m.tid==t.id) mode[m]);
31     forall (r in 1..NrRs)
32         rUsage[r] <= CapR[r];
33     forall (r in 1..NbNs)
34         nUsage[r] <= CapN[r];
35     forall (t1 in Tasks, t2id in t1.succs)
36         endBeforeStart(task[t1], task[<t2id>]);
37 }
```

Data reading:

tasks data:

task id

successors

modes data:

task id

processing time

renewable resources usage

non-renewable resources usage

Multi-Mode RCPSP

```
1 using CP;
2 int NbTasks = ...;
3 int NrRs = ...;
4 int NbNs = ...;
5 int CapR[1..NrRs] = ...;
6 int CapN[1..NbNs] = ...;
7
8 tuple Task { key int id; {int} succs; }
9 {Task} Tasks = ...;
10
11 tuple Mode {
12     int tid;
13     int pt;
14     int dmdR [1..NrRs];
15     int dmdN [1..NbNs];
16 }
17 {Mode} Modes = ...;
18
19 dvar interval task[t in Tasks];
20 dvar interval mode[m in Modes] optional size m.pt;
21
22 cumulFunction rUsage[r in 1..NrRs] =
23     sum (m in Modes: m.dmdR[r]>0) pulse(mode[m], m.dmdR[r]);
24 dexpr int nUsage[r in 1..NbNs] =
25     sum (m in Modes: m.dmdN[r]>0) m.dmdN[r] * presenceOf(mode[m]);
26
27 minimize max(t in Tasks) endOf(task[t]);
28 subject to {
29     forall (t in Tasks)
30         alternative(task[t], all(m in Modes: m.tid==t.id) mode[m]);
31     forall (r in 1..NrRs)
32         rUsage[r] <= CapR[r];
33     forall (r in 1..NbNs)
34         nUsage[r] <= CapN[r];
35     forall (t1 in Tasks, t2id in t1.succs)
36         endBeforeStart(task[t1], task[<t2id>]);
37 }
```

Decision variables:

$\text{task}[t]$: task intervals

$\text{mode}[m]$: mode intervals
(optional)

Objective:

minimize makespan

Multi-Mode RCPSP

```
1 using CP;
2 int NbTasks = ...;
3 int NrRs = ...;
4 int NbNs = ...;
5 int CapR[1..NrRs] = ...;
6 int CapN[1..NbNs] = ...;
7
8 tuple Task { key int id; {int} succs; }
9 {Task} Tasks = ...;
10
11 tuple Mode {
12     int tid;
13     int pt;
14     int dmdR [1..NrRs];
15     int dmdN [1..NbNs];
16 }
17 {Mode} Modes = ...;
18
19 dvar interval task[t in Tasks];
20 dvar interval mode[m in Modes] optional size m.pt;
21
22 cumulFunction rUsage[r in 1..NrRs] =
23     sum (m in Modes: m.dmdR[r]>0) pulse(mode[m], m.dmdR[r]);
24 dexpr int nUsage[r in 1..NbNs] =
25     sum (m in Modes: m.dmdN[r]>0) m.dmdN[r] * presenceOf(mode[m]);
26
27 minimize max(t in Tasks) endOf(task[t]);
28 subject to {
29     forall (t in Tasks)
30         alternative(task[t], all(m in Modes: m.tid==t.id) mode[m]);
31     forall (r in 1..NrRs)
32         rUsage[r] <= CapR[r];
33     forall (r in 1..NbNs)
34         nUsage[r] <= CapN[r];
35     forall (t1 in Tasks, t2id in t1.succs)
36         endBeforeStart(task[t1], task[<t2id>]);
37 }
```

Decision variables:

$\text{task}[t]$: task intervals

$\text{mode}[m]$: mode intervals
(optional)

Constraints:

- alternative modes for a task
- precedence constraints

Multi-Mode RCPSP

```
1 using CP;
2 int NbTasks = ...;
3 int NrRs = ...;
4 int NbNs = ...;
5 int CapR[1..NrRs] = ...;
6 int CapN[1..NbNs] = ...;
7
8 tuple Task { key int id; {int} succs; }
9 {Task} Tasks = ...;
10
11 tuple Mode {
12     int tid;
13     int pt;
14     int dmdR [1..NrRs];
15     int dmdN [1..NbNs];
16 }
17 {Mode} Modes = ...;
18
19 dvar interval task[t in Tasks];
20 dvar interval mode[m in Modes] optional size m.pt;
21
22 cumulFunction rUsage[r in 1..NrRs] =
23     sum (m in Modes: m.dmdR[r]>0) pulse(mode[m], m.dmdR[r]);
24 dexpr int nUsage[r in 1..NbNs] =
25     sum (m in Modes: m.dmdN[r]>0) m.dmdN[r] * presenceOf(mode[m]);
26
27 minimize max(t in Tasks) endOf(task[t]);
28 subject to {
29     forall (t in Tasks)
30         alternative(task[t], all(m in Modes: m.tid==t.id) mode[m]);
31     forall (r in 1..NrRs)
32         rUsage[r] <= CapR[r];
33     forall (r in 1..NbNs)
34         nUsage[r] <= CapN[r];
35     forall (t1 in Tasks, t2id in t1.succs)
36         endBeforeStart(task[t1], task[<t2id>]);
37 }
```

Renewable resource usage expressions

Non-renewable resource usage expressions

Constraints:

- renewable res. capacity
- non-renewable res. capacity

Multi-Mode RCPSP

```
1 using CP;
2 int NbTasks = ...;
3 int NrRs = ...;
4 int NbNs = ...;
5 int CapR[1..NrRs] = ...;
6 int CapN[1..NbNs] = ...;
7
8 tuple Task { key int id; {int} succs; }
9 {Task} Tasks = ...;
10
11 tuple Mode {
12     int tid;
13     int pt;
14     int dmdR [1..NrRs];
15     int dmdN [1..NbNs];
16 }
17 {Mode} Modes = ...;
18
19 dvar interval task[t in Tasks];
20 dvar interval mode[m in Modes] optional size m.pt;
21
22 cumulFunction rUsage[r in 1..NrRs] =
23     sum (m in Modes: m.dmdR[r]>0) pulse(mode[m], m.dmdR[r]);
24 dexpr int nUsage[r in 1..NbNs] =
25     sum (m in Modes: m.dmdN[r]>0) m.dmdN[r] * presenceOf(mode[m]);
26
27 minimize max(t in Tasks) endOf(task[t]);
28 subject to {
29     forall (t in Tasks)
30         alternative(task[t], all(m in Modes: m.tid==t.id) mode[m]);
31     forall (r in 1..NrRs)
32         rUsage[r] <= CapR[r];
33     forall (r in 1..NbNs)
34         nUsage[r] <= CapN[r];
35     forall (t1 in Tasks, t2id in t1.succs)
36         endBeforeStart(task[t1], task[<t2id>]);
37 }
```

Note the similar formulation of (atemporal) non-renewable resources and (temporal) renewable resources

Renewable resource usage expressions

Non-renewable resource usage expressions

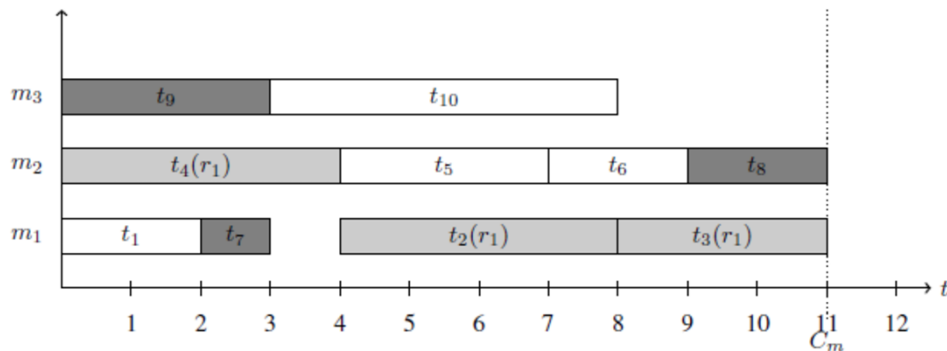
Constraints:

- renewable res. capacity
- non-renewable res. capacity

Multi-Mode RCPSP

- CP Optimizer won the CP-2015 Industrial Modelling Competition with a very similar model

Test	Duration	Executable on	Use of global resource
t_1	2	m_1, m_2, m_3	-
t_2	4	m_1, m_2, m_3	r_1
t_3	3	m_1, m_2, m_3	r_1
t_4	4	m_1, m_2, m_3	r_1
t_5	3	m_1, m_2, m_3	-
t_6	2	m_1, m_2, m_3	-
t_7	1	m_1	-
t_8	2	m_2	-
t_9	3	m_3	-
t_{10}	5	m_1, m_3	-

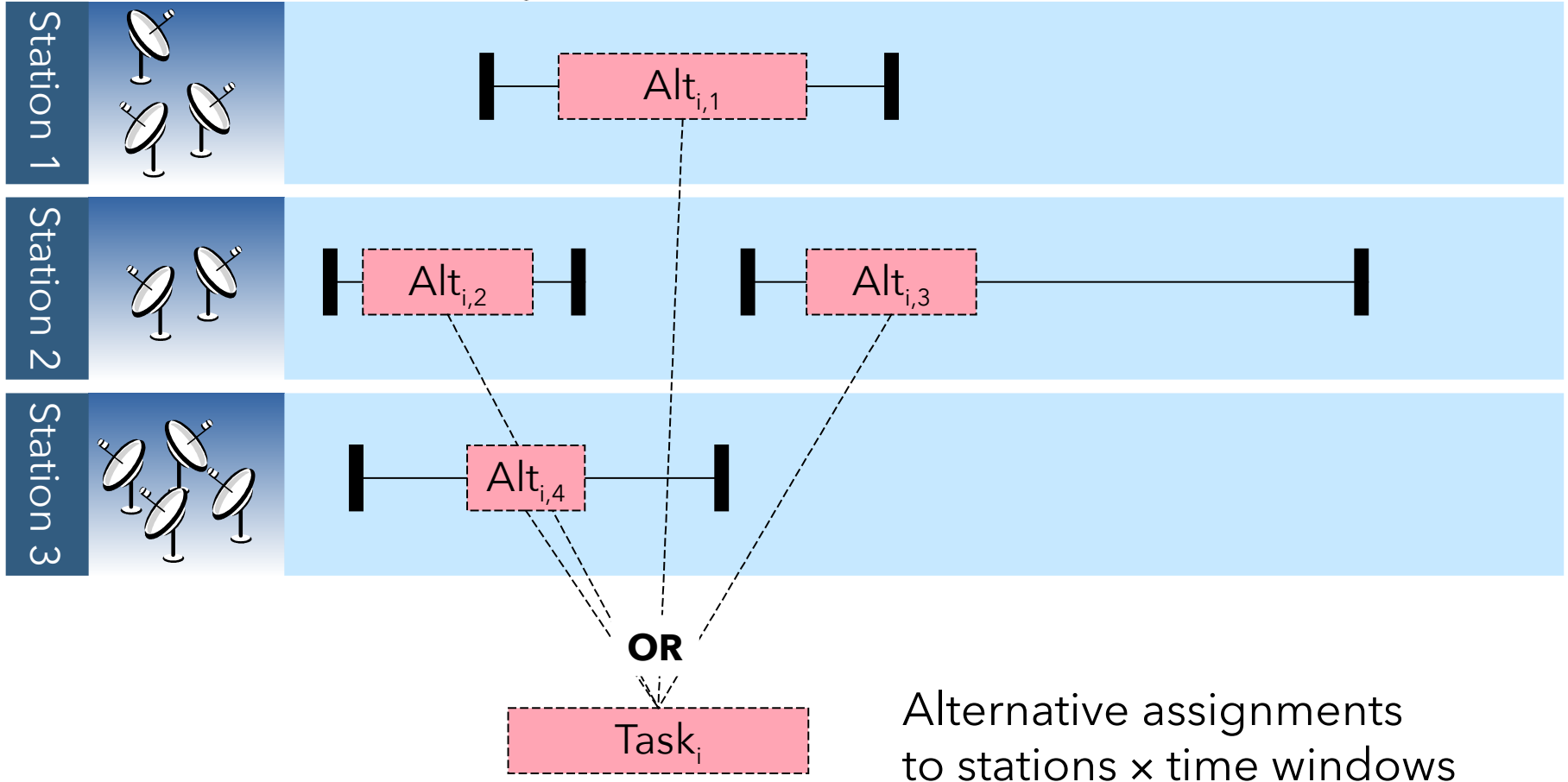
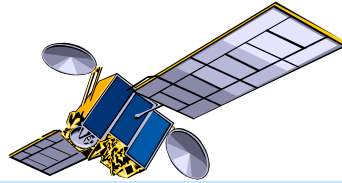


Instance	LB	UB	OPTIMAL
t40m10r3-2.pl	1725	1725	*
t50m10r3-9.pl	7279	7279	*
t100m50r10-11.pl	4970	4970	*
t500m50r5-5.pl	33848	33848	*
t500m100r10-1.pl	48814	48814	*
t500m100r10-2.pl	38303	39130	
t500m100r10-3.pl	35459	36018	
t500m100r10-4.pl	37658	37921	
t500m100r10-5.pl	33080	33338	
t500m100r10-6.pl	41078	41078	*
t500m100r10-7.pl	38921	39790	
t500m100r10-8.pl	42455	43199	
t500m100r10-9.pl	38758	39083	
t500m100r10-10.pl	42308	42308	*

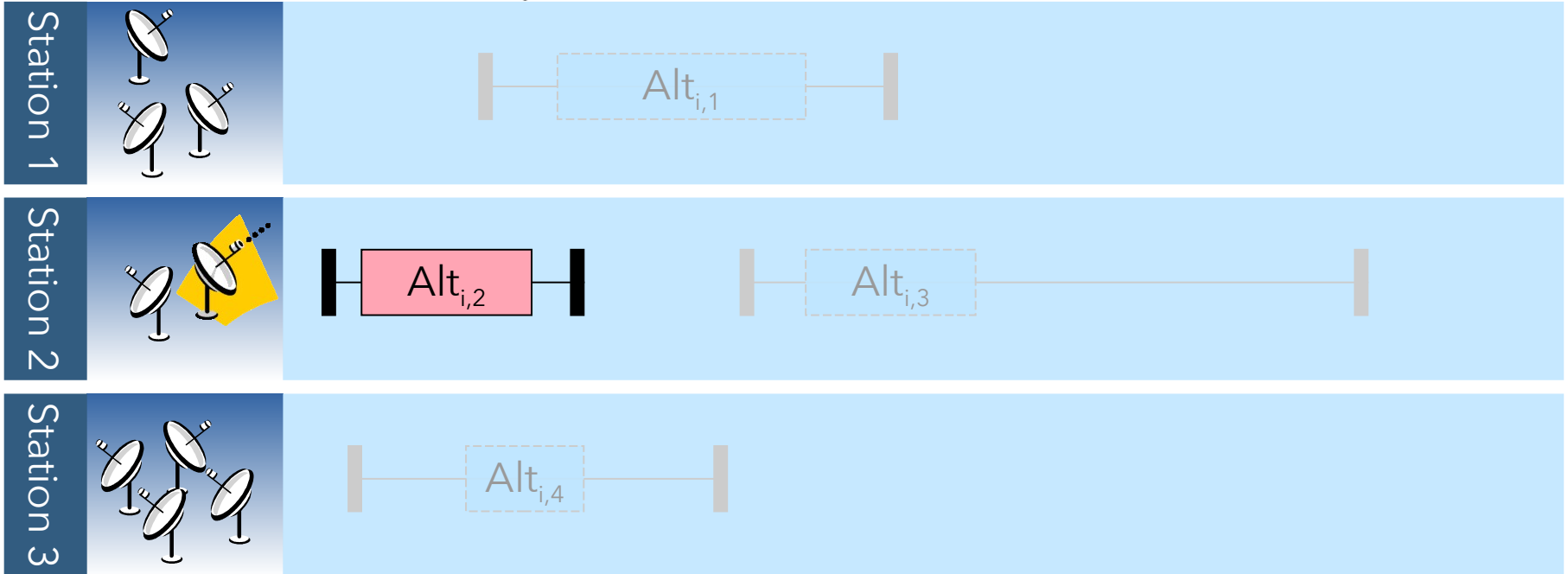
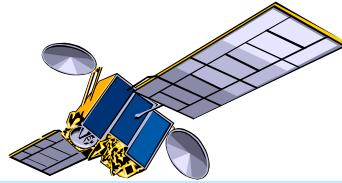
Satellite communication scheduling

- USAF Satellite Control Network scheduling problem described in L. Kramer, L. Barbulescu and S. Smith. *Understanding Performance Trade-offs in Algorithms for Solving Oversubscribed Scheduling*. In Proc. AAAI-2007.
- A set of n communication requests for Earth orbiting satellites must be scheduled on a total of 32 antennas spread across 13 ground-based tracking stations.
- Different priority level for requests
- Objective is to maximize the number of satisfied requests, starting with highest priority requests (lexicographical)
- In the instances, n ranges from 400 to 1300

Satellite communication scheduling



Satellite communication scheduling



Selected alternative will use
1 antenna for communication
with the satellite

Satellite communication scheduling

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

Satellite communication scheduling

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

Data reading:
resources (stations)
tasks
alternatives

Satellite communication scheduling

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

Decision variables:

task[t]: task intervals
(optional)

alt[a]: alternative
intervals (optional)

Objective:

maximize number of executed
tasks by lexicographical order
of priority

Satellite communication scheduling

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

Decision variables:

task[t]: task intervals
(optional)

alt[a]: alternative
intervals (optional)

Constraints:

each task (if executed)
must be allocated one of
its alternatives

Satellite communication scheduling

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

Decision variables:

task[t]: task intervals
(optional)

alt[a]: alternative
intervals (optional)

Constraints:

capacity of resources
(number of antennas)
must never be exceeded

Satellite communication scheduling

- Experimental results (CPAIOR-2009)

Problem set	TS	SWO	<i>CPO</i>	Problem set	TS	SWO	<i>CPO</i>
1.1	30.44	26.60	<i>27.50</i>	4.1	3.20	2.00	<i>1.96</i>
1.2	114.02	104.72	<i>98.10</i>	4.2	13.34	7.90	<i>7.48</i>
1.3	87.92	84.52	<i>86.04</i>	4.3	16.60	12.46	<i>9.68</i>
2.1	11.46	7.80	<i>7.84</i>	5.1	3.90	3.80	<i>3.76</i>
2.2	45.54	34.26	<i>30.64</i>	5.2	32.98	31.98	<i>31.72</i>
2.3	33.96	31.18	<i>32.14</i>	5.3	46.18	45.22	<i>44.34</i>
3.1	2.64	2.32	<i>2.28</i>	6.1	1.56	1.28	<i>1.24</i>
3.2	15.50	12.82	<i>11.82</i>	6.2	11.62	9.56	<i>8.92</i>
3.3	32.10	28.58	<i>24.00</i>	6.3	25.28	22.60	<i>19.48</i>

Table 2. Results for Satellite Scheduling

- In average, compared with SWO, the number of unscheduled tasks is decreased by 5.3%

Satellite communication scheduling

- Taste of the different APIs

C++

OPL

Python

Java

C#

```
dvar interval task[t in Tasks] optional;
dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;

dexpr int nb[p in Priorities] =
    sum(t in Tasks: t.prio==p) presenceOf(task[t]);

maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
subject to {
    forall(t in Tasks)
        alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
    forall(r in Res)
        sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
}
```

Satellite communication scheduling

- Taste of the different APIs

C++

OPL

Python

Java

C#

```
# Create environment and model
IloEnv env;
IloModel model(env);
# Create array of cumul function expressions for resources
IloCumulFunctionExprArray load(env, nbRes);
for (int k=0; k<nbRes; k++)
    load[k] = IloCumulFunctionExpr(env);
# Create array of number of present tasks per priority level
IloIntExprArray nb(env, 5);
for (int p=0; p<5; p++)
    nb[p] = IloIntExpr(env);
# Create array of interval variables for alternatives
IloIntervalVarArray alt(env, nbAlts);
for (int j=0; j<nbAlts; j++) {
    alt[j]= IloIntervalVar(env, Alts[j].dur);
    alt[j].setOptional();
    alt[j].setStartMin(Alts[j].smin);
    alt[j].setEndMax(Alts[j].emax);
    load[Alts[j].res] += IloPulse(alt[j],1);
}
# Create array of interval variables for tasks
IloIntervalVarArray task(env, nbTasks);
for (int i=0; i<nbTasks; i++) {
    task[i]= IloIntervalVar(env, Tasks[i].name);
    task[i].setOptional();
    nb[Tasks[i].prio] += IloPresenceOf(env, task[i]);
    IloIntervalVarArray altsi(env);
    for (int j=0; j<nbAlts; j++) {
        if (Alts[j].task == i)
            altsi.add(alt[j]);
    }
    model.add(IloAlternative(env, task[i], altsi));
}
# Lexicographical objective
model.add(IloMaximize(env, IloStaticLex(env, nb)));
# Limited capacity of resources
for (int k=0; k<nbRes; k++)
    model.add(load[k] <= Res[k].cap);
# Solve the model
IloCP cp(model);
cp.setParameter(IloCP::TimeLimit, 10);
cp.solve();
```

Satellite communication scheduling

- Taste of the different APIs

C++

OPL

Python

Java

C#

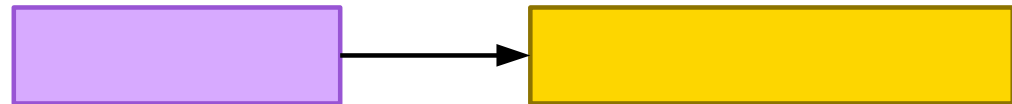
```
# Import CP Optimizer modelization functions
from docplex.cp.model import *
# Create model object
model = CpoModel()
# Create dictionary of interval variables for tasks
task = dict()
for t in Tasks:
    task[t]= interval_var(optional=True,name=t.name)
# Create dictionary of interval variables for alternatives
alt = dict()
for a in Alts:
    alt[a]= interval_var(optional=True,size=a.dur,start=(a.smin,a.emax-a.dur))
# Number of present tasks of a given priority
nb = [sum([presence_of(task[t]) for t in Tasks if t.prio==p]) for p in range(1,5)]
# Lexicographical objective
model.add(maximize_static_lex(nb))
# Selection of an alternative
for t in Tasks:
    model.add(alternative(task[t], [alt[a] for a in Alts if a.task==t.id]))
# Limited capacity of resources
load = [pulse(0,0,0) for r in Res]
for r in Res:
    for a in Alts:
        if a.res==r.id:
            load[r.id] += pulse(alt[a],1)
    model.add(load[r.id] <= r.cap)
# Solve the model
solution = model.solve(TimeLimit=10)
```

Semiconductor manufacturing scheduling

- This is the introductory problem

Wafer lots
(size, priority,
release date,
due date)

Each step has to be scheduled on a machine,
Possible machines for a given step depend on its
family (here, its color)



Semiconductor manufacturing scheduling

- This is the introductory problem

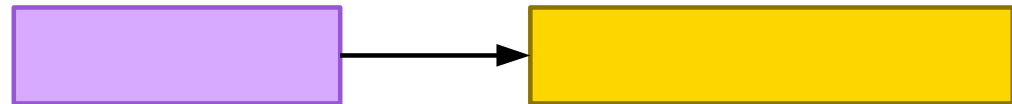
Wafer lots
(size, priority,
release date,
due date)

A machine specifies the step families it can process
and a family-dependent processing time

Example:

M1 can process  and 

M2 can process  and 

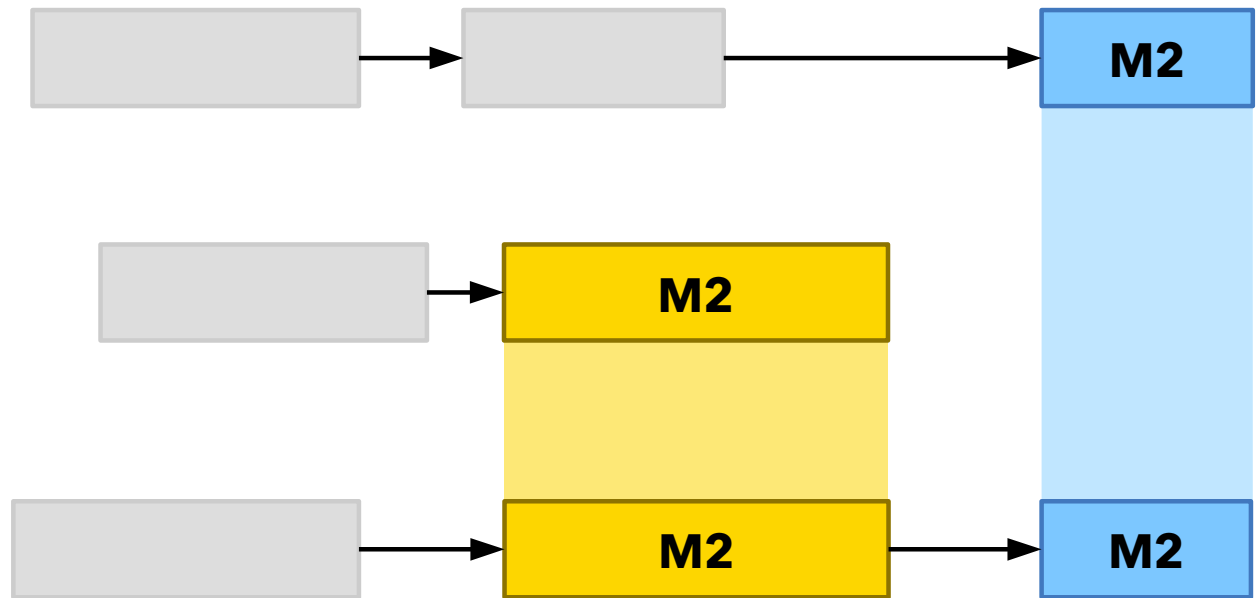


Semiconductor manufacturing scheduling

- This is the introductory problem

Wafer lots
(size, priority,
release date,
due date)

Steps of the same family can be processed together
on the same machine (**batch**)
Batched steps **start** and **end** at the same time
Batching **capacity**: max. number of wafers

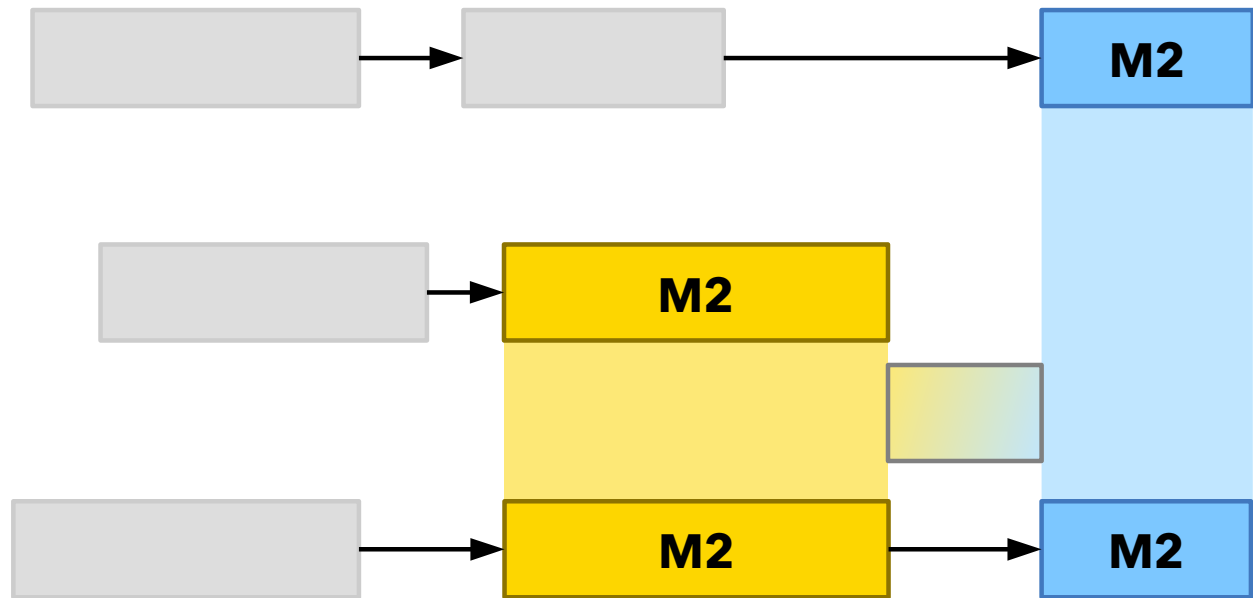


Semiconductor manufacturing scheduling

- This is the introductory problem

Wafer lots
(size, priority,
release date,
due date)

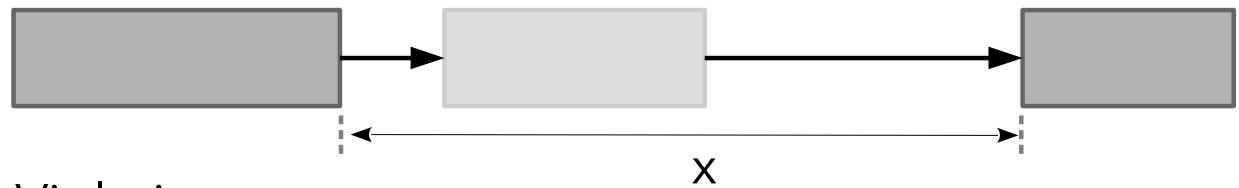
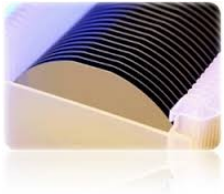
Sequence-dependent setup times on machines
Waiting time, e.g. required by temperature changes
Duration depends on family of adjacent steps



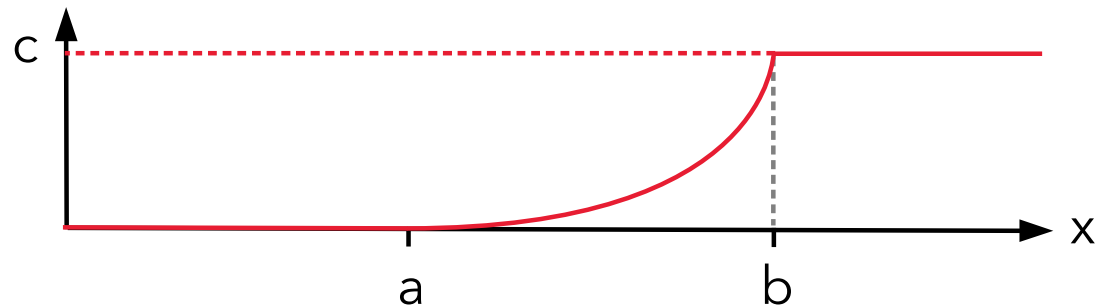
Semiconductor manufacturing scheduling

- This is the introductory problem

Physical and chemical properties imposes some **maximum time lags** between some pairs of lot steps
This is a soft constraint with a violation cost



Violation cost



$$V = \min(c, c \cdot \max(0, x-a)^2 / (b-a)^2)$$

Semiconductor manufacturing scheduling

- This is the introductory problem
- Lexicographical objective function:
 - Criterion 1: minimize total violation of maximum time lags
 - Criterion 2: minimize weighted tardiness cost of lots

The Promise:

After this tutorial, you will be able to solve it with a 50 lines long model achieving excellent performance

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c,m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27     sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30     sum(d in Lags) minl(d.c, d.c*maxl(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31     sum(l in Lots) l.w*maxl(0, endOf(lot[l])-l.dd));
32 subject to {
33     forall(l in Lots)
34         span(lot[l], all(s in Stps: s.l==l) stp[s]);
35     forall(s in Stps) {
36         alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37         if (s.pos>1)
38             endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39     }
40     forall(ms in MchStps)
41         alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42     forall(m in Mchs)
43         load[m] <= m.capacity;
44     forall(d in Lags)
45         endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c.m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27   sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30   sum(d in Lags) minl(d.c, d.c*maxl(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31   sum(l in Lots) l.w*maxl(0, endOf(lot[l])-l.dd));
32 subject to {
33   forall(l in Lots)
34     span(lot[l], all(s in Stps: s.l==l) stp[s]);
35   forall(s in Stps) {
36     alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37     if (s.pos>1)
38       endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39   }
40   forall(ms in MchStps)
41     alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42   forall(m in Mchs)
43     load[m] <= m.capacity;
44   forall(d in Lags)
45     endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Data reading:

lots
steps
time lags
machines
setup times

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c,m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27   sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30   sum(d in Lags) minl(d.c, d.c*maxl(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31   sum(l in Lots) l.w*maxl(0, endOf(lot[l])-l.dd));
32 subject to {
33   forall(l in Lots)
34     span(lot[l], all(s in Stps: s.l==l) stp[s]);
35   forall(s in Stps) {
36     alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37     if (s.pos>1)
38       endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39   }
40   forall(ms in MchStps)
41     alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42   forall(m in Mchs)
43     load[m] <= m.capacity;
44   forall(d in Lags)
45     endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Decision variables:

- lot[l] : lot interval variable
- stp[s] : step interval variable
- mchStp[ms] : step on a given machine (optional)
- lag[d] : time lag (integer variable)

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c,m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27   sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30   sum(d in Lags) min1(d.c, d.c*max1(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31   sum(l in Lots) l.w*max1(0, endOf(lot[l])-l.dd));
32 subject to {
33   forall(l in Lots)
34     span(lot[l], all(s in Stps: s.l==l) stp[s]);
35   forall(s in Stps) {
36     alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37     if (s.pos>1)
38       endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39   }
40   forall(ms in MchStps)
41     alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42   forall(m in Mchs)
43     load[m] <= m.capacity;
44   forall(d in Lags)
45     endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Decision variables:

lot[l] : lot interval variable
stp[s] : step interval variable
mchStp[ms] : step on a given machine (optional)
lag[d] : time lag (integer variable)

Lexicographical objective:

1. time lag violation
2. weighted tardiness

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c,m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27   sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30   sum(d in Lags) minl(d.c, d.c*maxl(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31   sum(l in Lots) l.w*maxl(0, endOf(lot[l])-l.dd));
32 subject to {
33   forall(l in Lots)
34     span(lot[l], all(s in Stps: s.l==l) stp[s]);
35   forall(s in Stps) {
36     alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37     if (s.pos>1)
38       endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39   }
40   forall(ms in MchStps)
41     alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42   forall(m in Mchs)
43     load[m] <= m.capacity;
44   forall(d in Lags)
45     endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Decision variables:

- `lot[l]` : lot interval variable
- `stp[s]` : step interval variable
- `mchStp[ms]` : step on a given machine (optional)
- `lag[d]` : time lag (integer variable)

Constraints:

- each lot spans its steps
- precedence constraints
- time lags

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c,m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27   sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30   sum(d in Lags) minl(d.c, d.c*maxl(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31   sum(l in Lots) l.w*maxl(0, endOf(lot[l])-l.dd));
32 subject to {
33   forall(l in Lots)
34     span(lot[l], all(s in Stps: s.l==l) stp[s]);
35   forall(s in Stps) {
36     alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37     if (s.pos>1)
38       endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39   }
40   forall(ms in MchStps)
41     alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42   forall(m in Mchs)
43     load[m] <= m.capacity;
44   forall(d in Lags)
45     endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Decision variables:

lot[l] : lot interval variable
stp[s] : step interval variable
mchStp[ms] : step on a given machine (optional)
lag[d] : time lag (integer variable)

Constraints:

- alternative machine for a step

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c,m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27   sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30   sum(d in Lags) minl(d.c, d.c*maxl(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31   sum(l in Lots) l.w*maxl(0, endOf(lot[l])-l.dd));
32 subject to {
33   forall(l in Lots)
34     span(lot[l], all(s in Stps: s.l==l) stp[s]);
35   forall(s in Stps) {
36     alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37     if (s.pos>1)
38       endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39   }
40   forall(ms in MchStps)
41     alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42   forall(m in Mchs)
43     load[m] <= m.capacity;
44   forall(d in Lags)
45     endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Decision variables:

$mchStp[ms]$: step on a given machine (optional)

$batch[m]$: state function describing the family processed by machine m

Constraints:

- machine batches with steps of a given family

Semiconductor manufacturing scheduling

```
1 using CP;
2 tuple Lot { key int id; int n; float w; int rd; int dd; }
3 tuple Stp { key Lot l; key int pos; int f; }
4 tuple Lag { Lot l; int pos1; int pos2; int a; int b; float c; }
5 tuple Mch { key int id; int capacity; }
6 tuple MchFml { Mch m; int f; int pt; }
7 tuple MchStp { Mch m; Stp s; int pt; }
8 tuple Setup { int f1; int f2; int dur; }
9
10 {Lot} Lots = ...;
11 {Stp} Stps = ...;
12 {Lag} Lags = ...;
13 {Mch} Mchs = ...;
14 {MchFml} MchFmls = ...;
15 {Setup} MchSetups[m in Mchs] = ...;
16
17 {MchStp} MchStps = {<c.m,s,c.pt> | s in Stps, c in MchFmls: c.f==s.f};
18
19 dvar interval lot[l in Lots] in l.rd..48*60;
20 dvar interval stp[s in Stps];
21 dvar interval mchStp[ms in MchStps] optional size ms.pt;
22
23 dvar int lag[Lags];
24
25 stateFunction batch[m in Mchs] with MchSetups[m];
26 cumulFunction load [m in Mchs] =
27     sum(ms in MchStps: ms.m==m) pulse(mchStp[ms],ms.s.l.n);
28
29 minimize staticLex(
30     sum(d in Lags) minl(d.c, d.c*maxl(0,lag[d]-d.a)^2/(d.b-d.a)^2),
31     sum(l in Lots) l.w*maxl(0, endOf(lot[l])-l.dd));
32 subject to {
33     forall(l in Lots)
34         span(lot[l], all(s in Stps: s.l==l) stp[s]);
35     forall(s in Stps) {
36         alternative(stp[s], all(ms in MchStps: ms.s==s) mchStp[ms]);
37         if (s.pos>1)
38             endBeforeStart(stp[<s.l,s.pos-1>],stp[s]);
39     }
40     forall(ms in MchStps)
41         alwaysEqual(batch[ms.m], mchStp[ms], ms.s.f, true, true);
42     forall(m in Mchs)
43         load[m] <= m.capacity;
44     forall(d in Lags)
45         endAtStart(stp[<d.l,d.pos1>], stp[<d.l,d.pos2>], lag[d]);
46 }
```

Decision variables:

`mchStp[ms]`: step on a given
machine (optional)

Cumul function expression:
- machine load

Constraints:
- machine capacity

Semiconductor manufacturing scheduling

- A very similar CP Optimizer model was recently studied in A. Ham and E. Cakici. *Flexible job shop scheduling problem with parallel batch processing machines: MIP and CP approaches*. Computers & Industrial Engineering, vol 102, p160-165. 2016.

*"[We] also provide the reasons why CP would be soon welcomed by the practitioners. Firstly, CP's **natural formulation** is closer to the problem description than the restricted linear programming formulation. Secondly, a **concise** CP code provides a **flexibility** and **scalability** to practitioners. Thirdly, unlike meta-heuristics which are tailor-made requiring a fine tuning of parameters to reach at the best performance, CP is **off-the-rack**. Namely, practitioners provide a high level description of the problem only. All settings of search algorithms and detailed tunings are automatically done by CP engine. Finally, **CP outperforms MIP** in the scheduling problems as we demonstrate in this study."*

Best Practices for modeling

- As for Mathematical Programming, there are many different ways to model the same problem using the CP Optimizer modeling concepts ... and all the formulations are not equivalent in terms of performance ...
- For complex problems, coming up with the most efficient formulation is still an art but there are a few guidelines that can help
- The best practices that follow are not guaranteed to always work, nothing replaces experience and creativity!
- They mostly tend to implement 2 meta-advice:
 - Factorize everything that can get factorized in the model
 - Exploit the power of the scheduling concepts (optionality, functions of time)

Best Practices for modeling

A measure of complexity of a model:

- Number of variables
- Number of groups of variables with different semantics
- Number of variable types (integer / interval variables)
- Take advantage of the expressiveness of the CP Optimizer modeling language to compact the model by decreasing the above indicators
- In particular if in a scheduling model, you are creating 1 decision variable for each time point t you are usually in bad shape
- Exploit the scheduling concepts to reason on time-lines (sequence variables, cumul functions, state functions) and avoid an explicit representation of time

Best Practices for modeling

A measure of complexity of a model is the number of constraints

- If the number of constraints grows more than linearly with the number of variables or the size of variables domains, the model will probably not scale well
- Furthermore, if a set of many small constraints can be reformulated more compactly, this often leads to stronger inference in the engine
- Example: when you need to model activities (and more generally intervals of time) that *-under some conditions-* cannot overlap, think of using sequence variables, noOverlap constraints and/or state functions

Best Practices for modeling

Be careful when using composite constraints: in scheduling models, exploit optionality of interval variables

- Examples :



$\text{presenceOf}(a) * \text{endOf}(a) + (1 - \text{presenceOf}(a)) * K$
 $\text{presenceOf}(a) \Rightarrow (10 \leq \text{startOf}(a))$



USE: $\text{endOf}(a, K)$
USE: $10 \leq \text{startOf}(a, 10)$

- Remarks:

- Exception: binary logical constraints on presence status (like $\text{presenceOf}(a) \Rightarrow \text{presenceOf}(b)$) are handled in a special (and efficient) way in the engine
- Except for constraint " presenceOf ", all constraints on scheduling constructs (interval and sequence variables, cumul and state functions) cannot be used in composite constraints

Best Practices for modeling

Be careful when using composite constraints: in scheduling models, exploit optionality of interval variables

- In case of composite constraints switching between different cases, use optional interval variables and alternative:

$(x==1) \Rightarrow (\text{lengthOf}(a)==10 \ \&\& \ \text{heightAtStart}(a,f)==5)$

$(x==2) \Rightarrow (\text{lengthOf}(a)==30 \ \&\& \ \text{heightAtStart}(a,f)==2)$

$(x==3) \Rightarrow (\text{lengthOf}(a)==70 \ \&\& \ \text{heightAtStart}(a,f)==1)$



USE: 3 optional intervals a_1, a_2, a_3 and $\text{alternative}(a, [a_1, a_2, a_3])$



- More generally, try to capture most of the non-temporal decisions by Boolean presence statuses only

- In OPL IDE: Press the **solve** button !
- In the other APIs: Call a function **solve()** !



Solving: properties

- Search is **complete**
- Search is **anytime**
- Search is **parallel** (unless stated otherwise)
- Search is **randomized**
 - Internally, some ties are broken using random numbers
 - The seed of the random number generator is a parameter of the search
- Search is **deterministic**
 - Solving twice the same problem on the same machine (even when using multiple parallel workers) with the same seed for the internal random number generator will produce the same result
 - Determinism of the search is essential in an industrial context and for debugging

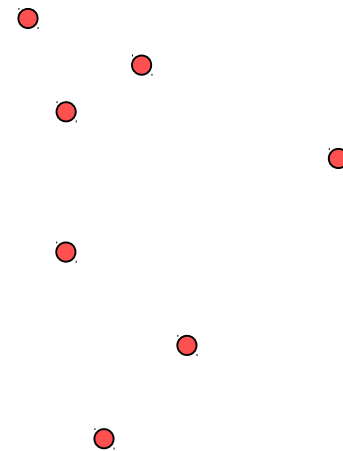
Performance on classical scheduling benchmarks

- Results published in CPAIOR-2015 (using V12.6)
 - Job-shop
 - 15 instances closed out of 48 open ones
 - Job-shop with operators
 - 208 instances closed out of 222 open ones
 - Flexible job-shop
 - 74 instances closed out of 107 open ones
 - RCPSP
 - 52 new lower bounds
 - RCPSP with maximum delays
 - 51 new lower bounds out of 58 instances
 - Multi-mode RCPSP
 - 535 instances closed out of 552
 - Multi-mode RCPSP with maximum delays
 - All 85 open instances of the benchmark closed

Performance evaluation

- We have seen quite a few scheduling models so far:

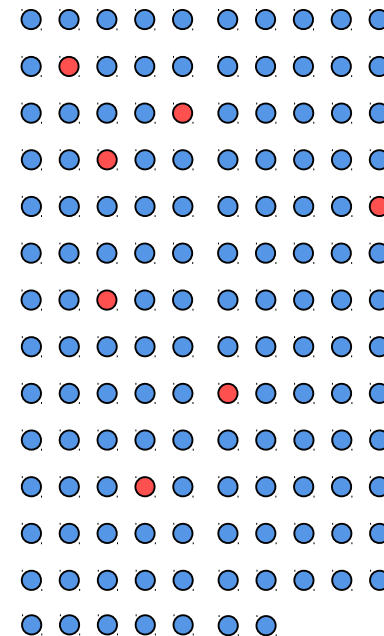
Job-shop
VRPTW
RCPSP
Flow-shop w/ E/T
Multi-Mode RCPSP
Satellite communication
Semi-conductor manufacturing



Performance evaluation

- We have seen quite a few scheduling models so far:

Job-shop
VRPTW
RCPSP
Flow-shop w/ E/T
Multi-Mode RCPSP
Satellite communication
Semi-conductor manufacturing



- As of June 2017, our performance evaluation benchmark contains 137 different scheduling models tested on a total of 3264 instances

Performance evaluation

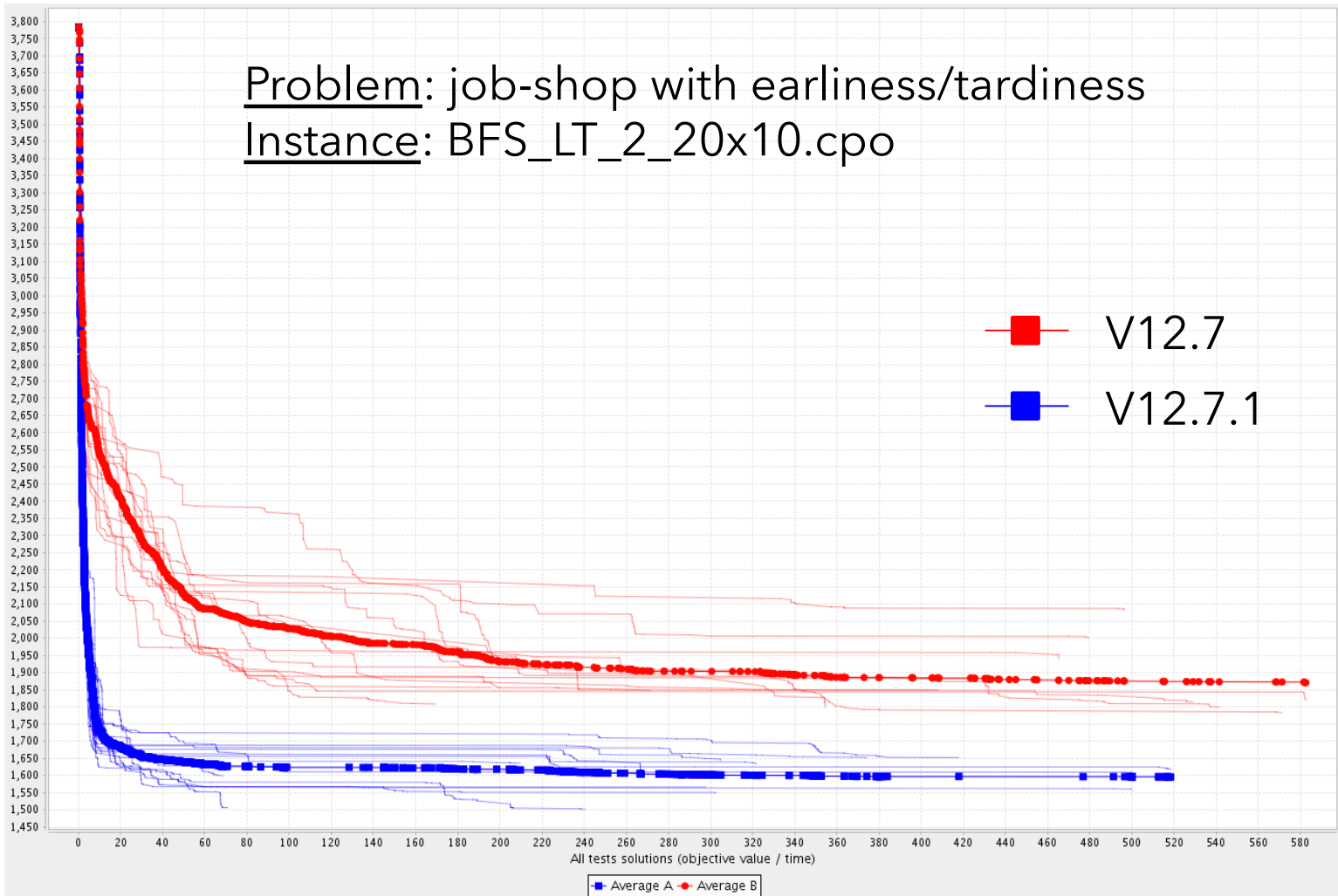
- The benchmark collects problems from different sources:
 - Classical problems (job-shop, RCPSP, ...)
 - New problems proposed in academic papers
 - Industrial scheduling problems from:
 - Customers
 - Business partners
 - End users
 - Problems discussed on our Optimization forum
 - ...
- Problems are quite diverse
 - Size range: 30 to 1.000.000 interval variables
 - Resource types: disjunctive, cumulative
 - Objective functions: makespan, weighted earliness/tardiness, resource allocation costs, activity non-execution penalties, resource transition costs, ...

Performance evaluation

- The benchmark is mostly used to monitor the performance of the automatic search
- Though the search is **complete**, it is (still) not able to solve all problems to optimality 😊
- Each problem instance is run with a given time-limit on a given number of random seeds (search is **randomized**)
- Two versions of the search algorithm A and B are compared by computing a **speed-up ratio** that estimates how much faster the best algorithm (say A) finds a solution equivalent to the best solution found by the worst algorithm (here, B)
- Speed-up ratios are aggregated on the different problem instances to compute an **average speed-up ratio**

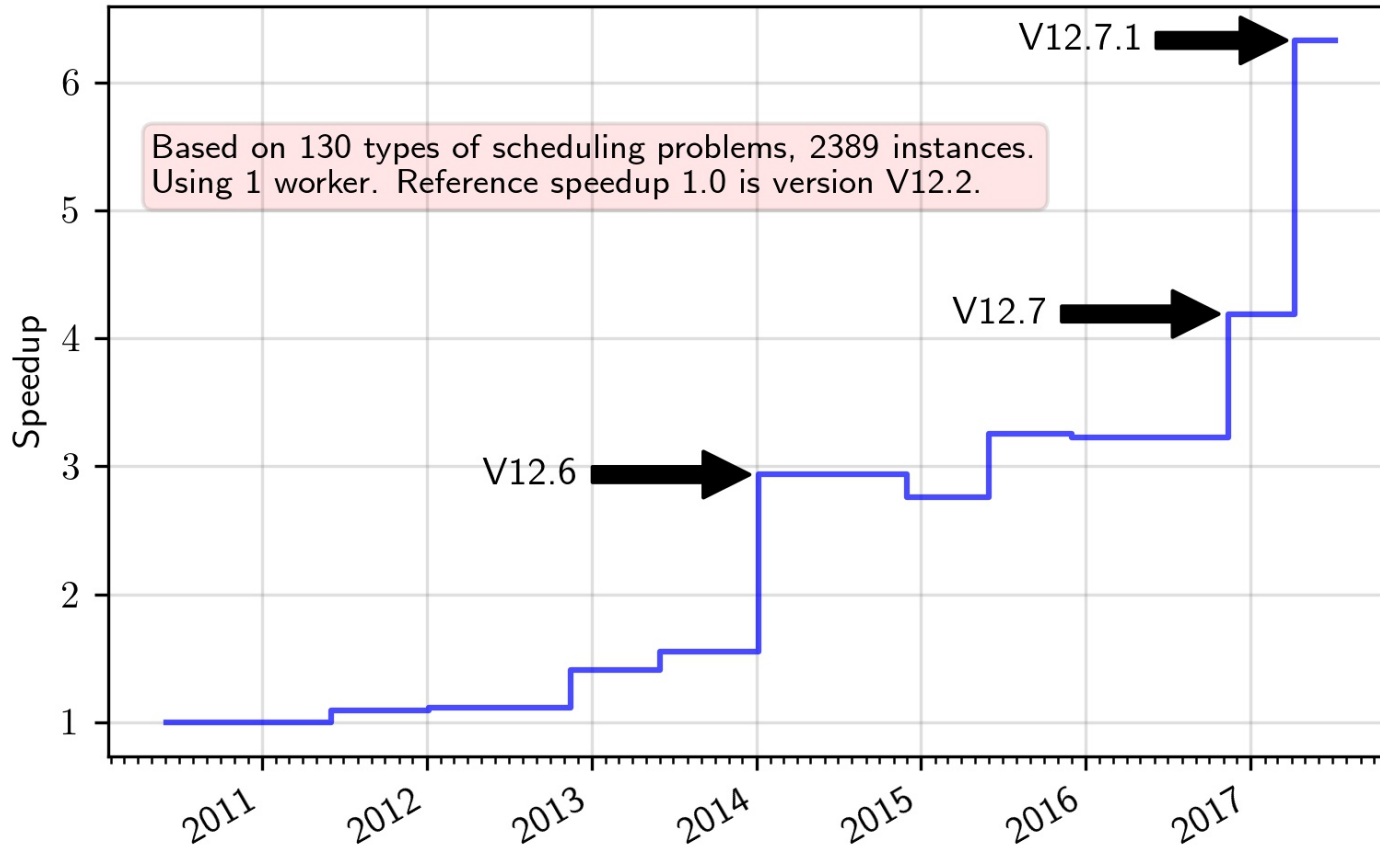
Performance evaluation

- Example



Performance evolution

CP Optimizer average speedup for scheduling problems



Performance

- Flow-shop with earliness and tardiness costs
 - Speed-up ratio between 12.2 and 12.7.1: **x 329.8**
- Multi-Mode RCPSP
 - Speed-up ratio between 12.2 and 12.7.1: **x 8.6**
- Satellite communication scheduling
 - Speed-up ratio between 12.2 and 12.7.1: **x 4.9**
- Semiconductor manufacturing scheduling
 - Speed-up ratio between 12.7 and 12.7.1: **x 2.1**
- GEO-CAPE Observation scheduling
 - Speed-up ratio between 12.7 and 12.7.1: **x 3.5**
- +132 other scheduling models
 - Average speed-up ratio between 12.2 and 12.7.1: **x 6.3**

Solving: influencing the search

Three concepts can be used to influence the search

- Search parameters
- Search phases
- Starting points

Solving: influencing the search

Search parameters (the most useful ones)

- TimeLimit = t [**+** ∞] (in seconds)
- TemporalRelaxation = **On** | Off
- XXXInferenceLevels = Low | **Basic** | Medium | Extended
- SearchType = **Restart** | DepthFirst | MultiPoint
- Workers = n [**#cores**]
- LogPeriod = n [**1000**]
- RandomSeed = n [**1**]

Solving: influencing the search

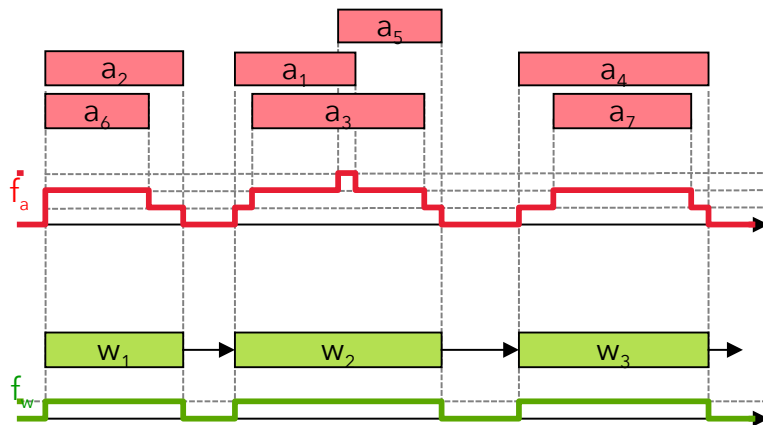
Search phases

- A **search phase** P is a subset of decision variables of the same type (integer, interval, sequence)
- The search can be passed a list of search phases P_1, P_2, \dots, P_n it will give it a hint to try to fix all variables in P_i before fixing the ones in P_{i+1} . Variables that do not appear in any search phase are fixed in the end.

Solving: influencing the search

Search phases

- A **search phase** P is a subset of decision variables of the same type (integer, interval, sequence)
- The search can be passed a list of search phases P_1, P_2, \dots, P_n it will give it a hint to try to fix all variables in P_i before fixing the ones in P_{i+1} . Variables that do not appear in any search phase are fixed in the end.



`cp.setSearchPhases(a)`

Solving: influencing the search

Starting point

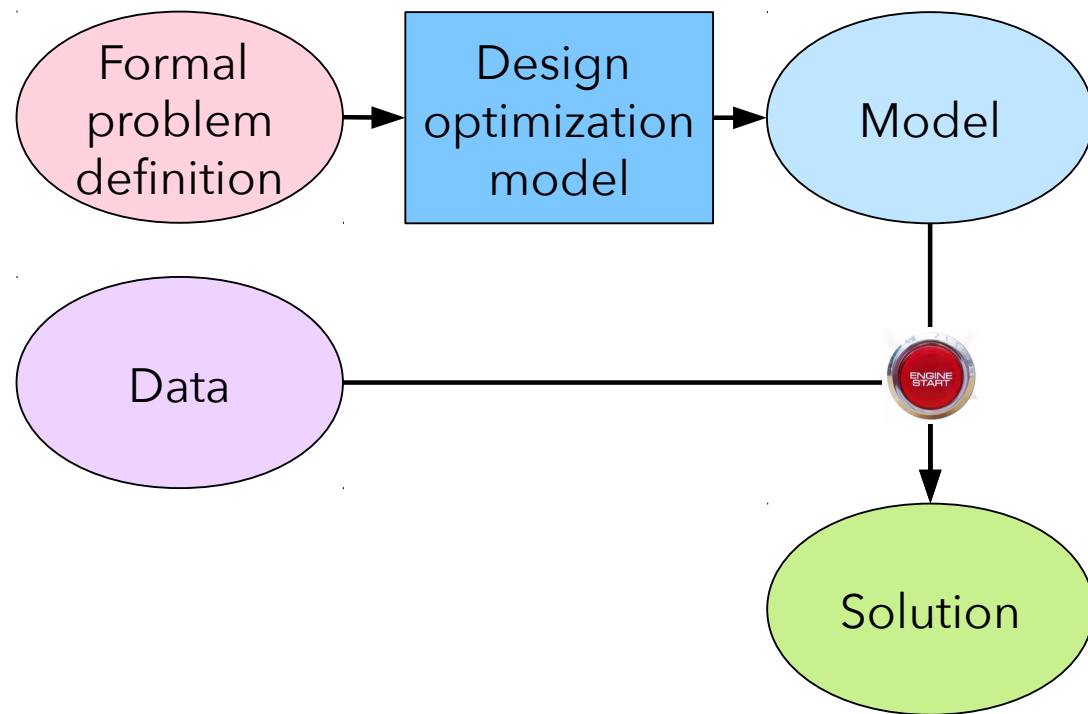
- The search can be specified a **starting point solution** as input
- Use cases:
 - Search process has been interrupted; restart from last solution
 - A problem specific heuristic is available to provide a solution to start from
 - Multi-objective lexicographical objective: minimize f_1 , then minimize f_2 with some constraint on f_1 , ...
 - When hard to find a feasible solution: start from a relaxed problem that minimizes constraint violation
 - Solving very similar successive models, for instance in dynamic scheduling, in re-scheduling

Solving: influencing the search

Starting point

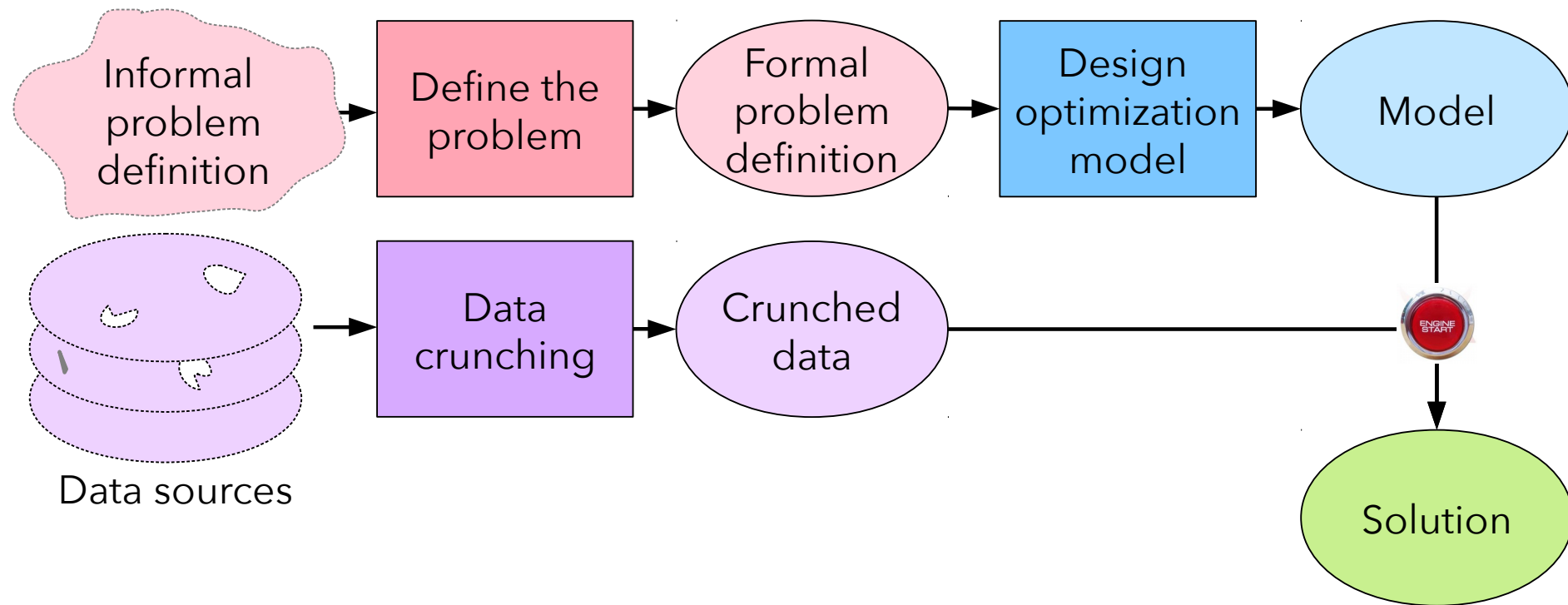
- The search can be specified a **starting point solution** as input
- If the starting point is feasible and complete, the search is **guaranteed** to first visit this solution
- Otherwise, the information in the starting point is used as a heuristic guideline for the search

- Process for building an optimization engine for an application



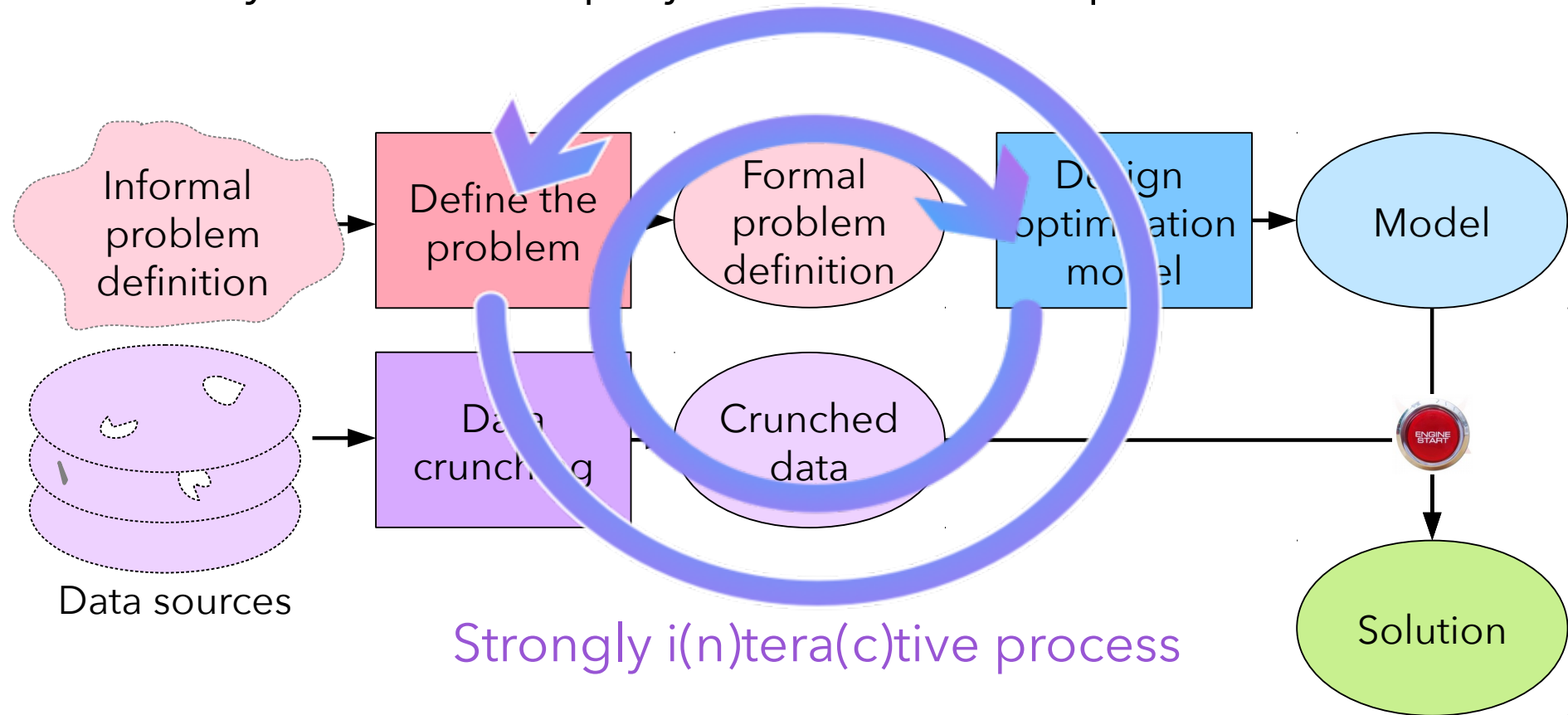
Tools

- Process for building an optimization engine for an application
- Reality of industrial projects is more complex



Tools

- Process for building an optimization engine for an application
- Reality of industrial projects is more complex



- Typical questions/issues arising during model design
 - How does my current model look like when instantiated on some data ?
 - Does it contains some weird things I'm not aware of ?
 - Why is it infeasible ?
 - Bug in the model ?
 - Bug in the data ?
 - Why is it difficult to find a feasible solution?
 - Is my model performing better than another variant I tried?
 - I'd like some advice/help from some CP Optimizer expert

- CP Optimizer provides the same type of tools as Mathematical Programming (e.g. CPLEX) for helping designing / debugging models:
 - Input/output file format
 - Model warnings
 - Search log
 - Conflict refiner
 - Interactive executable

Tools: Input/output file format (.cpo)

- Objective:
 - Make it easier to understand the content of a model
 - Communicate a model to engine experts (IBM, partners, ...) regardless of the API used to build it (OPL, C++, Python, Java, .NET)
- Structure of a .cpo file
 - Human readable
 - Flat (no cycle, no forall statements)
 - Internal information such as CPO version or platform used
 - Information such as source code line numbers
 - Includes search parameter values, phases, starting point
- Facilities
 - Export model before/instead of solve
 - Export model during solve (with current domains)
 - Import model instead of normal modeling

Tools: Input/output file format (.cpo)

```
////////////////////////////////////  
// CPO file generated from:  
// FlowShopET.py  
////////////////////////////////////  
  
//--- Variables ---  
"00-0" = intervalVar(size=14);  
"00-1" = intervalVar(size=1);  
"00-2" = intervalVar(size=12);  
"01-0" = intervalVar(size=16);  
...  
  
//--- Objective ---  
#line 14 "FlowShopET.py"  
minimize(sum([3.29366 * abs(endOf("00-2") - 51), 2.86608 * abs(endOf("01-2") - 69), ...]) / 579.56056);  
  
//--- Constraints ---  
#line 19 "FlowShopET.py"  
endBeforeStart("00-0", "00-1");  
endBeforeStart("00-1", "00-2");  
...  
#line 22 "FlowShopET.py"  
noOverlap(["00-0", "01-0", "02-0", "03-0", "04-0", "05-0", "06-0", "07-0", "08-0", "09-0"]);  
noOverlap(["00-1", "01-1", "02-1", "03-1", "04-1", "05-1", "06-1", "07-1", "08-1", "09-1"]);  
noOverlap(["00-2", "01-2", "02-2", "03-2", "04-2", "05-2", "06-2", "07-2", "08-2", "09-2"]);  
  
//--- Parameters ---  
#line off  
parameters {  
    TimeLimit = 10;  
}
```


Tools: Model warnings

- Like a compiler, CP Optimizer can analyze the model and print some warnings
 - When there is something suspicious in the model
 - Regardless how the model was created (C++, OPL, Python ...)
 - Including guilty part of the model in the cpo file format
 - Including source code line numbers (if known)
 - 3 levels of warnings, more than 70 types of warnings

```
cppfile.cpp:24: Warning: Unused interval variable 'x'.
    x = intervalVar(start=1..50, size=5..10)

javafile.java:20: Warning: Interval variable 'itv' has empty domain.
    itv = intervalVar(start=0..10, length=5, end=100..110)

pythonfile.py:7: Warning: Constraint is always true.
    u+v >= 5
pythonfile.py:8: Warning: Constraint is always false,
pythonfile.py:8: the model is infeasible.
    u+v < 5
    u+v < 5

satellite.cpo:2995:29: Warning: Constraint 'alternative':
satellite.cpo:2995:29: there is only one alternative interval variable.
    alternative("task(134A,176,1)", ["alt(170,6,1232,19,1266)"], 1)
```

Tools: Model warnings

```
"Transition matrix for constraint 'noOverlap' does not satisfy the triangle inequality (path %d->%d->%d)."  
"Constraint '%s': empty range [%d..%d]. Interval variable '%w' will be set to absent."  
"Constraint '%s': the constraint is false because the range [%d..%d] is empty."  
"Constraint 'alwaysIn': empty range [%d..%d]. The effect is the same as alwaysNoState."  
"Constraint '%s': interval variable '%w' has zero length therefore the constraint is always true."  
"Interval variable '%w' used in pulse has zero length, the pulse is zero everywhere."  
"Interval variable '%w' used in expression 'overlapLength' has zero length."  
"Boolean expression 'presenceOf' is always true because interval variable '%w' is declared present."  
"Boolean expression 'presenceOf' is always false because interval variable '%w' is declared absent."  
"Constraint 'alternative': array of alternatives is empty. Interval variable '%w' will be set to absent."  
"Constraint 'span': array of spanned interval variables is empty. Interval variable '%w' will be set to absent."  
"Sequence variable '%w' is defined over empty set of interval variables."  
"Constraint 'synchronize': empty set of synchronized interval variables."  
"Constraint 'isomorphism': first set of interval variables is empty."  
"Constraint 'isomorphism': second set of interval variables is empty."  
"Constraint 'alternative': there is only one alternative interval variable."  
"Constraint 'span': there is only one spanned interval variable."  
"Sequence variable '%w' is defined over only one interval variable."  
"Constraint 'isomorphism': first set of interval variables contains only one interval."  
"Constraint 'isomorphism': second set of interval variables contains only one interval."  
"Constraint 'alternative': interval variable '%w' is the master but also one of the alternatives. "  
"Constraint 'span': interval variable '%w' is the master but also one of the spanned intervals."  
"Constraint 'synchronize': interval variable '%w' is the master but also one of the synchronized intervals."  
"Constraint '%s': specified interval [%d..%d] has zero length, the constraint is always true."  
"Expression 'pulse': specified interval [%d..%d] has zero length, the pulse is zero everywhere."  
"Expression 'overlapLength': specified interval [%d..%d] has zero length."  
"Cumulative expression '%s': height is zero, expression is zero everywhere."  
"Constraint 'span': master interval variable '%w' is optional but spanned interval '%w' is present."  
"Constraint '%s': interval variable '%w' is used more than once."  
"Constraint 'isomorphism' is false because of inconsistent number of present/absent intervals in the two sets."
```

Tools: Model warnings

```
"Unused integer variable '%w'."
"Integer variable '%w' is used only once."
"Unused interval variable '%w'."
"Interval variable '%w' is used only once."
"Unused sequence variable '%w'."
"The constraint is always true, it will be removed."
"The constraint is always false, the model is infeasible."
"Unnecessary integer variable '%w' is used only once."
"Unnecessary interval variable '%w' is used only once."
"Division by zero."
"Function '%s' does not support negative argument."
"Parameter of function '%s' is out of range."
"Sequence variable without noOverlap constraint."
"Matrix doesn't specify transition distance for type %d."
"Comparison of floating point expressions may result in true or false for very small changes in expression values."
"Model found inconsistent while processing '%w'."
"Interval variable '%w' used in expression %s is declared absent."
"Interval variable '%w' used in constraint %s is declared absent."
"Interval variable '%w' used in constraint %s is declared absent. Therefore the constraint is always true."
"Interval variable '%w' is declared with zero length."
"Constraint 'alternative': all alternative interval variables are declared absent."
"Constraint 'span': all spanned interval variables are declared absent."
"Sequence variable '%w' is defined over only absent interval variables."
"Constraint 'synchronize': all synchronized interval variables are declared absent."
"Constraint 'isomorphism': all intervals variables in the first set are absent."
"Constraint 'isomorphism': all intervals variables in the second set are absent."
"Constraint 'alternative': all but one of the alternative interval variables is declared absent: '%w'."
"Constraint 'span': all but one of the spanned interval variables is declared absent: '%w'."
"Sequence variable '%w': all but one of the interval variables is declared absent: '%w'."
"Constraint 'isomorphism': all but one of the intervals in the first set is declared absent: '%w'."
"Constraint 'isomorphism': all but one of the intervals in the second set is declared absent: '%w'."
```

- Objective: understand the behavior of the search

```
! -----  
! Maximization problem - 2980 variables, 853 constraints  
! Workers = 2  
! TimeLimit = 30  
! Initial process time : 0.01s (0.00s extraction + 0.01s propagation)  
! . Log search space : 4627.3 (before), 4627.3 (after)  
! . Memory usage : 16.9 MB (before), 19.7 MB (after)  
! Using parallel search with 2 workers.  
! -----  
! Best Branches Non-fixed W Branch decision  
* 746 3945 0.79s 1 -  
746 4000 2924 1 on task("8")  
746 4000 2908 2 on alt({"186",2,66})  
...  
! Time = 1.37s, Explored branches = 35832, Memory usage = 55.5 MB  
! Best Branches Non-fixed W Branch decision  
818 12000 2920 1 on task("184")  
...  
! -----  
! Search terminated by limit, 6 solutions found.  
! Best objective : 826  
! Number of branches : 709092  
! Number of fails : 179648  
! Total memory usage : 54.5 MB (52.9 MB CP Optimizer + 1.6 MB Concert)  
! Time spent in solve : 30.03s (30.01s engine + 0.01s extraction)  
! Search speed (br. / s) : 23625.4  
! -----
```

Problem
characteristics

- Objective: understand the behavior of the search

```
! -----  
! Maximization problem - 2980 variables, 853 constraints  
! Workers = 2  
! TimeLimit = 30  
! Initial process time : 0.01s (0.00s extraction + 0.01s propagation)  
! . Log search space : 4627.3 (before), 4627.3 (after)  
! . Memory usage : 16.9 MB (before), 19.7 MB (after)  
! Using parallel search with 2 workers.  
! -----  
! Best Branches Non-fixed W Branch decision  
* 746 3945 0.79s 1 -  
746 4000 2924 1 on task("8")  
746 4000 2908 2 on alt({"186",2,66})  
...  
! Time = 1.37s, Explored branches = 35832, Memory usage = 55.5 MB  
! Best Branches Non-fixed W Branch decision  
818 12000 2920 1 on task("184")  
...  
! -----  
! Search terminated by limit, 6 solutions found.  
! Best objective : 826  
! Number of branches : 709092  
! Number of fails : 179648  
! Total memory usage : 54.5 MB (52.9 MB CP Optimizer + 1.6 MB Concert)  
! Time spent in solve : 30.03s (30.01s engine + 0.01s extraction)  
! Search speed (br. / s) : 23625.4  
! -----
```

Modified
parameter values

Tools: Search log

- Objective: understand the behavior of the search

```
! -----  
! Maximization problem - 2980 variables, 853 constraints  
! Workers                = 2  
! TimeLimit              = 30  
! Initial process time : 0.01s (0.00s extraction + 0.01s propagation)  
! . Log search space    : 4627.3 (before), 4627.3 (after)  
! . Memory usage       : 16.9 MB (before), 19.7 MB (after)  
! Using parallel search with 2 workers.  
! -----  
!           Best Branches  Non-fixed    W      Branch decision  
*           746           3945 0.79s      1         -  
           746           4000      2924    1       on task("8")  
           746           4000      2908    2       on alt({"186",2,66})  
...  
! Time = 1.37s, Explored branches = 35832, Memory usage = 55.5 MB  
!           Best Branches  Non-fixed    W      Branch decision  
           818           12000      2920    1       on task("184")  
...  
! -----  
! Search terminated by limit, 6 solutions found.  
! Best objective        : 826  
! Number of branches    : 709092  
! Number of fails       : 179648  
! Total memory usage    : 54.5 MB (52.9 MB CP Optimizer + 1.6 MB Concert)  
! Time spent in solve   : 30.03s (30.01s engine + 0.01s extraction)  
! Search speed (br. / s) : 23625.4  
! -----
```

Root node
information

Tools: Search log

- Objective: understand the behavior of the search

```
! -----
! Maximization problem - 2980 variables, 853 constraints
! Workers                = 2
! TimeLimit              = 30
! Initial process time  : 0.01s (0.00s extraction + 0.01s propagation)
! . Log search space    : 4627.3 (before), 4627.3 (after)
! . Memory usage       : 16.9 MB (before), 19.7 MB (after)
! Using parallel search with 2 workers.
! -----
!           Best Branches  Non-fixed   W      Branch decision
*           746      3945 0.79s        1        -
           746      4000      2924    1      on task("8")
           746      4000      2908    2      on alt({"186",2,66})
...
! Time = 1.37s, Explored branches = 35832, Memory usage = 55.5 MB
!           Best Branches  Non-fixed   W      Branch decision
           818      12000      2920    1      on task("184")
...
! -----
! Search terminated by limit, 6 solutions found.
! Best objective        : 826
! Number of branches    : 709092
! Number of fails       : 179648
! Total memory usage    : 54.5 MB (52.9 MB CP Optimizer + 1.6 MB Concert)
! Time spent in solve   : 30.03s (30.01s engine + 0.01s extraction)
! Search speed (br. / s) : 23625.4
! -----
```

New incumbent
solutions
(time, worker)

Tools: Search log

- Objective: understand the behavior of the search

```
! -----
! Maximization problem - 2980 variables, 853 constraints
! Workers                = 2
! TimeLimit              = 30
! Initial process time  : 0.01s (0.00s extraction + 0.01s propagation)
! . Log search space    : 4627.3 (before), 4627.3 (after)
! . Memory usage       : 16.9 MB (before), 19.7 MB (after)
! Using parallel search with 2 workers.
! -----
!           Best Branches  Non-fixed   W      Branch decision
*          746      3945 0.79s        1        -
           746      4000      2924    1      on task("8")
           746      4000      2908    2      on alt({"186",2,66})
...
! Time = 1.37s, Explored branches = 35832, Memory usage = 55.5 MB
!           Best Branches  Non-fixed   W      Branch decision
           818      12000      2920    1      on task("184")
...
! -----
! Search terminated by limit, 6 solutions found.
! Best objective        : 826
! Number of branches    : 709092
! Number of fails       : 179648
! Total memory usage    : 54.5 MB (52.9 MB CP Optimizer + 1.6 MB Concert)
! Time spent in solve   : 30.03s (30.01s engine + 0.01s extraction)
! Search speed (br. / s) : 23625.4
! -----
```

Periodical log
with fail information,
number of unfixed
vars, current decision

Tools: Search log

- Objective: understand the behavior of the search

```
! -----
! Maximization problem - 2980 variables, 853 constraints
! Workers                = 2
! TimeLimit              = 30
! Initial process time  : 0.01s (0.00s extraction + 0.01s propagation)
! . Log search space    : 4627.3 (before), 4627.3 (after)
! . Memory usage       : 16.9 MB (before), 19.7 MB (after)
! Using parallel search with 2 workers.
! -----
!           Best Branches  Non-fixed   W      Branch decision
*           746      3945 0.79s        1         -
           746      4000      2924    1      on task("8")
           746      4000      2908    2      on alt({"186",2,66})
...
! Time = 1.37s, Explored branches = 35832, Memory usage = 55.5 MB
!           Best Branches  Non-fixed   W      Branch decision
           818      12000      2920    1      on task("184")
...
! -----
! Search terminated by limit, 6 solutions found.
! Best objective        : 826
! Number of branches    : 709092
! Number of fails       : 179648
! Total memory usage    : 54.5 MB (52.9 MB CP Optimizer + 1.6 MB Concert)
! Time spent in solve   : 30.03s (30.01s engine + 0.01s extraction)
! Search speed (br. / s) : 23625.4
! -----
```

Final information
with solution status
and search statistics

- Objective: **eXplainable scheduling**
- Provide an explanation for an infeasible model by providing a **minimal infeasible subset** of constraints
 - Algorithm described in: *P. Laborie. An Optimal Iterative Algorithm for Extracting MUCs in a Black-box Constraint Network. In: Proc. ECAI-2014*
- Use cases:
 - **Model debugging** (errors in model)
 - **Data debugging** (inconsistent data)
 - Providing explanations to the user (**why P in the solution?**)
=> add $\neg P (\wedge \text{obj} \leq \text{obj}^*)$ and compute a minimal infeasible subset
 - The model and data are correct, but the associated data represents a **real-world conflict** in the system being modeled

Tools: Conflict refiner

- Example: satellite communication problem
- On a given instance explain why we cannot schedule all the tasks

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

Tools: Conflict refiner

- Example: satellite communication problem
- On a given instance explain why we cannot schedule all the tasks

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

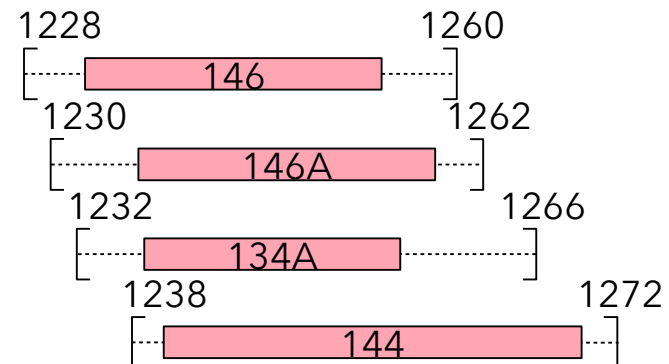
Tools: Conflict refiner

- Example: satellite communication problem
- On a given instance explain why we cannot schedule all the tasks

```
1 using CP;
2 tuple Resource { string name; key int id; int cap; }
3 tuple Task { string name; key int id; int prio; }
4 tuple Alt { int task; int res; int smin; int dur; int emax; }
5
6 {Resource} Res = ...;
7 {Task} Tasks = ...;
8 {Alt} Alts = ...;
9 {int} Priorities = { 1,2,3,4,5 };
10
11 dvar interval task[t in Tasks] optional;
12 dvar interval alt[a in Alts] optional in a.smin..a.emax size a.dur;
13
14 dexpr int nb[p in Priorities] =
15     sum(t in Tasks: t.prio==p) presenceOf(task[t]);
16
17 maximize staticLex(nb[1], nb[2], nb[3], nb[4], nb[5]);
18 subject to {
19     forall(t in Tasks)
20         alternative(task[t], all(a in Alts: a.task==t.id) alt[a]);
21     forall(r in Res)
22         sum(a in Alts: a.res==r.id) pulse(alt[a],1) <= r.cap;
23 }
```

A conflict:

Line	Iteration
20	t = <"134A",176,1>
20	t = <"144",191,1>
20	t = <"146",193,1>
20	t = <"146A",194,1>
22	r = <"LION",6,3>



Tools: Interactive CP Optimizer

- An executable delivered with CP Optimizer that let you:
 - Load/save models in .cpo format
 - Set or change parameter values
 - Invoke a solve or a conflict refinement
 - Interrupt it
 - Evaluate variability in performance by solving the model multiple times with different random seeds and provide statistics
- Example: is job-shop instance `ft10` better solved in sequential mode with default or extended noOverlap inference level?

Tools: Interactive CP Optimizer

```
<static_pic> cpooptimizer
Welcome to IBM(R) ILOG(R) CP Interactive Optimizer 12.7.0.0
Copyright IBM Corp. 1990, 2016. All Rights Reserved
```

```
CP-Optimizer> read JobShop-ft10.cpo
CP-Optimizer> set Workers 1
New value for parameter Workers: 1
CP-Optimizer> tools runseeds 100
Benchmarking current problem on 100 runs...
Run          Solution  Proof      Time (s)  Objective
-----
1            1          1          4.05      930
...
100         1          1          3.64      930
-----
All runs complete
Average  100 %      100 %      3.4151     930
Std dev          0.4205     0
```

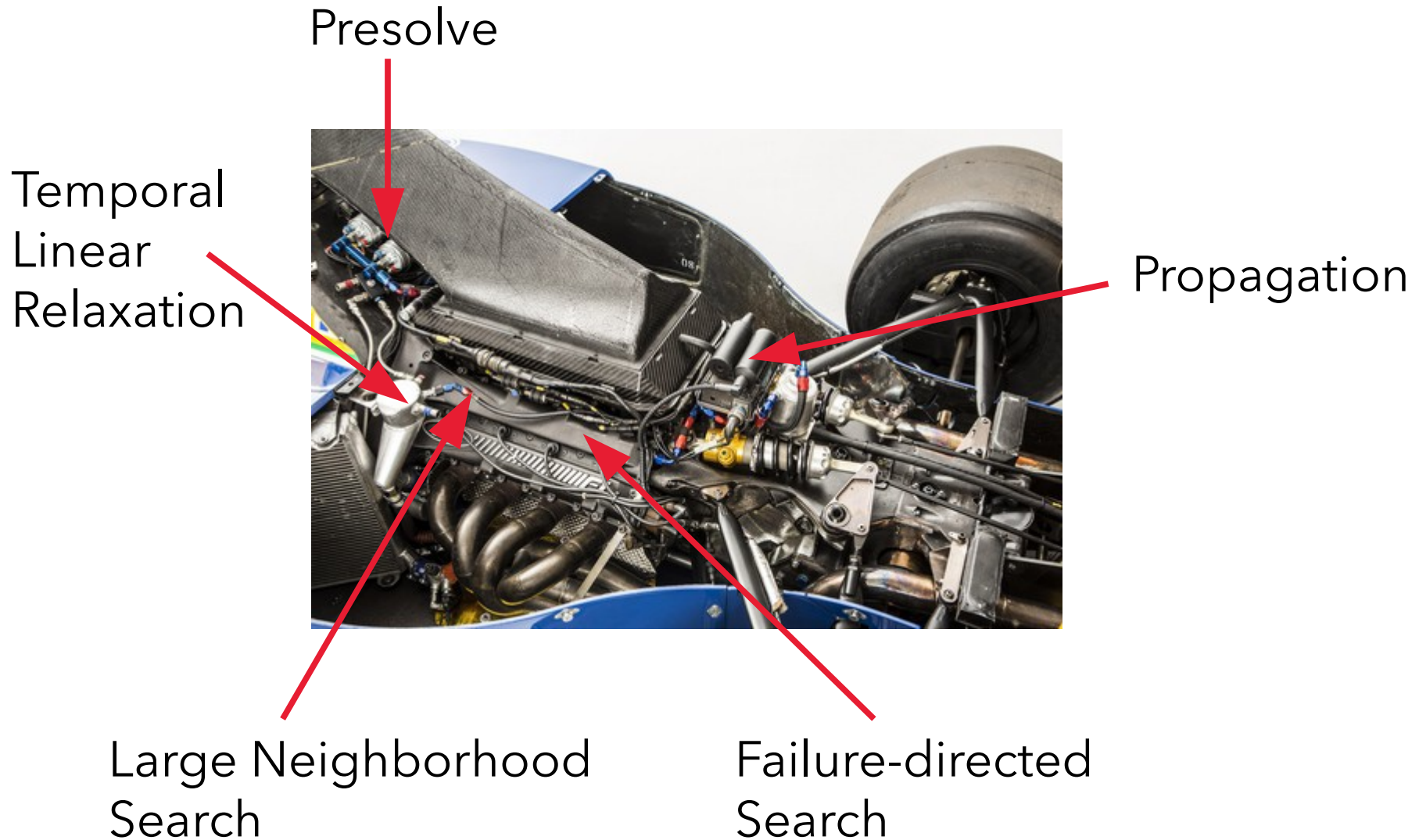
```
CP-Optimizer> set NoOverlapInferenceLevel Extended
New value for parameter NoOverlapInferenceLevel: Extended
CP-Optimizer> tools runseeds 100
Benchmarking current problem on 100 runs...
Run          Solution  Proof      Time (s)  Objective
-----
1            1          1          4.13      930
...
100         1          1          3.39      930
-----
All runs complete
Average  100 %      100 %      3.9882     930
Std dev          0.3892     0
CP-Optimizer> quit
```

Q: Is job-shop instance ft10 better solved in sequential mode with `default` or `extended noOverlap` inference level?

A: Default level seems better ...

Note: ft10 is a small but very challenging and famous job-shop instance that remained open for more than 25 years (1963-1989). Automatic search of CP Optimizer solves it in just a few seconds.

Under the hood



Under the hood: Presolve

- Before actually starting the search, the model is automatically presolved/reformulated in order to increase performance by applying a set of **presolve rules**
- Example of presolve rules:
 - $\text{endOf}(x, -\infty) \leq \text{startOf}(y, +\infty)$
 $\equiv \text{endBeforeStart}(x, y)$
 - $\text{overlapLength}(x, y) = 0, \text{startBeforeEnd}(x, y)$
 $\equiv \text{endBeforeStart}(x, y)$
 - $\text{presenceOf}(x) - \text{presenceOf}(y) \leq 0$
 $\equiv \text{presenceOf}(x) \Rightarrow \text{presenceOf}(y)$
 - $x \neq y, y \neq z, x \neq z$
 $\equiv \text{allDifferent}([x, y, z])$
 - $xy \neq z + t, z + xy == a + b, 100 \leq xy - z$
 $\equiv u == xy, u \neq z + t, z + u == a + b, 100 \leq u - z$

Under the hood: Propagation

- Logical network
- Temporal network
- Time-lines (noOverlap, cumuFunction, stateFunction)

Logical network

- Aggregates all binary constraints on interval presence as an implication graph between literals or their opposite
 $[!]presenceOf(u) \Rightarrow [!]presenceOf(v)$
- Equivalent to a 2-SAT model
- Computes graph condensation and transitive closure:
 - Detects infeasibility
 - Allows querying in $O(1)$ whether $[!]presenceOf(x) \Rightarrow [!]presenceOf(y)$ for any (x,y)

Precedence network

- Aggregates all precedence constraints (like $\text{endBeforeStart}(u,v,d_{uv})$) in a STN extended with Boolean presence status of nodes
 - Nodes: start or end of (optional) interval variables
 - Arcs: minimal delay between two nodes (when both are present)
- Temporal domain of a node t is maintained as a range $[t_{min}, t_{max}]$ representing the possible values **if the interval is present**
- Propagation exploits the **Logical network**

Precedence network

- Propagation exploits the **Logical network**

- Example:

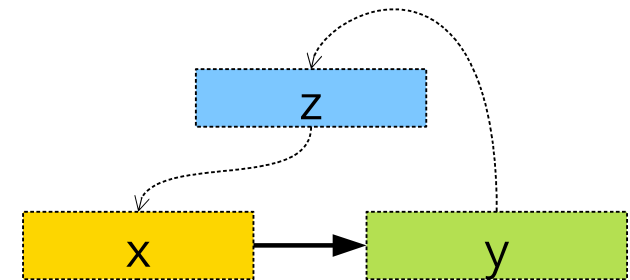
`endBeforeStart(x,y)`

`presenceOf(y)=>presenceOf(z)`

`presenceOf(z)=>presenceOf(x)`

- Logical network deduces:

`presenceOf(y)=>presenceOf(x)`



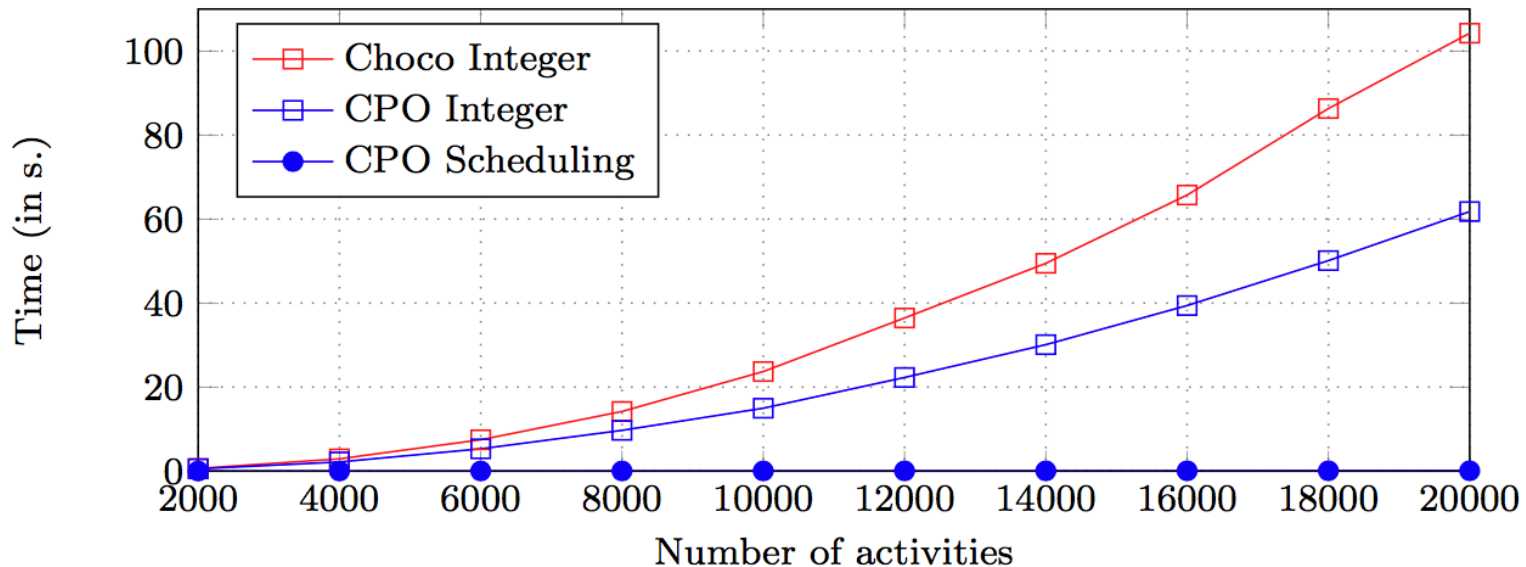
- The precedence constraint can propagate the bounds of x on y: $\text{smin}(y) \leftarrow \max(\text{smin}(y), \text{emin}(x))$
- This is very powerful: propagation occurs even when the presence status is still unfixed

Precedence network

- Propagation exploits the **Logical network**
- This is very powerful: propagation occurs even when the presence status is still unfixed
- Classical STN propagation algorithms are extended to perform this type of directional propagation
- Algorithms used in CP Optimizer:
 - At root node: extension of an improved version of Bellman-Ford algorithm: B. Cherkassky, A. Goldberg, T. Radzic. *Shortest Paths Algorithms: Theory and Experimental Evaluation*. Mathematical Programming 73, 129-174 (1996)
 - During the search: extension of the algorithm described in: A. Cesta, A. Oddi. *Gaining Efficiency and Flexibility in the Simple Temporal Problem*. In: Proc. TIME-96 (1996)

Precedence network

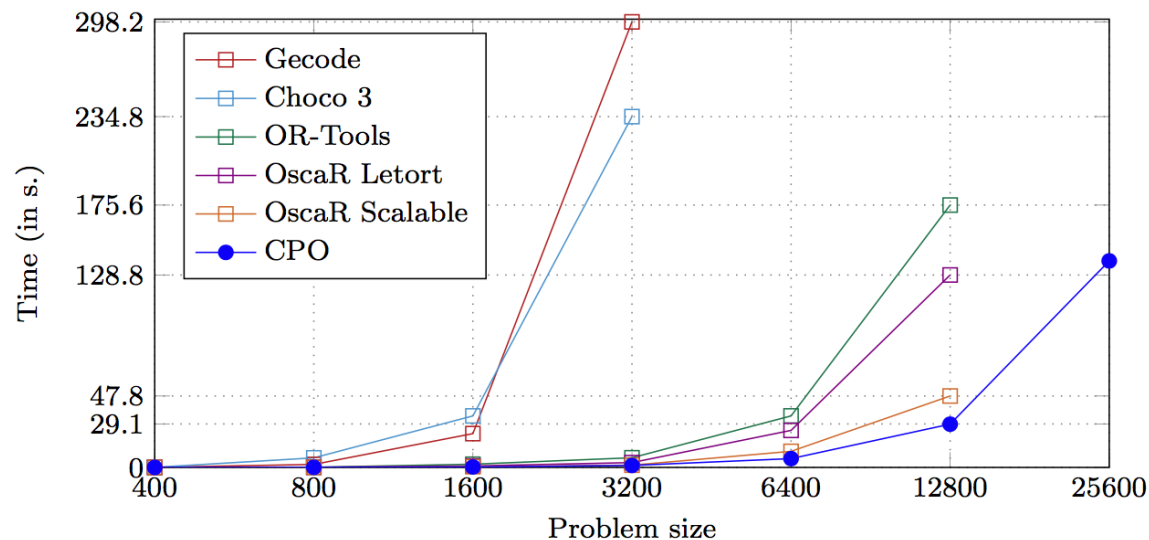
- These specialized algorithms for precedence networks are much faster than naive CP propagation of individual precedence constraints (like $x \leq y$)
 - Example: root node propagation time for a chain of n activities (`endBeforeStart(x[i],x[i+1])`)



Under the hood: Propagation

Time-lines (noOverlap, cumulFunction, stateFunction)

- Default propagation algorithm is the **timetable** that incrementally maintains the domain of the function as a set of segments with bounds on the function values
- Comparison with recent algorithms studied in: *S. Gay, R. Hartert, P. Schaus: Simple and Scalable Time-Table Filtering for the Cumulative Constraint. In: Proc. CP 2015*



Under the hood: Propagation

Time-lines (noOverlap, cumulFunction, stateFunction)

- Sequence variables are internally represented as a **precedence graph** on interval variables

Time-lines (noOverlap, cumulFunction, stateFunction)

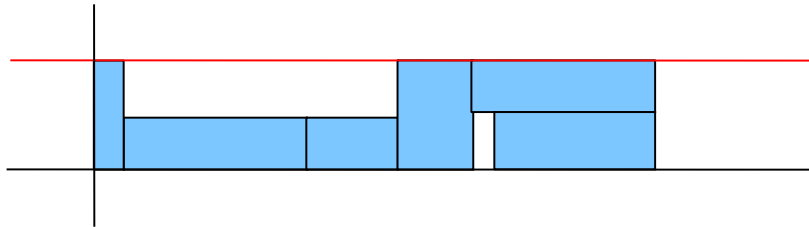
- Stronger propagation algorithms are available and are automatically turned on in the search depending on the context:
 - Multiple $O(n \log(n))$ algorithms for disjunctions (noOverlap):
P. Vilím: Global constraints in scheduling. Ph.D. thesis. 2007.
 - $O(n^2)$ time-table edge-finding for cumul functions:
P. Vilím: Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources. Proc. CPAIOR-2011.
- This additional propagation can also be turned on with some parameters, typically:
 - NoOverlapInferenceLevel = Medium | Extended
 - CumulFunctionInferenceLevel = Medium | Extended

Under the hood: Search

- The automatic search concurrently runs two algorithms:
 - **Large Neighborhood Search:** A heuristic search aiming at converging quickly to good quality solutions
P. Laborie, D. Godard. *Self-Adapting Large Neighborhood Search: Application to Single-Mode Scheduling Problems*. In: Proc. MISTA-2007
 - **Failure-Directed Search:** A complete search aiming at proving no solution exist better than the current one
P. Vilím, P. Laborie, P. Shaw. *Failure-directed Search for Constraint-based Scheduling*. In: Proc. CPAIOR-2015
- An alternative search based on **Genetic Algorithms** is also available (SearchType=MultiPoint):
R. Dumeur. *Evolutionary Multi Point Search In CPLEX Studio's Constraint Programming Engine*. INFORMS-2014

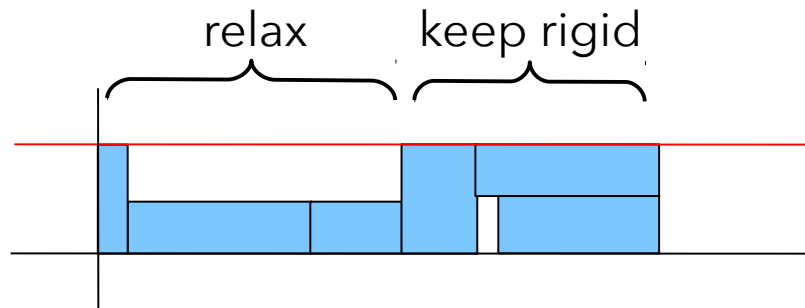
Under the hood: Large Neighborhood Search

- Iterative improvement method:
 1. Start with an existing solution (produced using some heuristics + classical CP search tree)



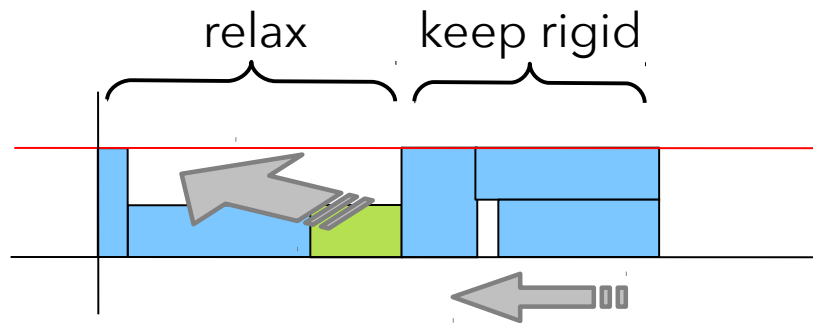
Under the hood: Large Neighborhood Search

- Iterative improvement method:
 1. Start with an existing solution (produced using some heuristics + classical CP search tree)
 2. Take part of the solution (fragment) and relax it. Fix the structure of the rest (but no start/end values: notion of Partial Order Schedule)



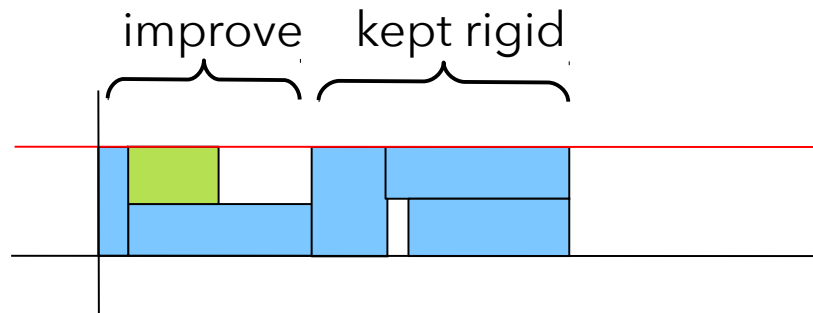
Under the hood: Large Neighborhood Search

- Iterative improvement method:
 1. Start with an existing solution (produced using some heuristics + classical CP search tree)
 2. Take part of the solution (fragment) and relax it. Fix the structure of the rest (but no start/end values: notion of Partial Order Schedule)
 3. Find (improved) solution using a limited search tree



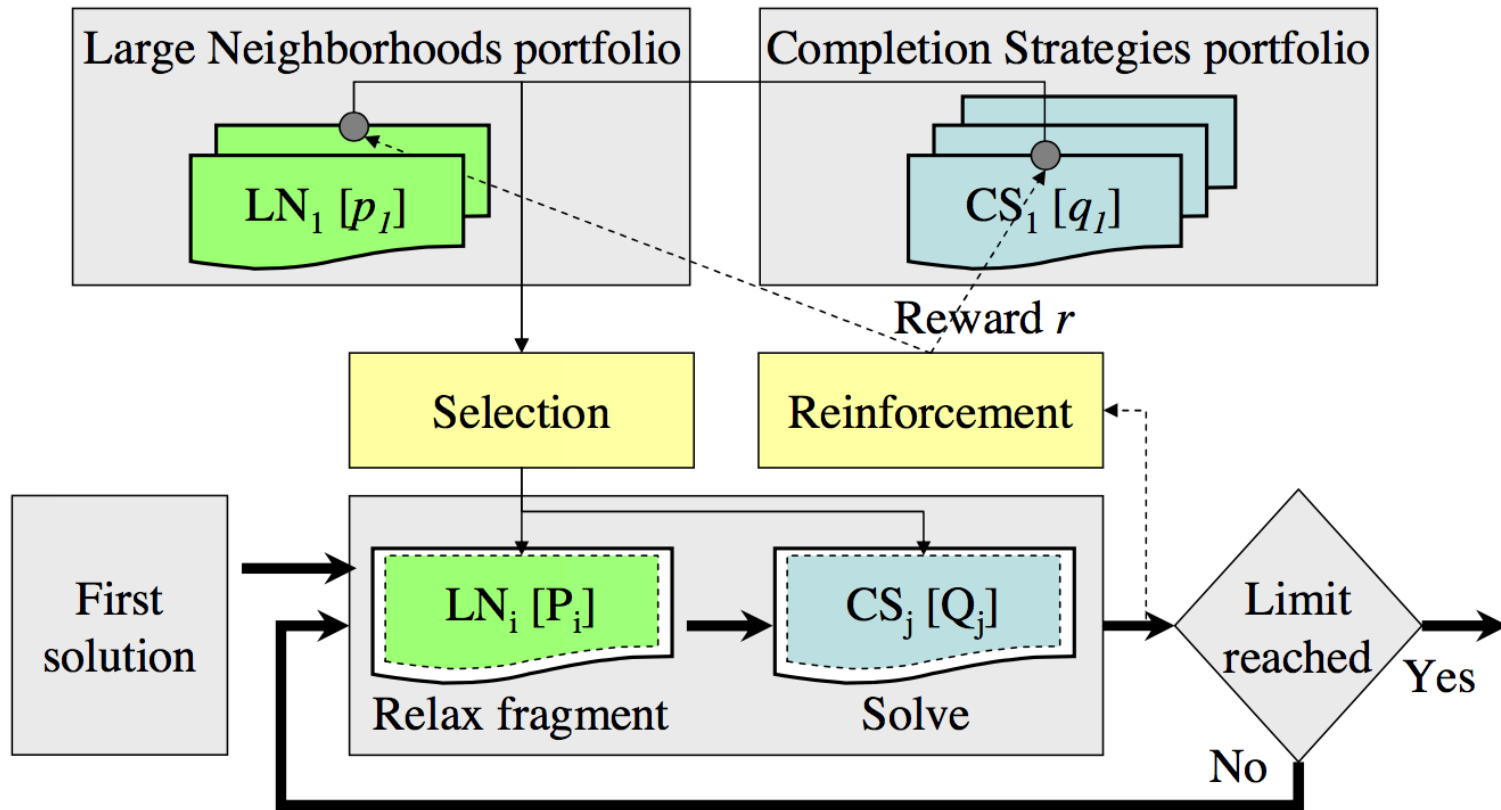
Under the hood: Large Neighborhood Search

- Iterative improvement method:
 1. Start with an existing solution (produced using some heuristics + classical CP search tree)
 2. Take part of the solution (fragment) and relax it. Fix the structure of the rest (but no start/end values: notion of Partial Order Schedule)
 3. Find (improved) solution using a limited search tree



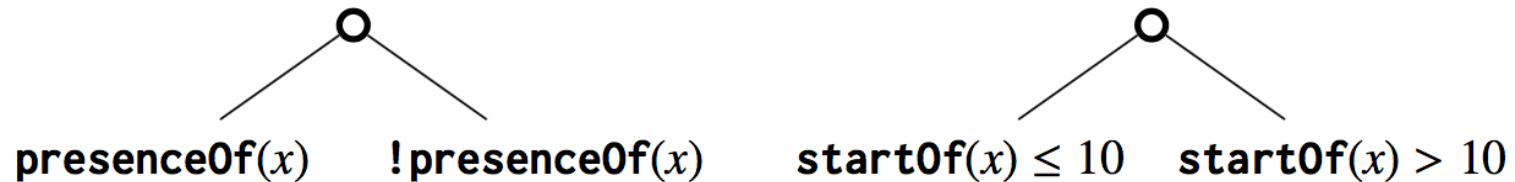
Under the hood: Large Neighborhood Search

- Uses portfolios and online reinforcement learning



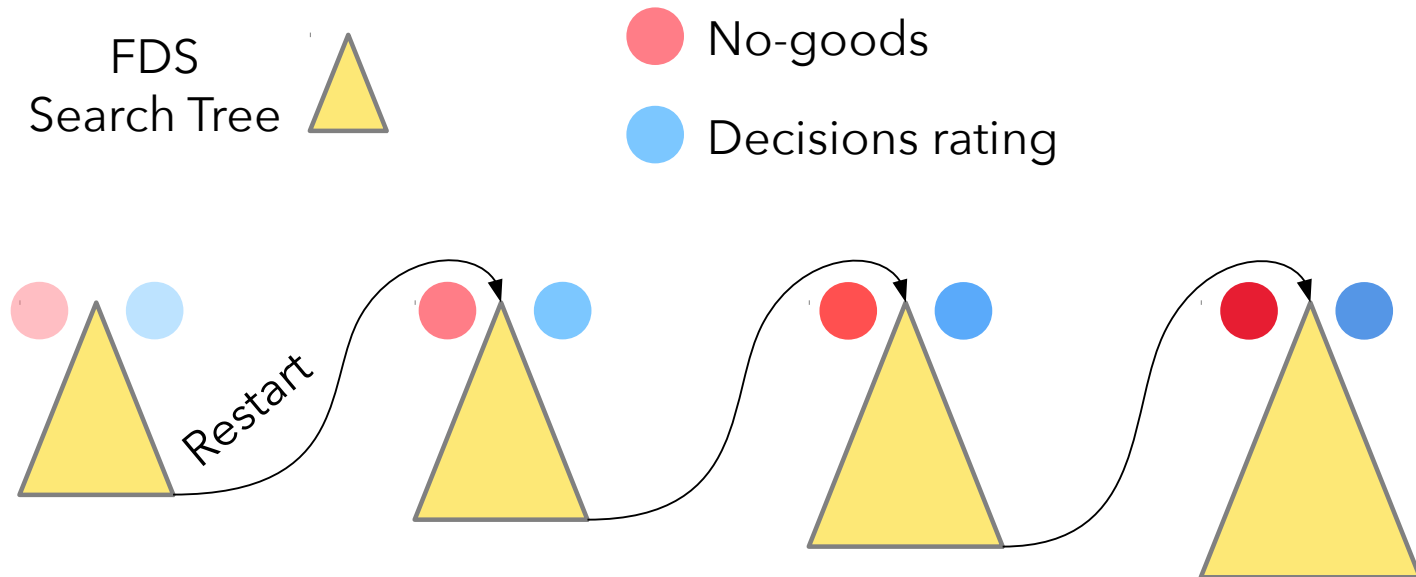
Under the hood: Failure-Directed Search

- FDS is automatically activated when:
 - The search space seems to be small enough, and
 - LNS has difficulties improving the current solution
- Assumption is that in these conditions:
 - There probably isn't any (better) solution
 - If there is one, it is very hard to find
 - It is necessary to explore the whole search space
- FDS uses periodic restarts and focuses on finding dead-ends (failures) in the search tree as quickly as possible
- FDS branches on ranges



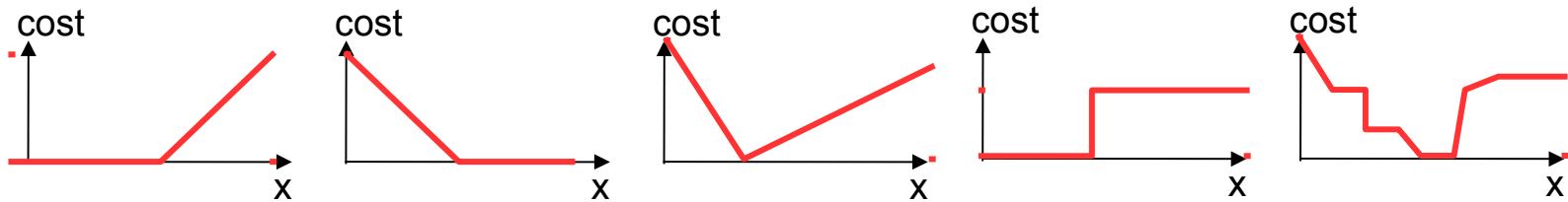
Under the hood: Failure-Directed Search

- Decisions are rated and the ones that often lead to infeasibility or strong domain reduction in the search are preferred: they are used earlier in the search during the next restarts
- FDS also records no-goods for avoiding exploring some identical part of the search space



Under the hood: Temporal Linear Relaxation

- Traditionally, early/tardy problems are challenging for CP-based tools as they miss a good global view of the cost



- Approach:
 - Automatically use CPLEX's LP solver to provide a solution to a relaxed version of the problem
 - Use the LP solution to guide heuristics. Start an operation as close as possible to the time proposed by LP solution

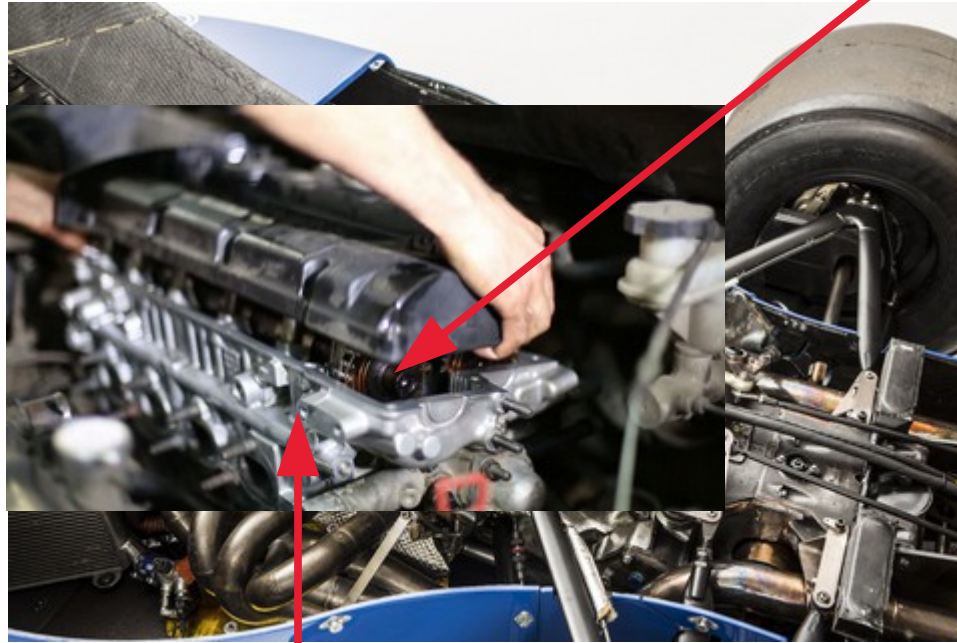
Under the hood: Temporal Linear Relaxation

- What is linearized?
 - Precedences
 - Optionality, logical constraints on optional intervals
 - Alternative and Span constraints
 - Cost function terms which are functions of start/end times

- Detailed description in P. Laborie, J. Rogerie. *Temporal Linear Relaxation in IBM ILOG CP Optimizer*. *Journal of Scheduling* 19(4), 391-400 (2016)

Under the hood: CP Optimizer is open (C++)

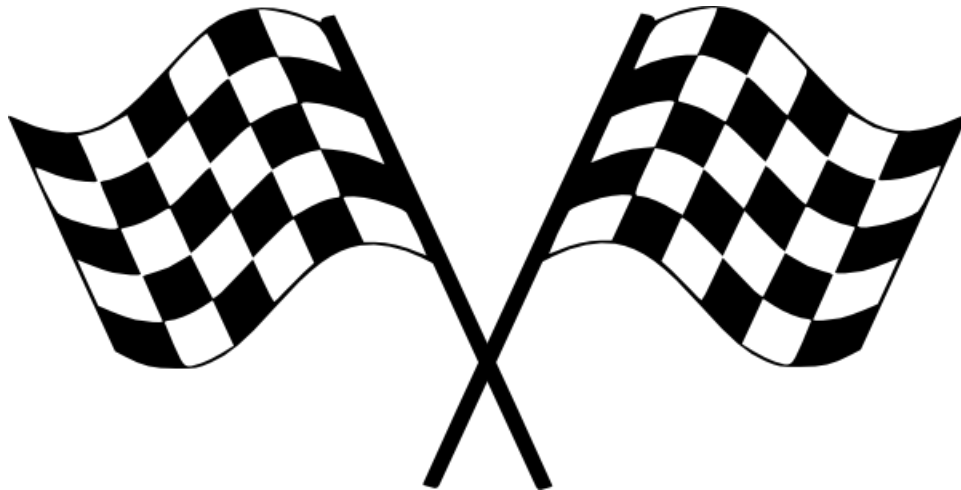
User-defined
Constraints and
Propagation



User-defined
Search

Conclusion

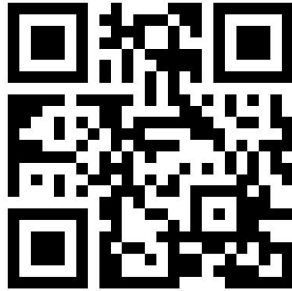
- The messages I tried to convey:
 - CP Optimizer is **easy to try**
 - CP Optimizer is **easy to learn**
 - CP Optimizer is **easy to use**
 - CP Optimizer is **powerful** for solving scheduling problems
 - CP Optimizer is **free** for students, teachers and researchers
 - CP Optimizer is **fun** !



Useful links

- In order to download CP Optimizer (together with OPL and CPLEX), please follow instructions on these pages:

Researchers,
teachers,
university staff



http://ibm.biz/COS_Faculty

Students



http://ibm.biz/COS_Student




- CP Optimizer forums:

[Forum Directory](#) > [IBM ILOG](#) > [IBM ILOG Optimization](#) > [Constraint Programming](#)

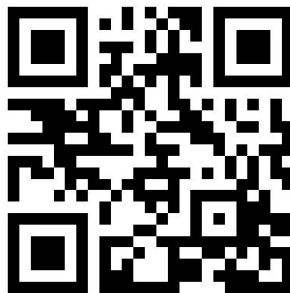
Category: Constraint Programming

1-3 of 3

[Previous](#) | [Next](#)

Forums	Topics/Messages	Latest Post
 OPL using CP Optimizer For topics on OPL modeling, OPL APIs, the IDE and scripting. Topics on CP Optimizer algorithms, parameters, error messa...	473/2044	Today 4:49 PM AlexCPLEX
 CP Optimizer For topics on CP Optimizers algorithms, parameters, tuning, etc. For topics on OPL modeling, go to the "OPL using CP Opti...	572/2393	Today 9:37 AM PhilippeLaborie
 Constraint Programming - General Topics that are not product-specific, related to modeling and solving CP problems, including best...	282/1084	Apr 29 Dr.Meng

CP Optimization
forums



http://ibm.biz/COS_Forums