

Modelling & Solving Detailed Scheduling Problems with IBM ILOG CP Optimizer

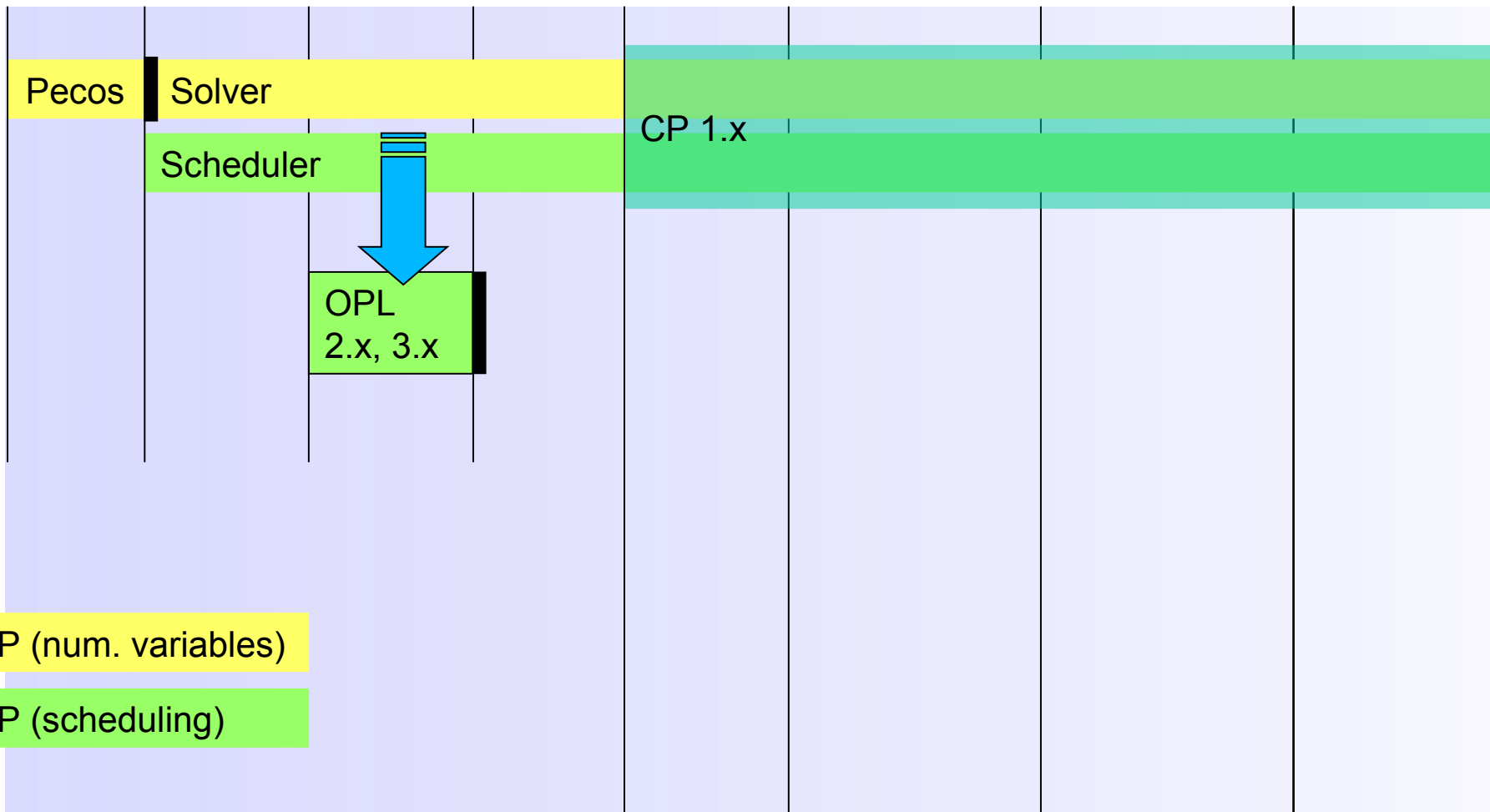
Séminaire LAAS
Apr 24, 2009

Philippe Laborie
laborie@fr.ibm.com

- Overview:
 - Context
 - Modelling scheduling problems with CPO
 - Examples
 1. Flowshop with earliness/tardiness costs
 2. Oversubscribed satellite communication scheduling
 3. Personal tasks scheduling with preferences
 - Solving scheduling problems with CPO

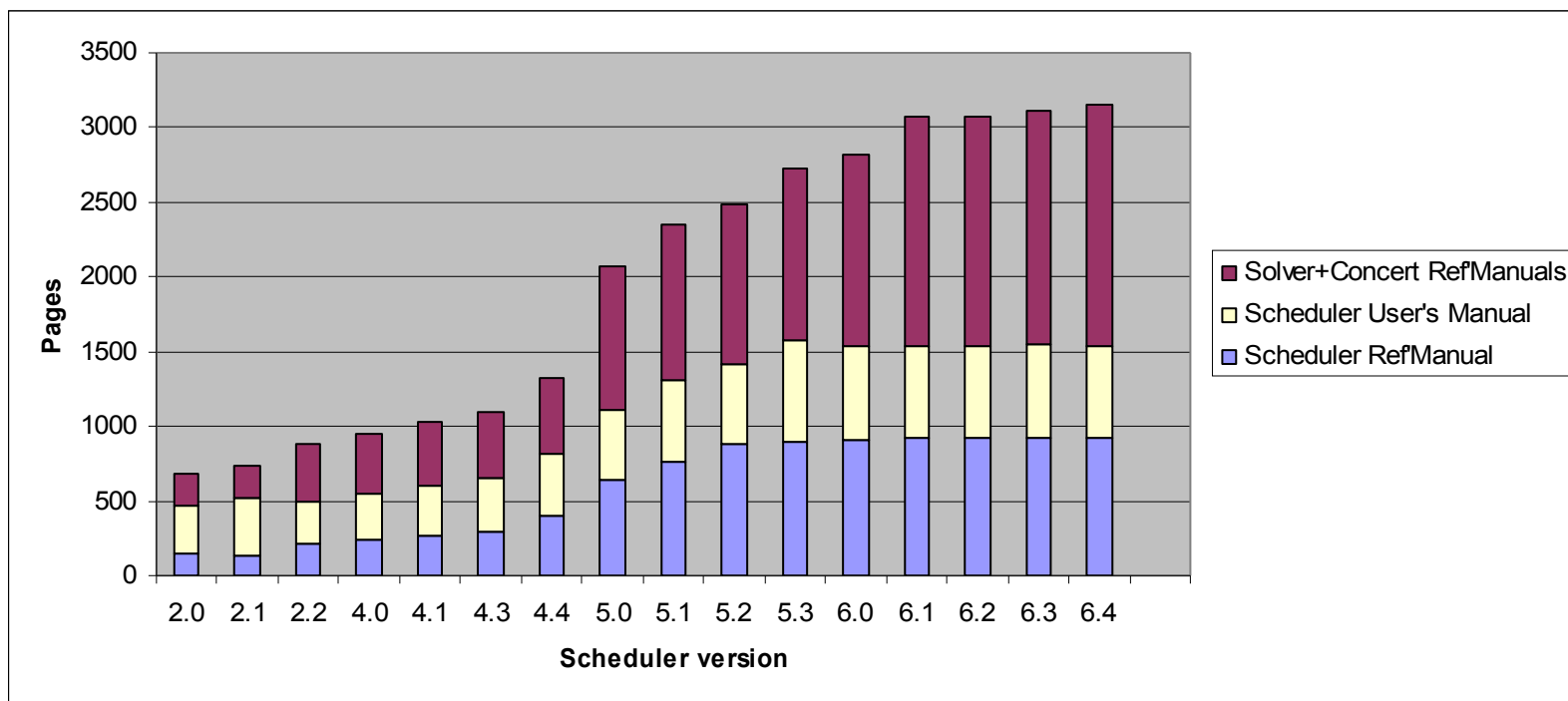
CP at ILOG: an historical overview

1991 1994 1999 2003 2005 May 2007 May 2008 Today



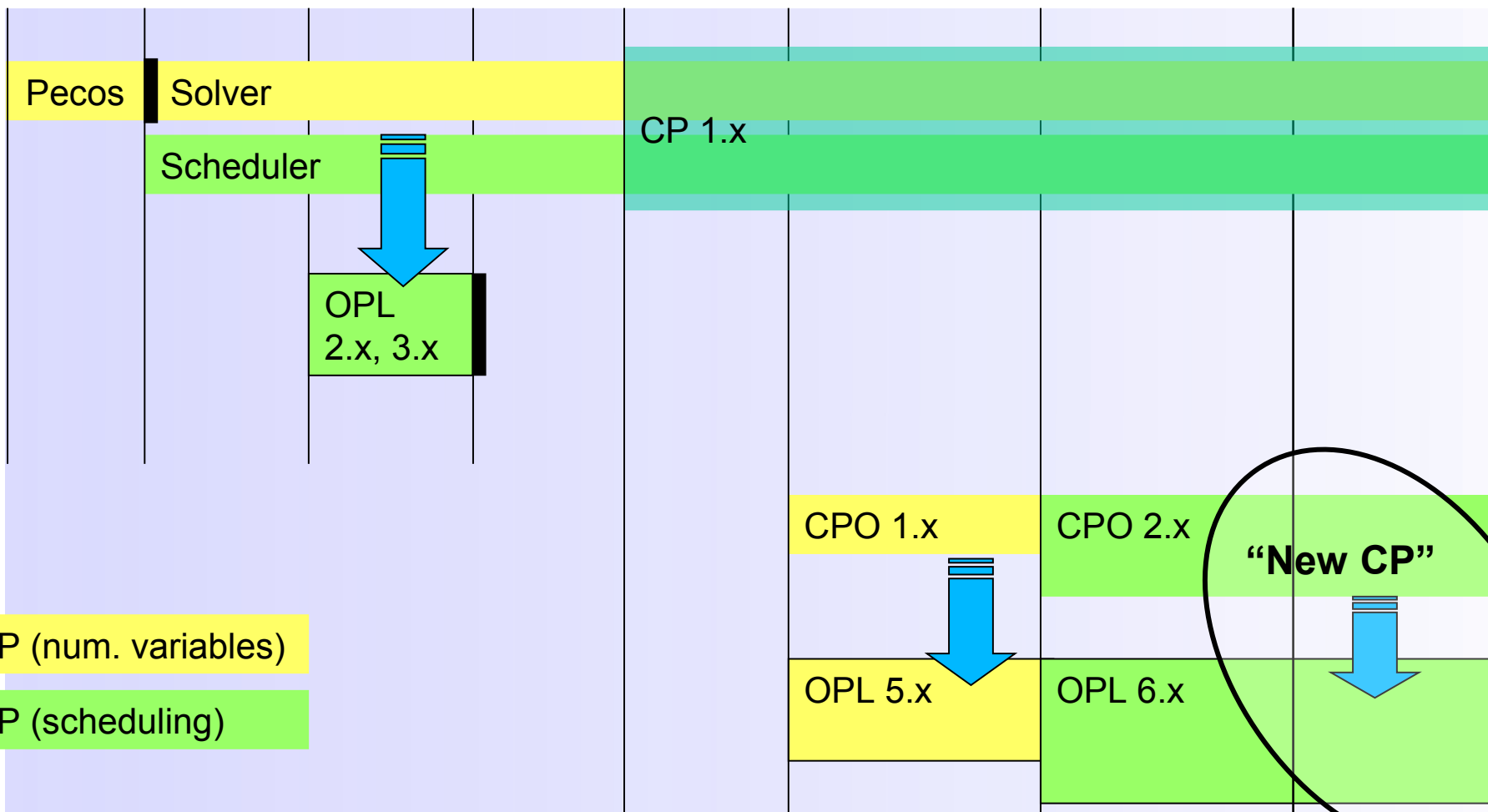
- ILOG Scheduler 1994→2006:
 - Many industrial successes, BUT
 - The product has become very complex to use: to develop a scheduling application, a customer must master:
 - C++
 - Constraint Programming (if writing new constraints)
 - Non-deterministic programming (search tree exploration)
 - Scheduling theory (for writing efficient search, for deciding which propagation algorithm will be useful, ...)
 - The larger and larger API of Scheduler

- ILOG Scheduler 1994→2006:
 - Many industrial successes, BUT
 - The product has become very complex to use



CP at ILOG: an historical overview

1991 1994 1999 2003 2005 May 2007 May 2008 Today



CP (num. variables)

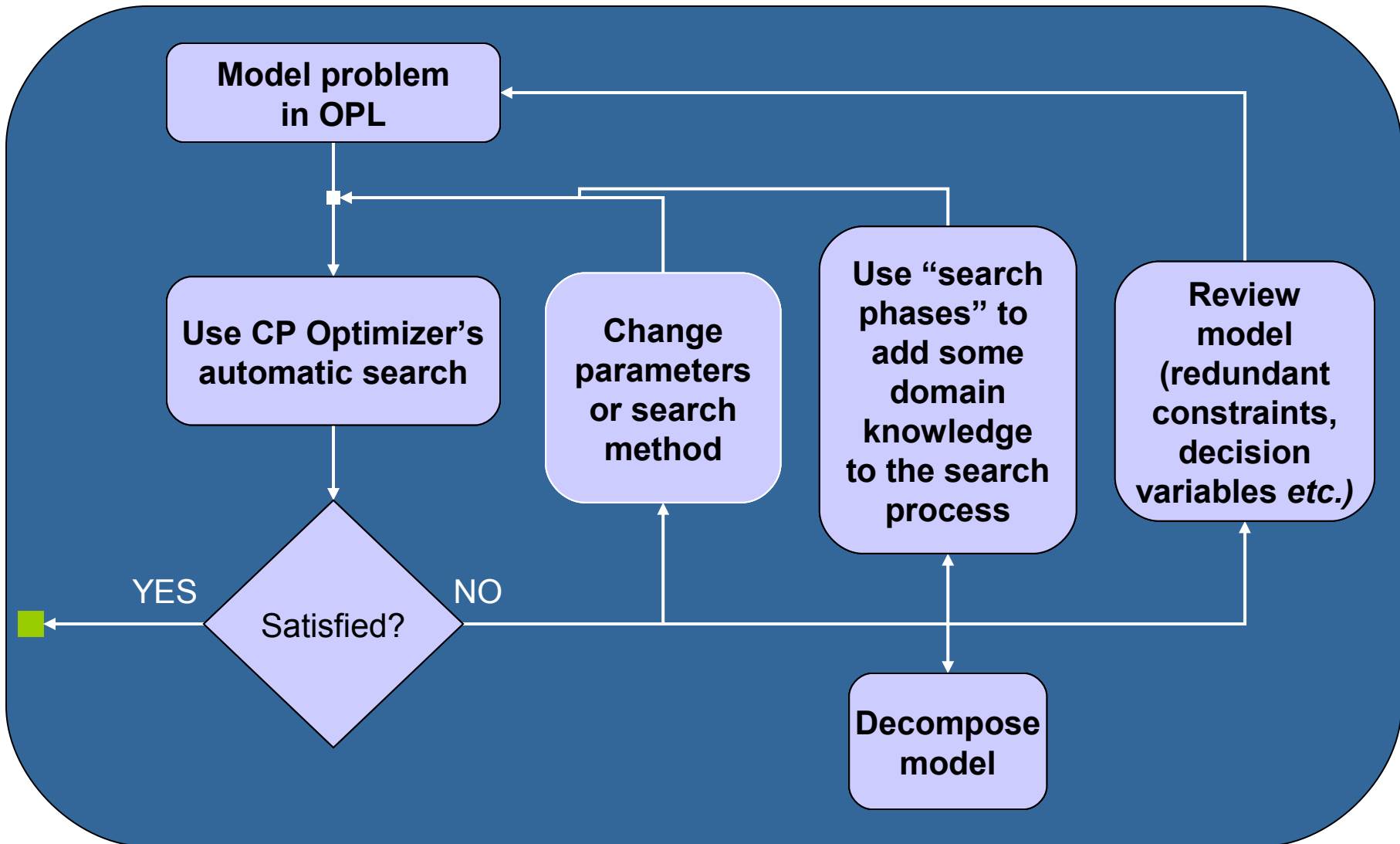
CP (scheduling)

What is IBM ILOG CP Optimizer?



- A Constraint Programming engine for combinatorial and detailed scheduling problems
- Roughly covers problem classes addressed by Solver & Scheduler
- ☞ Model&Solve paradigm (a-la CPLEX)
 - Flexible modeling language
 - But smaller than Solver and Scheduler
 - Powerful automatic search procedure
 - User can influence search based on their knowledge of the problem

Typical use of CP Optimizer



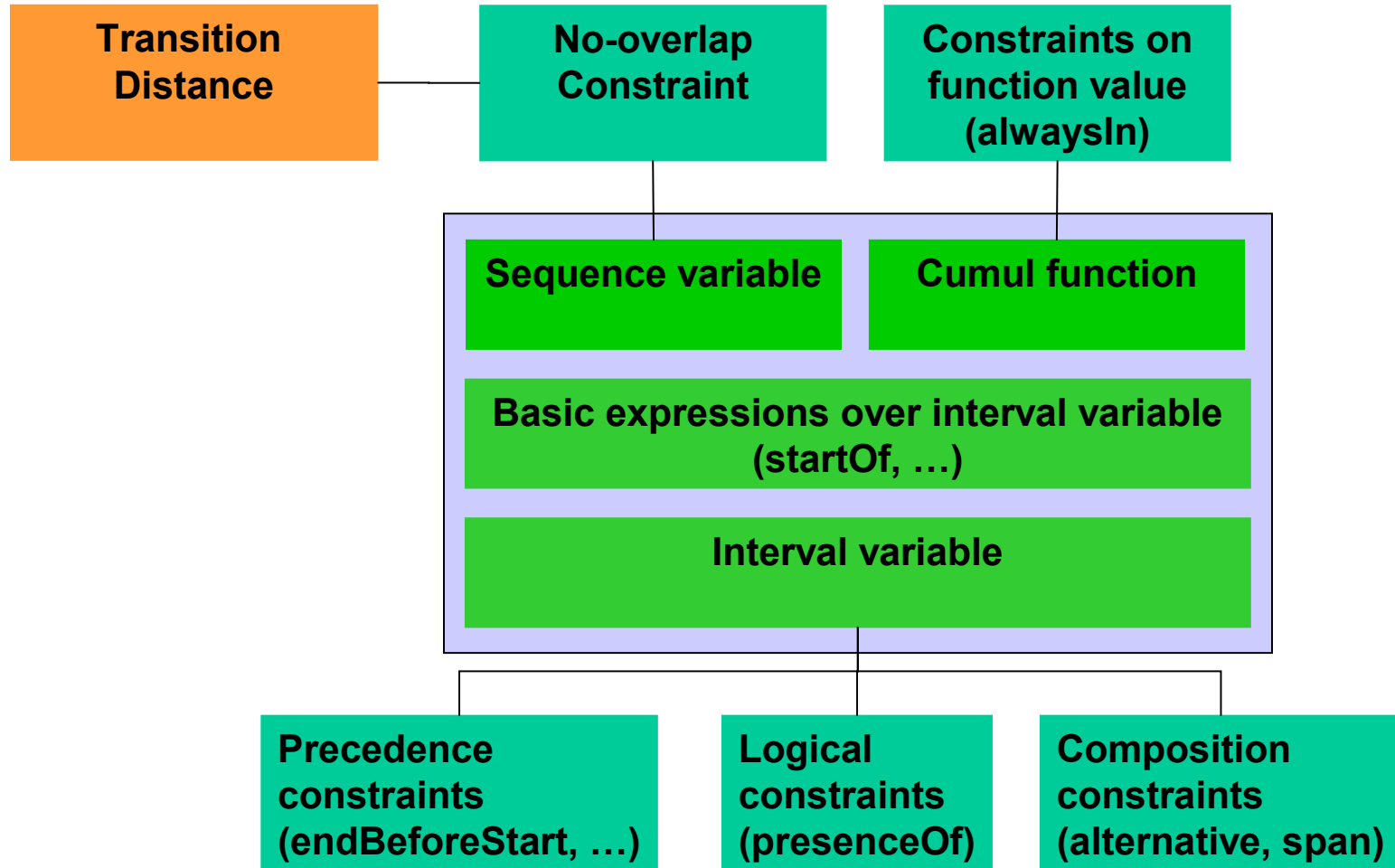
- CP Optimizer is available through the following interfaces:
 - OPL
 - C++: native interface
 - Java: wrapping of the C++ engine
 - .NET: wrapping of the C++ engine
- And on the following platforms:
 - 32-bit
 - Windows, Debian, Solaris, AIX, Darwin (Mac OS X)
 - 64-bit
 - Windows, Debian, Solaris, AIX

Language for detailed scheduling

Variable/expression

Constraint

Data structure

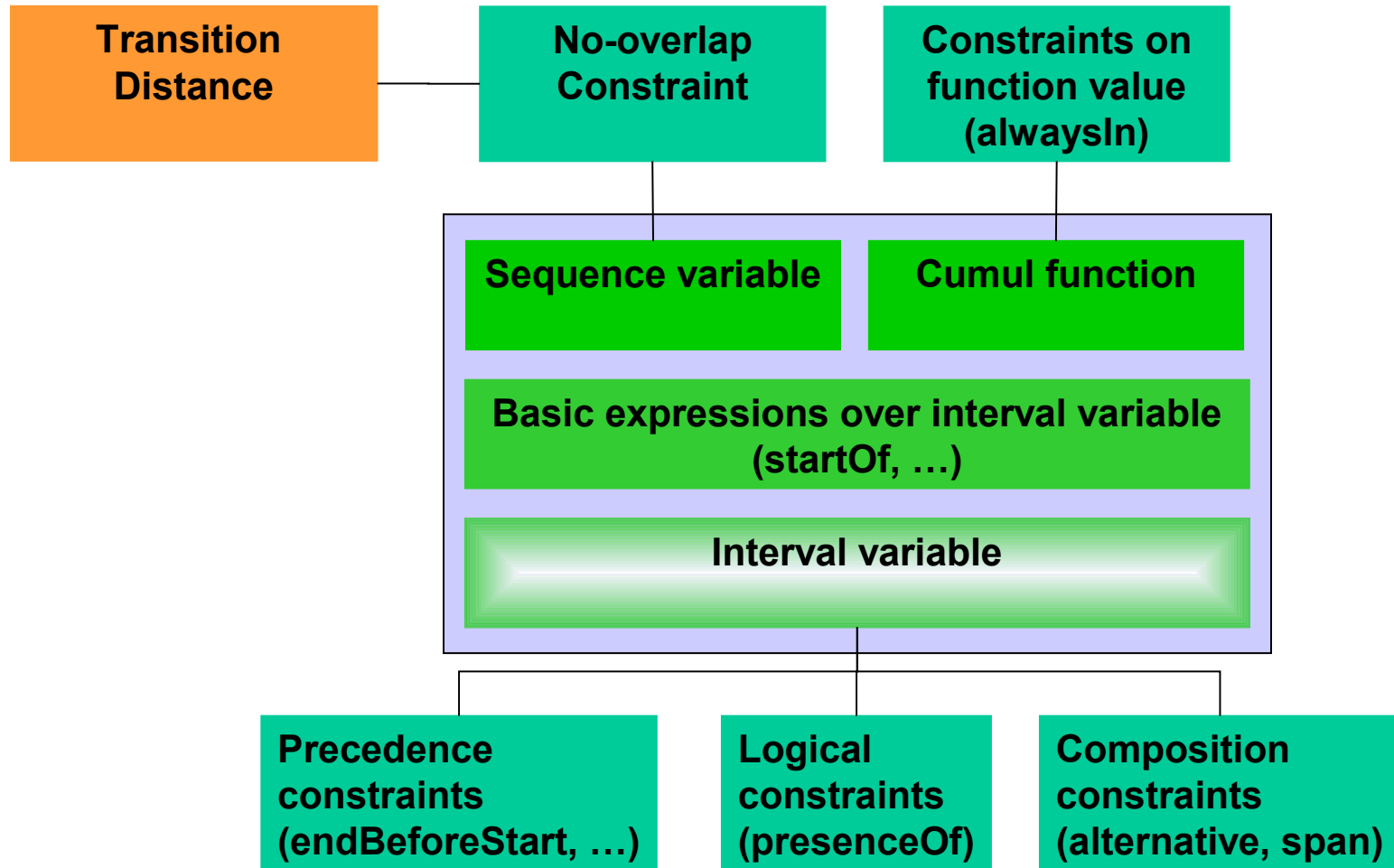


Language for detailed scheduling

Variable/expression

Constraint

Data structure



- A new type of first class citizen decision variable is introduced: **interval variable**
- Models time intervals whose end-points (start/end) are decisions of the problem
 - A production order, a recipe in a production order, an operation in a recipe
 - A sub-project in a project, a task in a sub-project
 - A batch of operations
 - The setup of a tool on a machine
 - The moving of an item by a transportation device
 - The utilization interval of a machine
 - The filling or emptying of a tank

- A new type of first class citizen decision variable is introduced: **interval variable**
- An interval variable can be **optional** meaning that it is a decision to have it **present** or **absent** in a solution
 - Unperformed tasks and optional sub-projects
 - Operations that can be processed in different temporal modes (e.g. series or parallel), left unperformed or externalized
 - Alternative modes or recipes for processing an order, each mode specifying a particular combination of operational resources

- A new type of first class citizen decision variable is introduced: **interval variable**

- Domain of values for an interval variable a :

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

Absent interval

Interval of integers

- Notations: let a be a **fixed** interval variable
 - If $a=[s,e)$ (a is **present**), we denote:
 $x(a)=\text{true}$, $s(a)=s$, $e(a)=e$, $l(a)=e-s$
 $s(a)$ is the **start** of a
 $e(a)$ is the **end** of a
 $l(a)$ is the **length** of a
 - If $a=\perp$ (a is **absent**), we denote:
 $x(a)=\text{false}$
In this case $s(a)$, $e(a)$ and $l(a)$ are meaningless

■ Interval variable declaration in OPL

```
dvar interval a
  [optional]           Specifies interval as optional
  [in smin..emax]       Start min, end max if present
  [size szmin..szmax];  Size min, size max if present
```

For now, we assume size and length are identical concepts

smin, emax: integers

szmin, szmax: non-negative integers

By default:

interval is present, smin=0, emax=+ ∞ , szmin=0, szmax=+ ∞

■ Examples:

dvar interval a;

dvar interval b optional;

dvar interval c in 0..1000 size 10;

$\text{Dom}(c) = \{ [0,10), [1,11), \dots, [990,1000) \}$

dvar interval d optional in 1..3 size 1..2;

$\text{Dom}(d) = \{\perp\} \cup \{ [1,2), [2,3), [1,3) \}$

dvar interval e optional in 0..1 size 0..1;

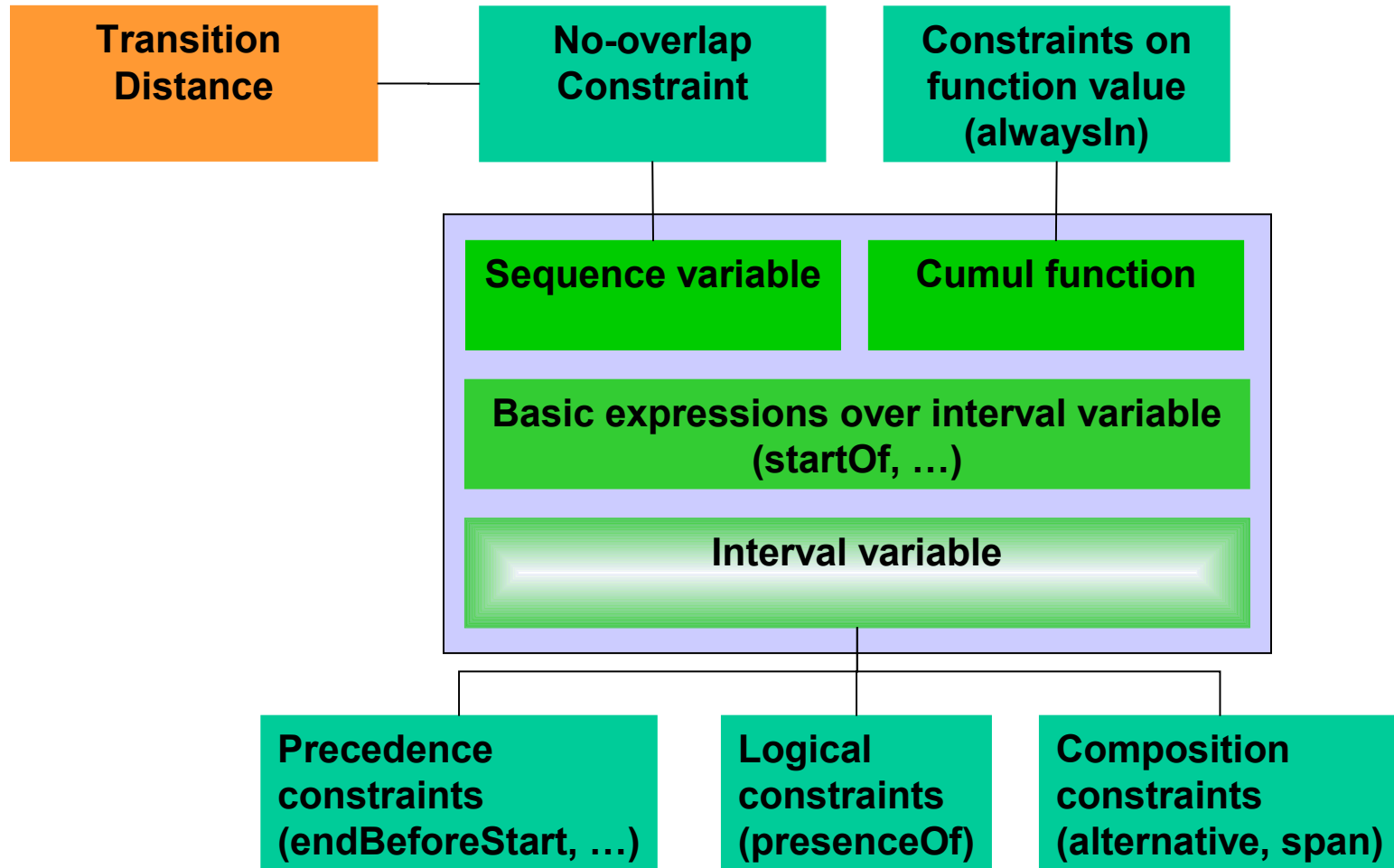
$\text{Dom}(e) = \{\perp\} \cup \{ [0,0), [1,1), [0,1) \}$

Language for detailed scheduling

Variable/expression

Constraint

Data structure

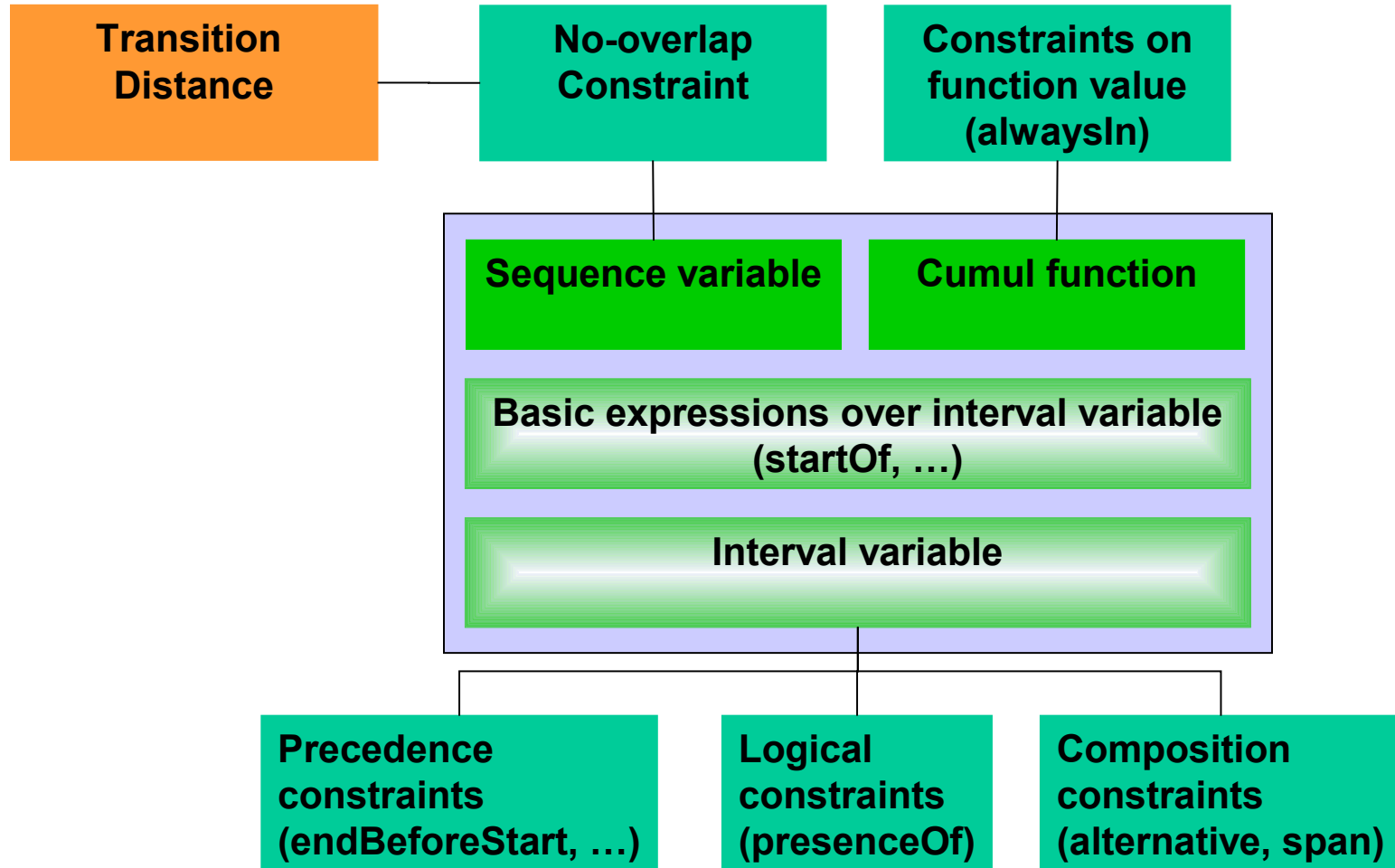


Language for detailed scheduling

Variable/expression

Constraint

Data structure



- Integer expressions to get the start/end/length/size of an interval variable
- OPL syntax:

```
dexpr int startOf(dvar interval a, int v=0);
```

```
dexpr int endOf(dvar interval a, int v=0);
```

```
dexpr int sizeOf(dvar interval a, int v=0);
```

```
dexpr int lengthOf(dvar interval a, int v=0);
```

- The integer v is the value of the expression if interval a is absent (default: 0)

- These expressions can be mixed with other numerical expressions in CP Optimizer:

- General expressions:

`x*y, k*x, x+y, x+k, x-y, abs(x),
min(x,y), max(x,y), ...`

- Integer expressions:

`x div y, x mod y, ...`

- Floating point expressions:

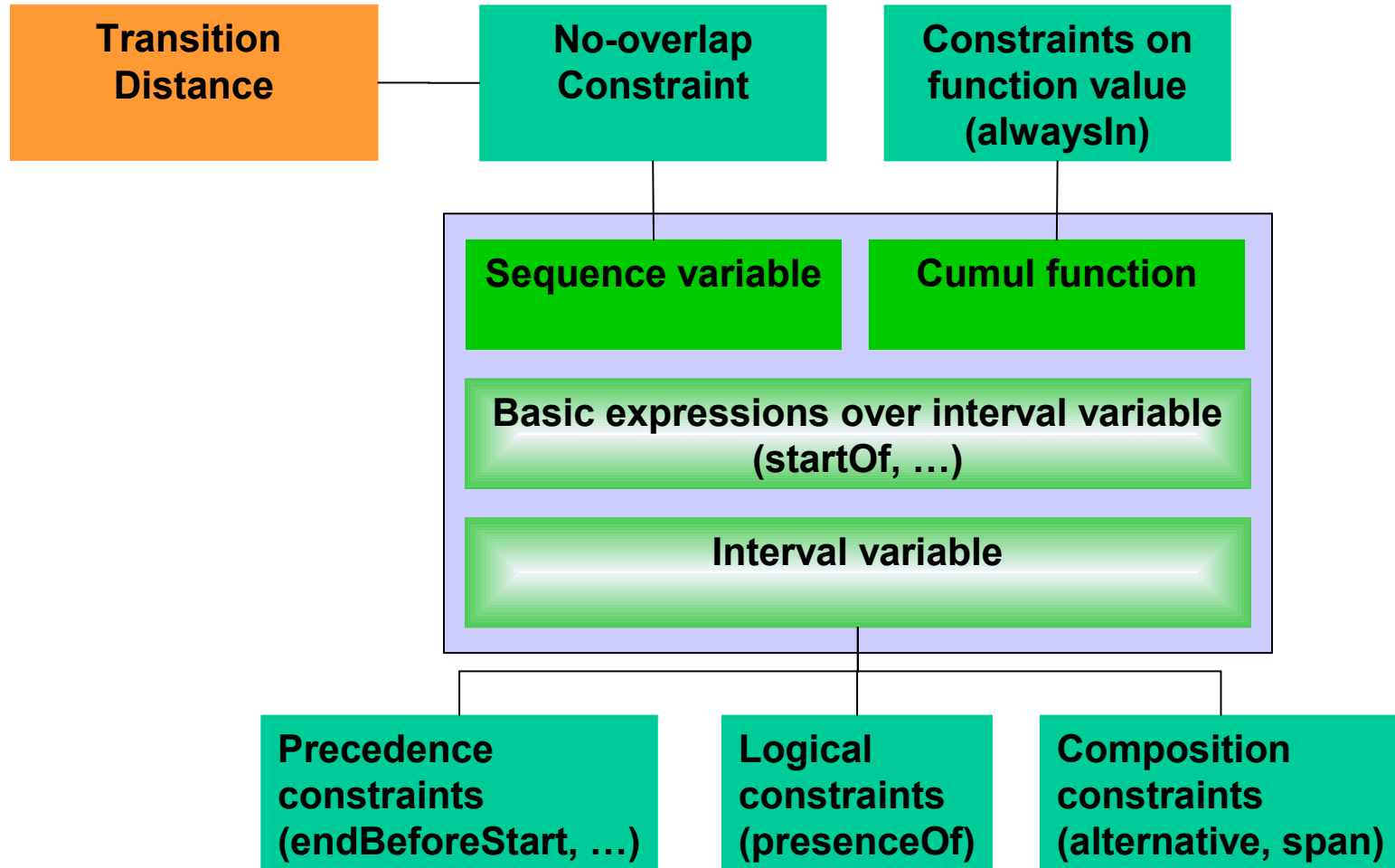
`ceil(x), floor(x), frac(x), x/y,
sqrt(x), exp(x), log(x), pow(x,y), ...`

Language for detailed scheduling

Variable/expression

Constraint

Data structure

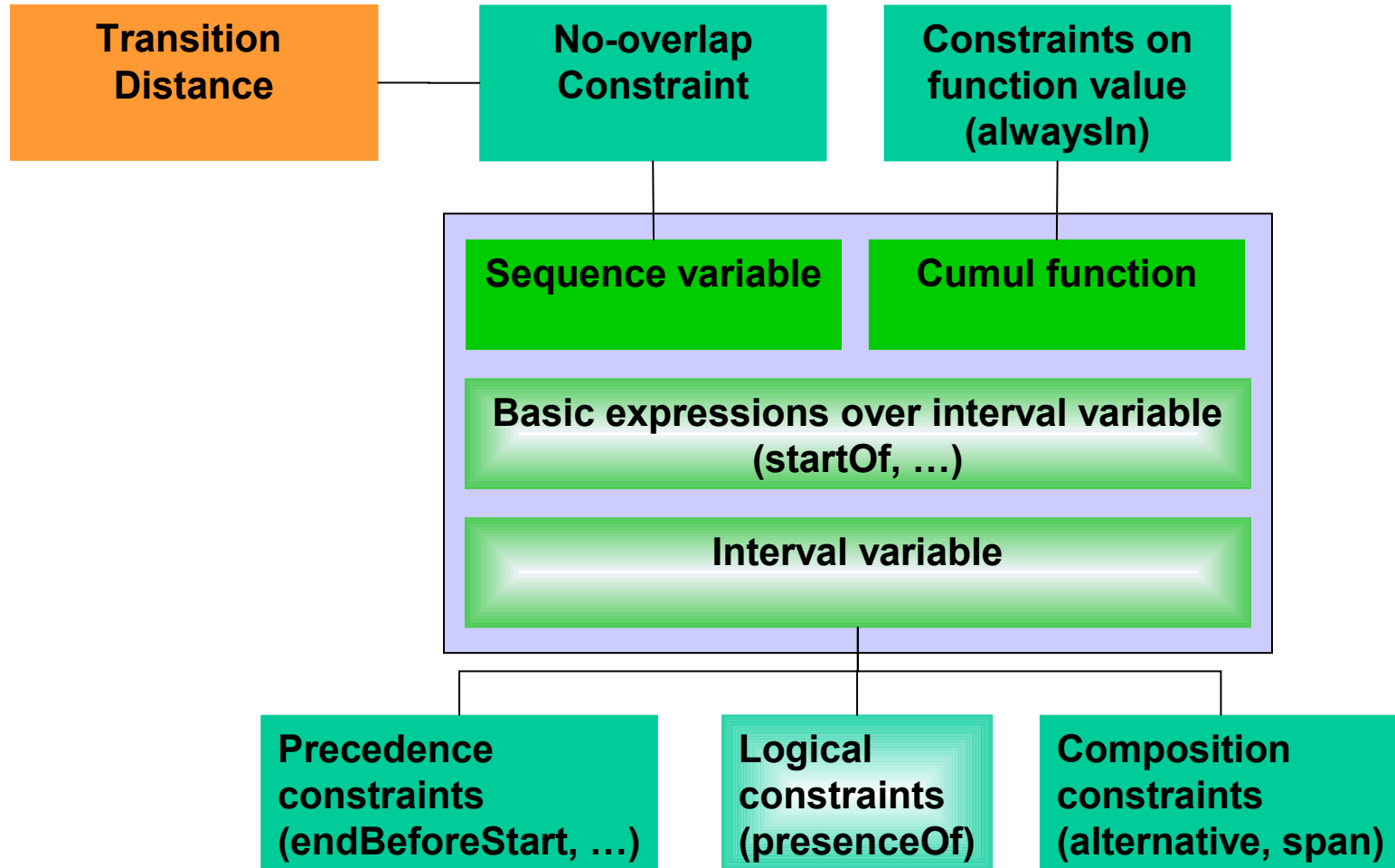


Language for detailed scheduling

Variable/expression

Constraint

Data structure



- Unary constraint on interval variable presence
- OPL Syntax:

```
presenceOf(dvar interval a);
```

- Can be composed (meta-constraints):

```
presenceOf(a) => presenceOf(b);
```

```
presenceOf(a) => !presenceOf(b);
```

```
presenceOf(a) || presenceOf(b);
```

- Can be casted as Boolean expression:

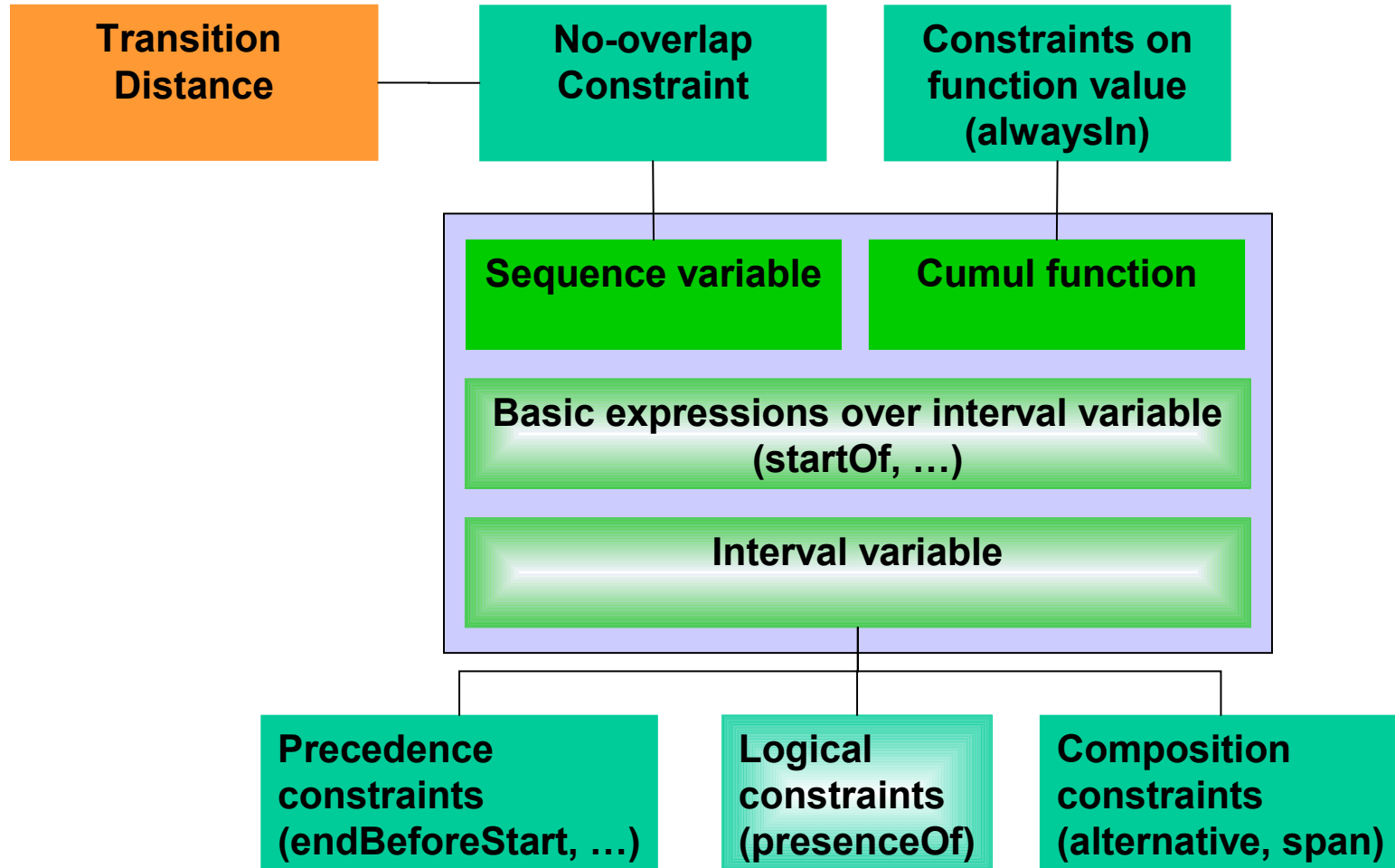
```
dexpr int nbPres = sum(i in 1..n) presenceOf(a[i]);
```

Language for detailed scheduling

Variable/expression

Constraint

Data structure

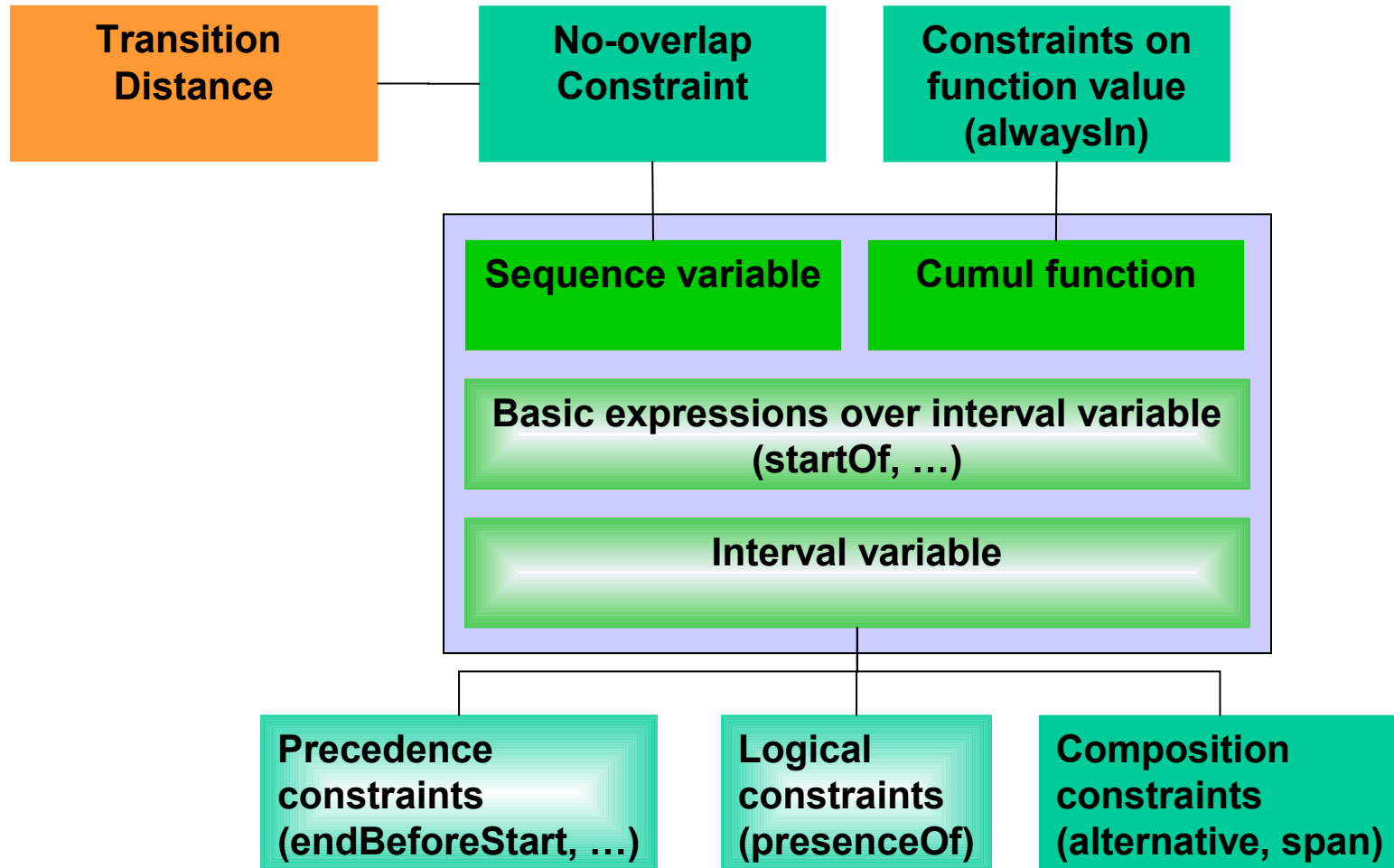


Language for detailed scheduling

Variable/expression

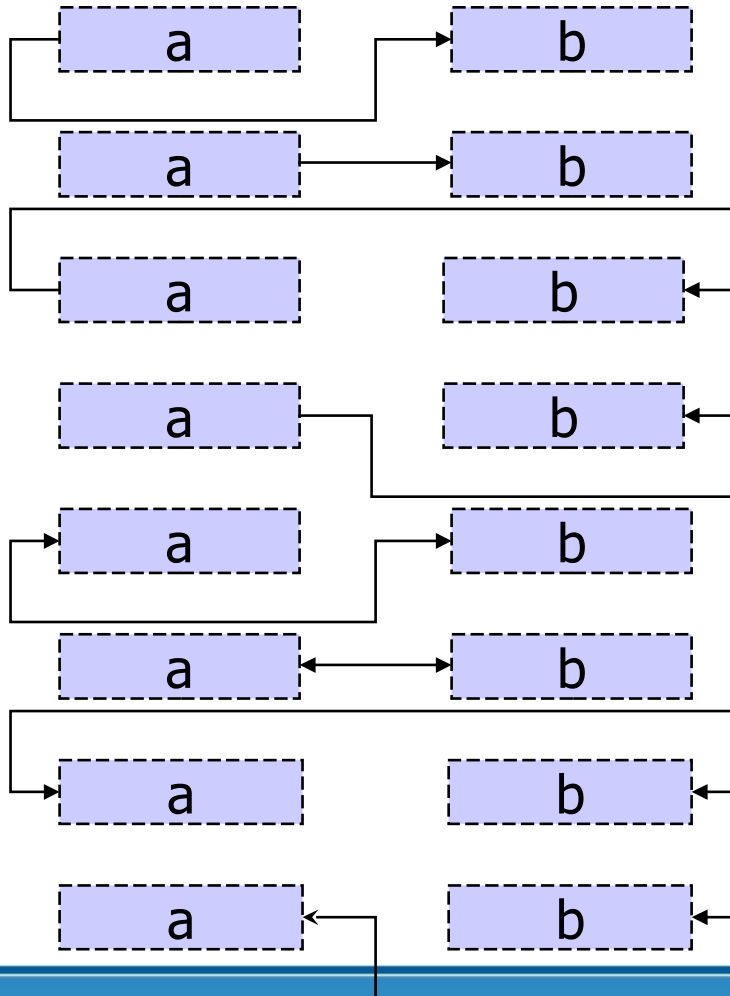
Constraint

Data structure



- Binary constraints on interval variables
- Classical precedence constraints of Constraint-Based Scheduling **but**
- Precedence Constraints definition $t_i + z \leq t_j$ is reified by optionality statuses
- Example:
 $\text{endBeforeStart}(a,b,z)$ means:
$$x(a) \wedge x(b) \Rightarrow e(a) + z \leq s(b)$$
- Precedence Constraints cannot be used in logical constraints

- Graphical conventions



startBeforeStart

endBeforeStart

startBeforeEnd

endBeforeEnd

startAtStart

endAtStart

startAtEnd

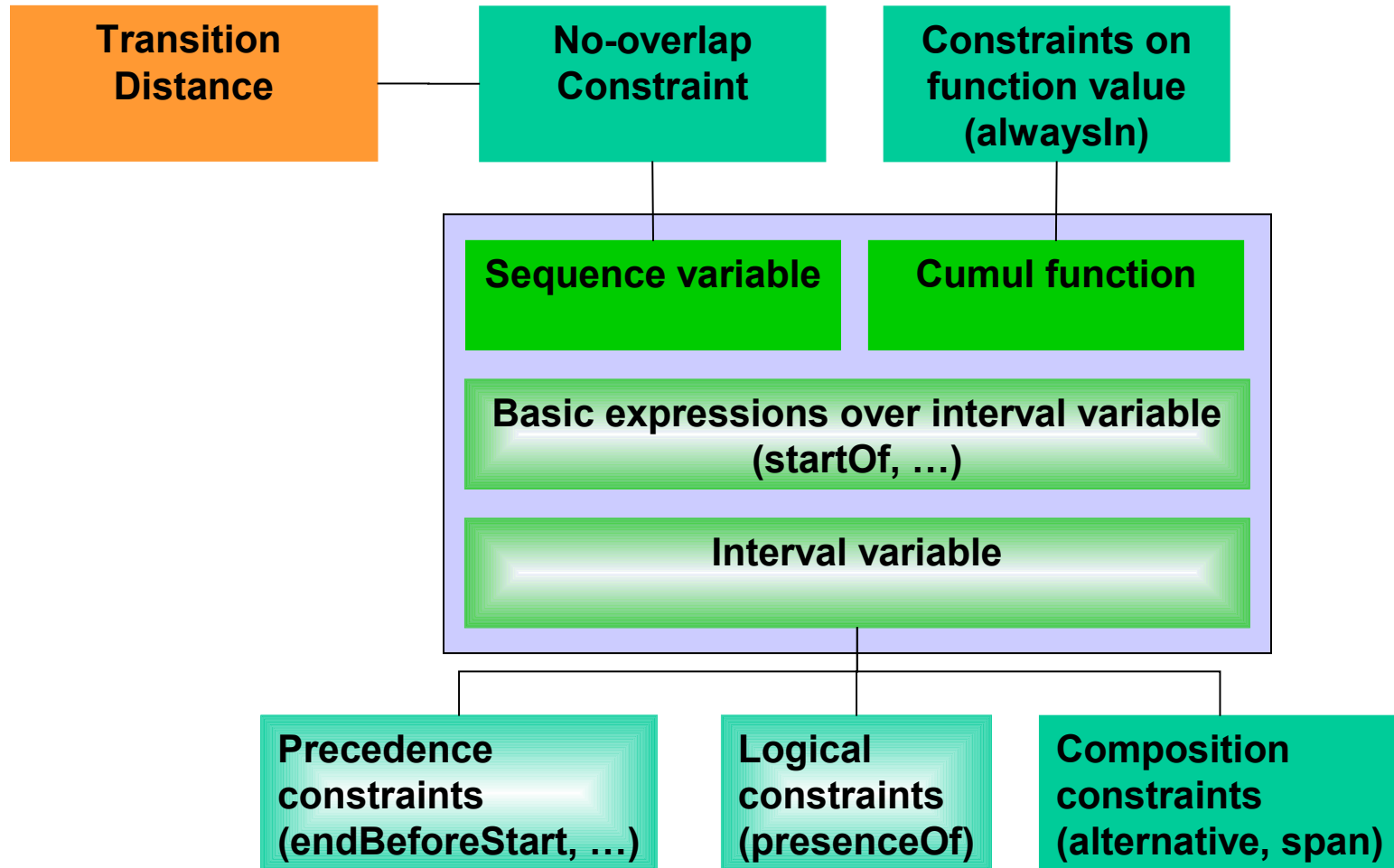
endAtEnd

Language for detailed scheduling

Variable/expression

Constraint

Data structure

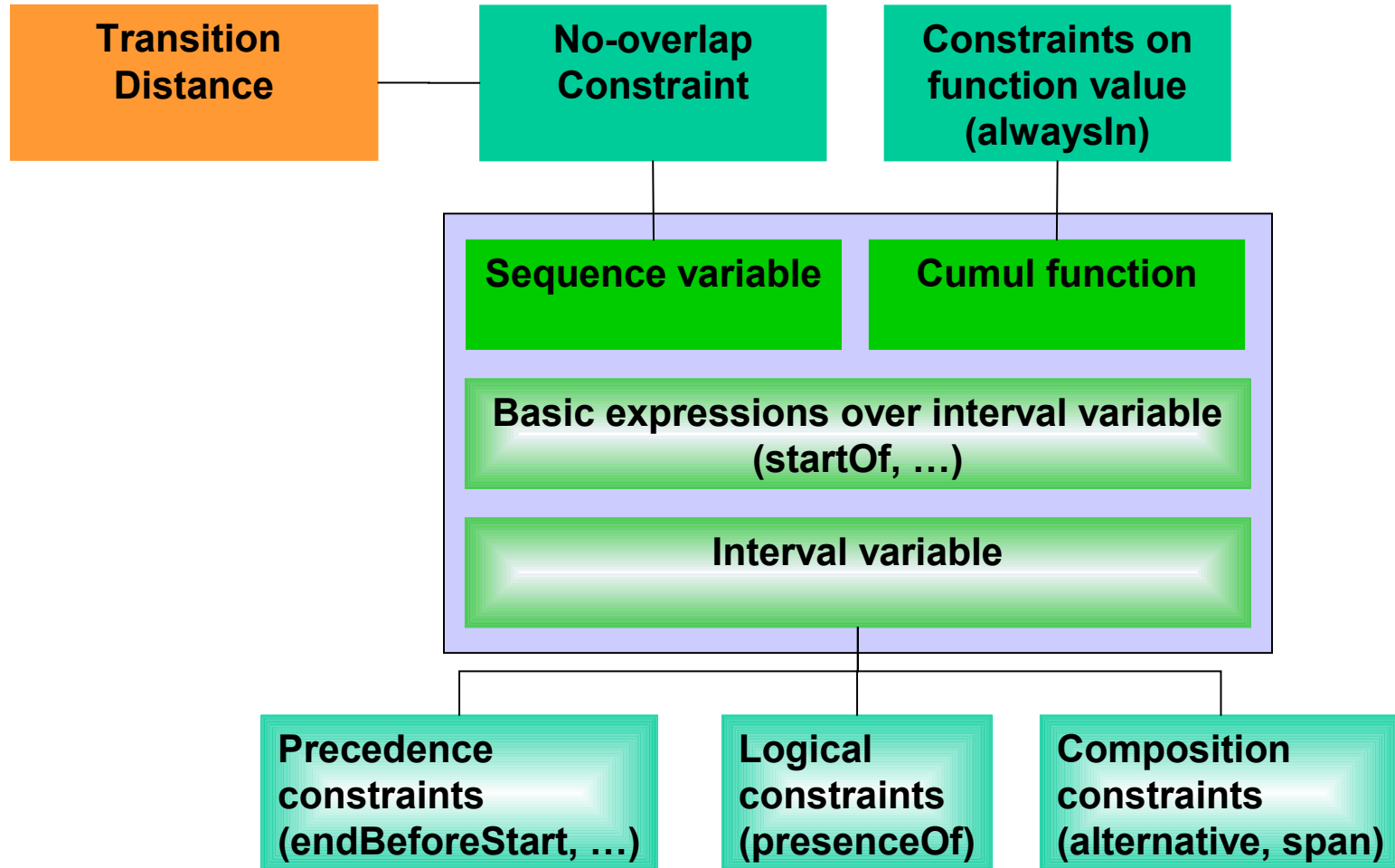


Language for detailed scheduling

Variable/expression

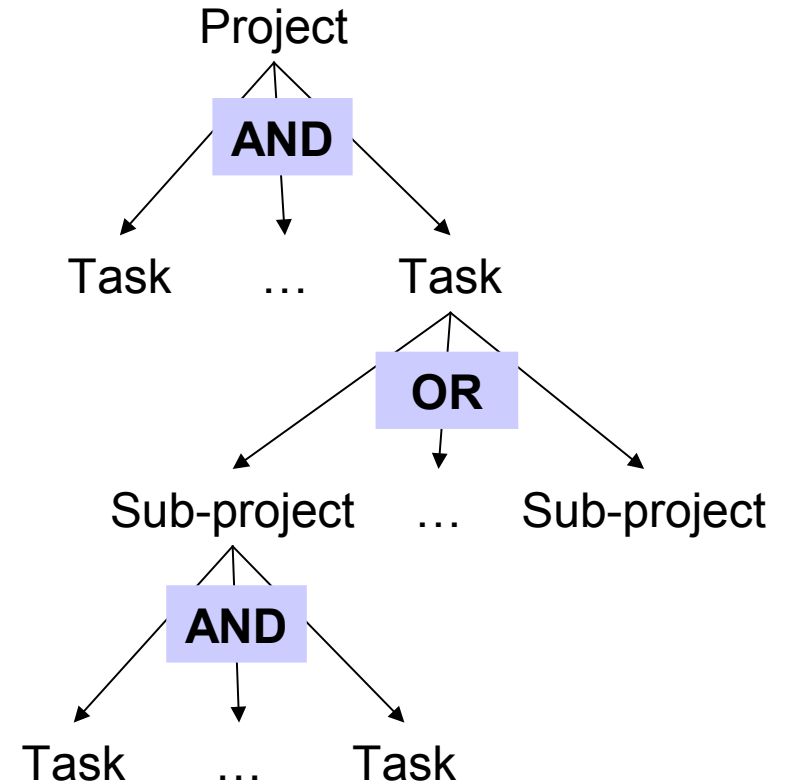
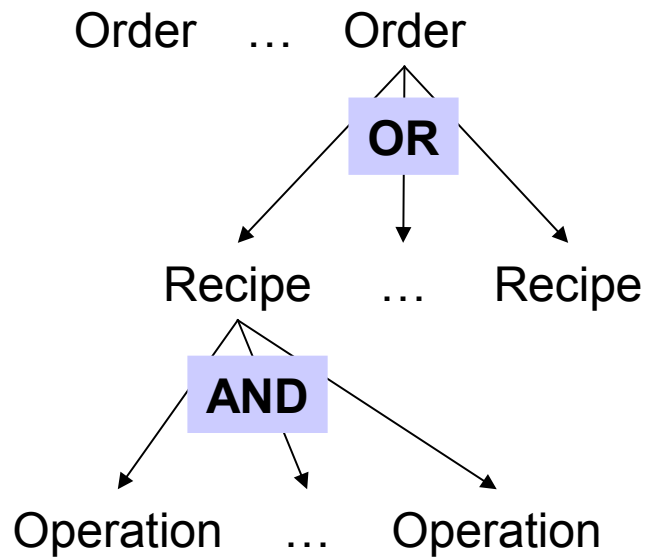
Constraint

Data structure

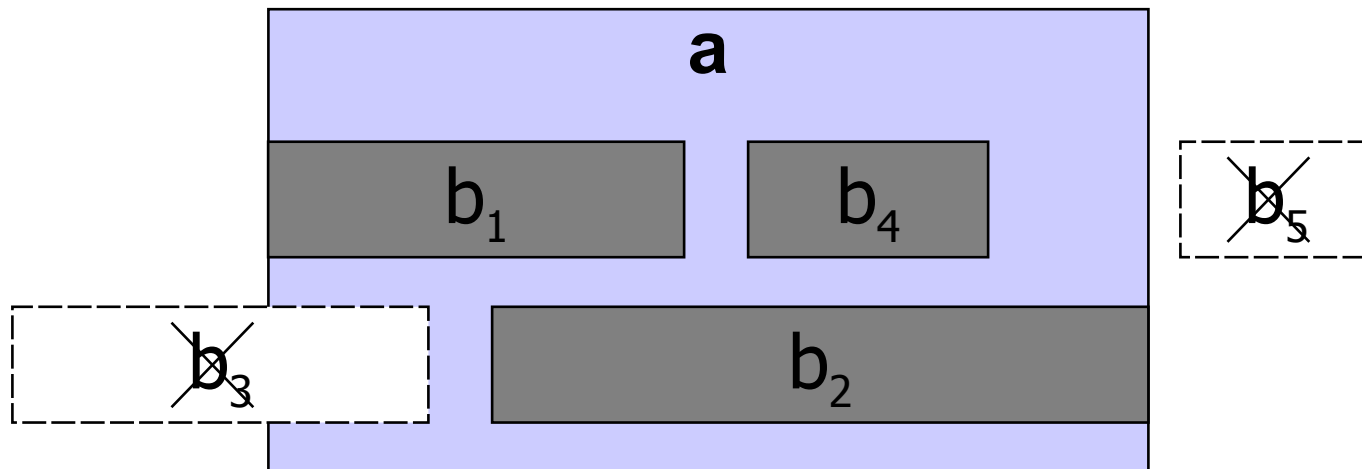


- Many scheduling problems are hierarchically organized as AND/OR trees:
 - AND nodes: a detailed description of how a high-level activity a decomposes into sub-activities $\{b_1, \dots, b_n\}$
 - OR nodes: a set of alternatives $\{b_1, \dots, b_n\}$ for executing an activity a
 - These nodes may represent optional parts of the schedule

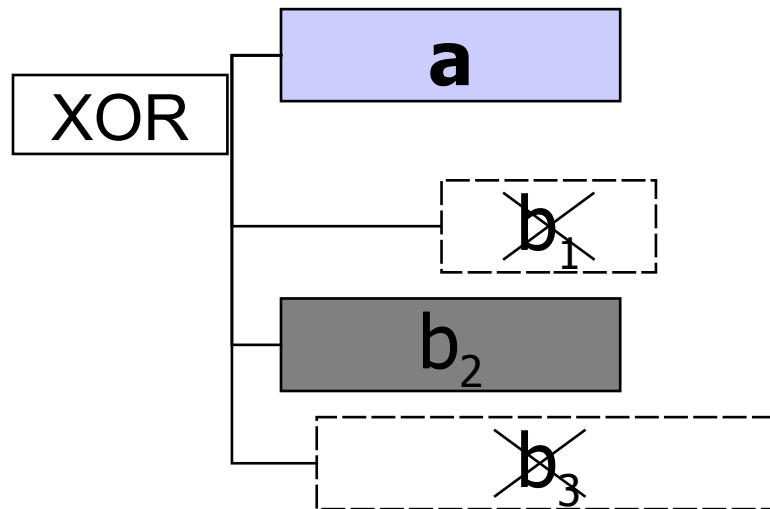
Composition constraints



- Span constraint $\text{span}(a, \{b_1, \dots, b_n\})$ means that if a is present, it spans all present intervals from $\{b_1, \dots, b_n\}$ that is, at least one of b_i support the start (resp. end) of a . a is absent if and only if all the b_i are absent.



- Alternative constraint **alternative**($a, \{b_1, \dots, b_n\}$) means that if a is present, then exactly one of the $\{b_1, \dots, b_n\}$ is present and synchronized with a . a is absent if and only if all the b_i are absent.



- OPL Syntax:

```
dvar interval a ...;  
dvar interval b[i in 1..n] ...;  
  
span( a, all(i in 1..n) b[i] );  
alternative( a, all(i in 1..n) b[i] );
```

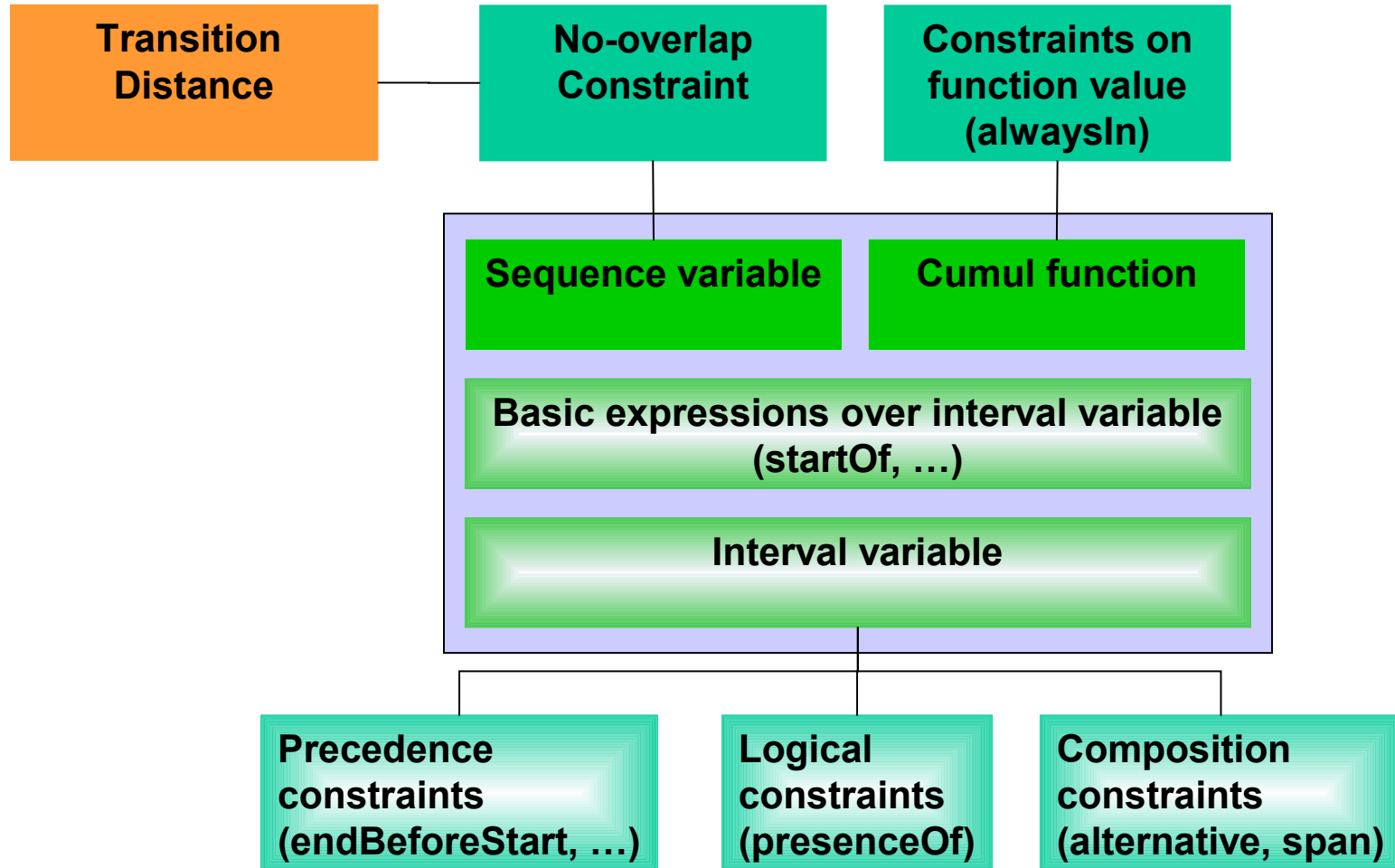
- Note that *a* can be an optional interval variable too
...

Language for detailed scheduling

Variable/expression

Constraint

Data structure

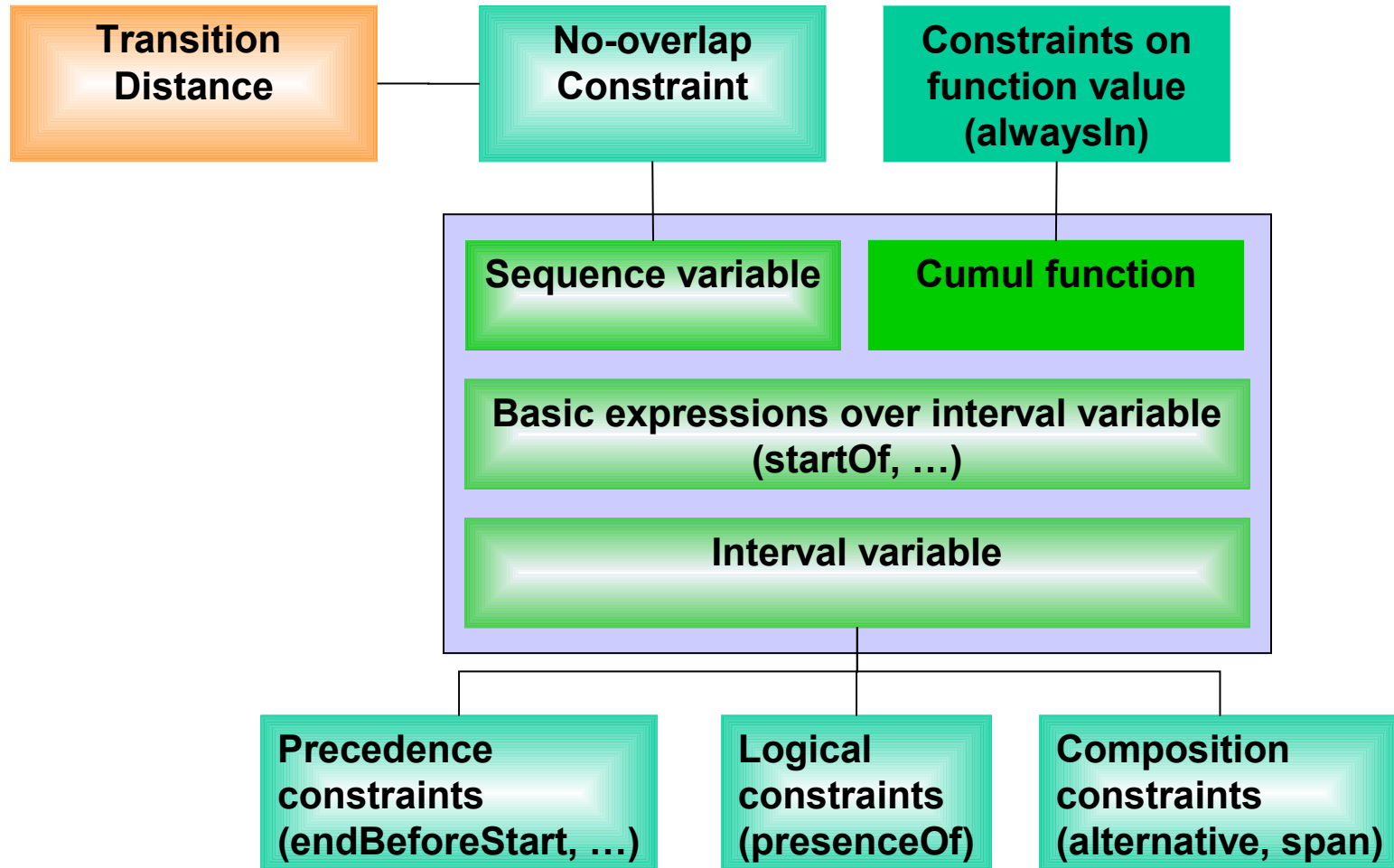


Language for detailed scheduling

Variable/expression

Constraint

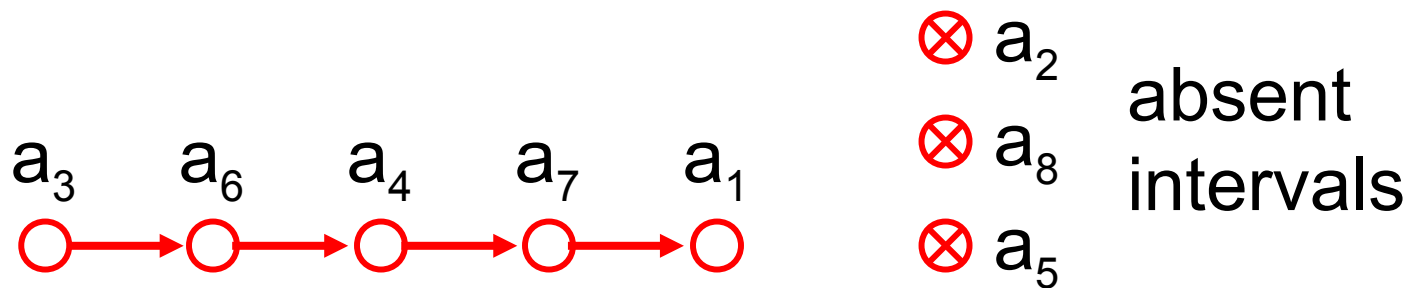
Data structure



- A **sequence variable** p is defined on a set of interval variables $A = \{a_1, \dots, a_n\}$



- A **value** of p is a total ordering of the present intervals in A



- Sequence variable declaration in OPL

```
dvar interval a[i in ...] ...;
```

```
dvar sequence p in A;
```

- What is the domain of p in this model?

```
dvar interval a[i in 1..3] optional (i%2==1) ;
```

```
dvar sequence p in a;
```

{ (a2),

(a2 \rightarrow a1), (a1 \rightarrow a2), (a2 \rightarrow a3), (a3 \rightarrow a2),

(a1 \rightarrow a2 \rightarrow a3), (a1 \rightarrow a3 \rightarrow a2), (a2 \rightarrow a1 \rightarrow a3),

(a2 \rightarrow a3 \rightarrow a1), (a3 \rightarrow a1 \rightarrow a2), (a3 \rightarrow a2 \rightarrow a1) }

- If both a and b are present and a is before b in the sequence p , then a is constrained to end before the start of b
- More formally, if p is a sequence on A ,
 $\text{noOverlap}(p)$:
$$\forall a, b \in A, 0 < p(a) < p(b) \Leftrightarrow e(a) \leq s(b)$$

- Typically, a no-overlap constraint can be used to model a set of activities requiring a unary resource.
- OPL declaration of a no-overlap constraint

```
dvar interval A[i in ...] ...;  
dvar sequence p in A;  
constraints {  
    noOverlap(p) ;  
}
```

- Transition distance:
 - An integer type $T(a)$ can be associated with any interval a in a sequence p .
 - A minimal transition distance M between interval variables in the chain can be specified in the no-overlap constraint. It is specified as a matrix indexed by the interval types.

Model 1 - OPL Model for Flow-shop with Earliness and Tardiness Costs

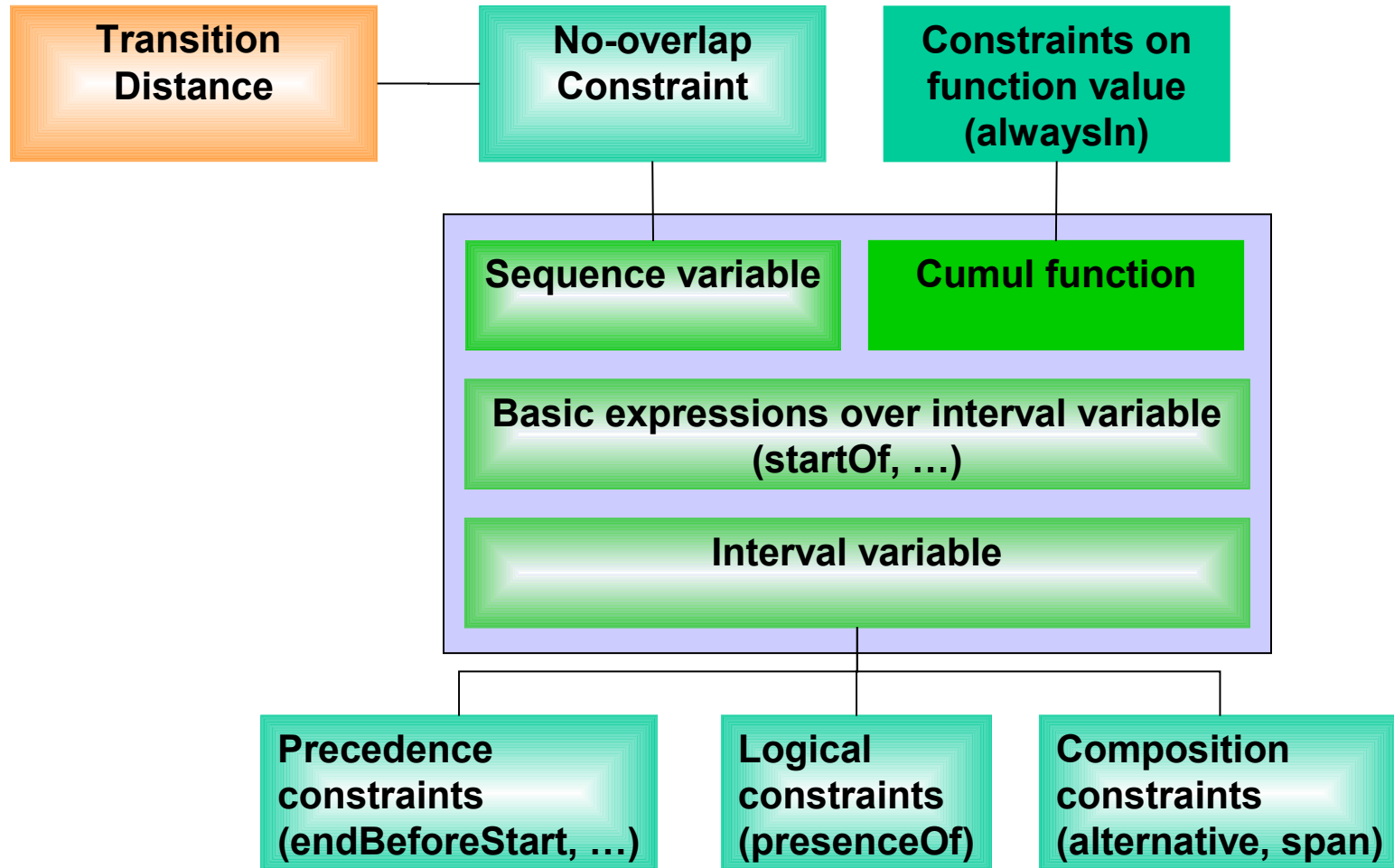
```
1: using CP;
2: int n = ...;
3: int m = ...;
4: int rd[1..n] = ...;
5: int dd[1..n] = ...;
6: float w[1..n] = ...;
7: int pt[1..n][1..m] = ...;
8: float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);
9: dvar interval op[i in 1..n][j in 1..m] size pt[i][j];
10: dexpr int C[i in 1..n] = endOf(op[i][m]);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

Language for detailed scheduling

Variable/expression

Constraint

Data structure

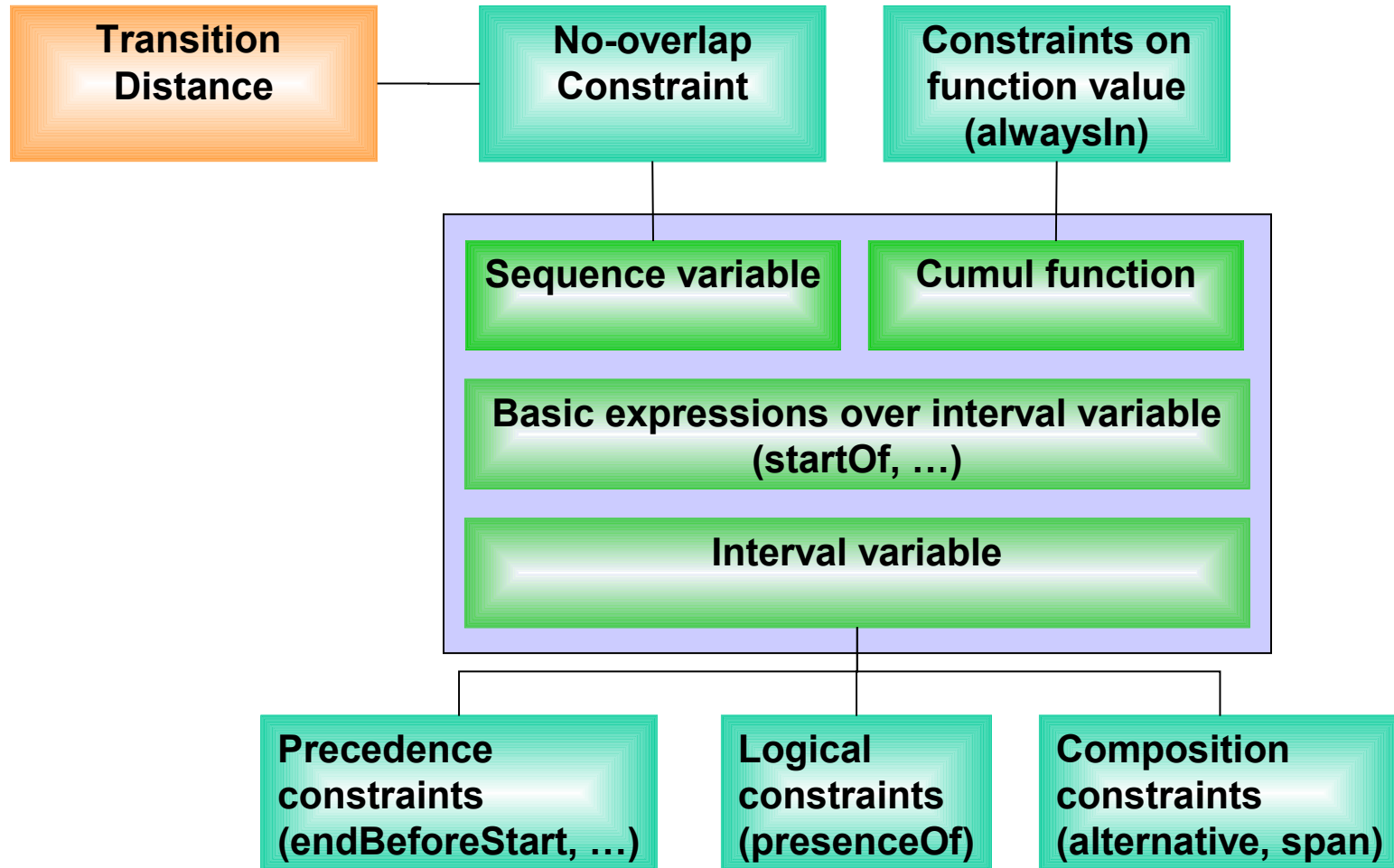


Language for detailed scheduling

Variable/expression

Constraint

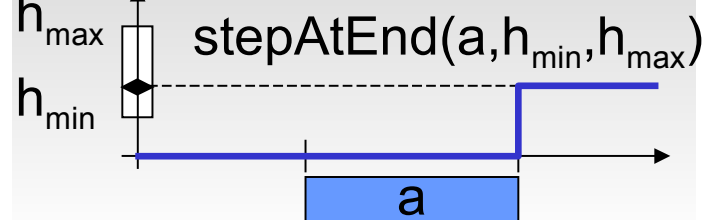
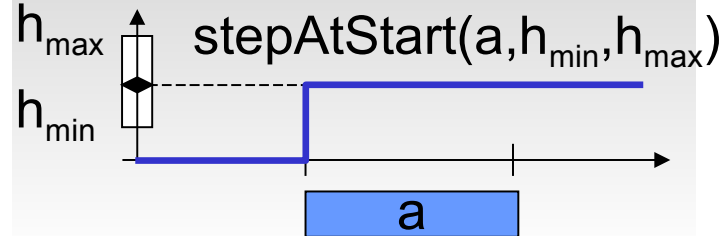
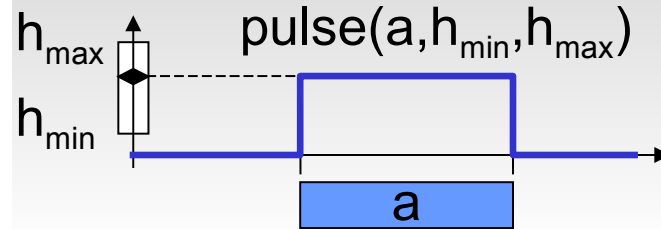
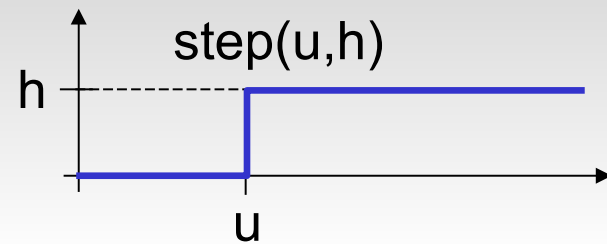
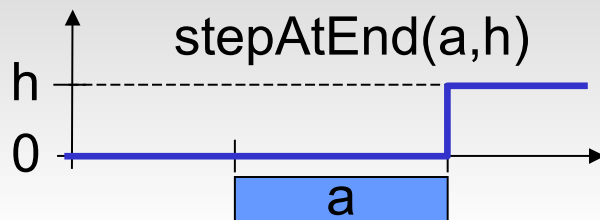
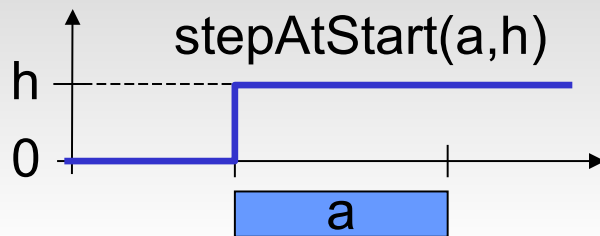
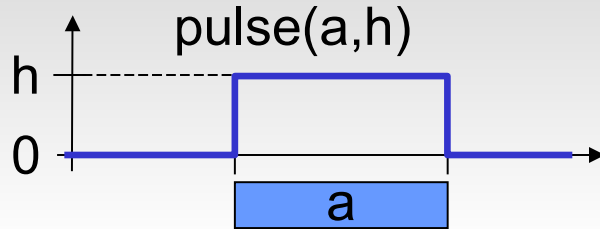
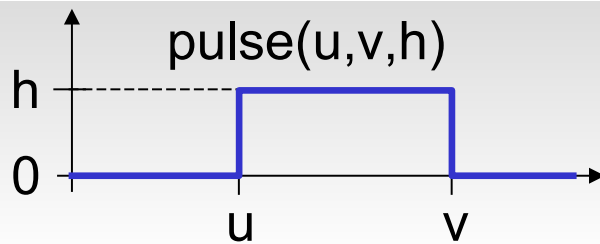
Data structure



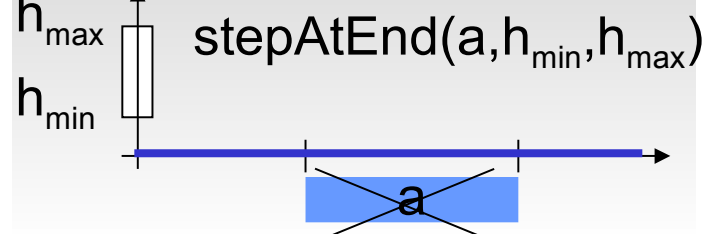
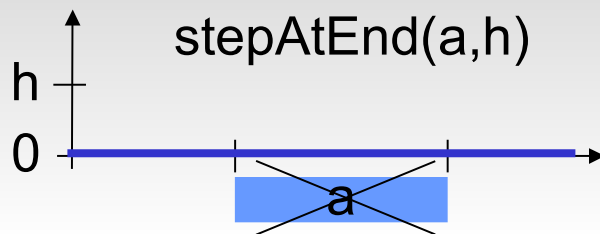
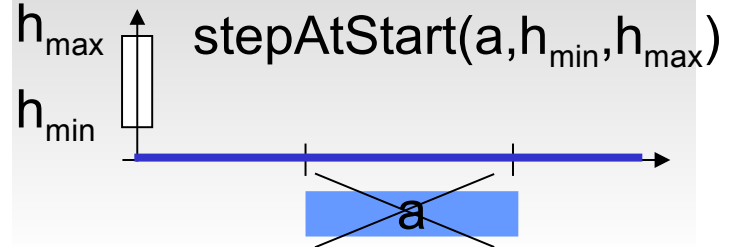
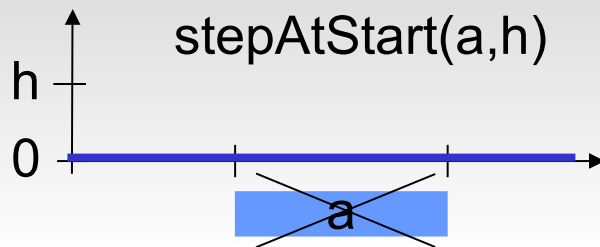
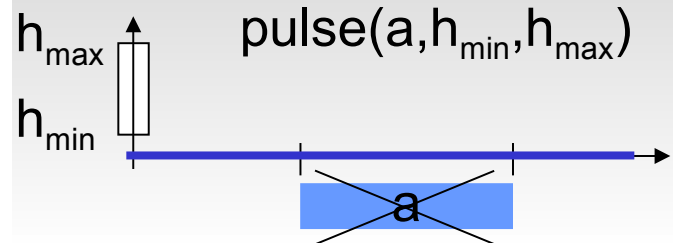
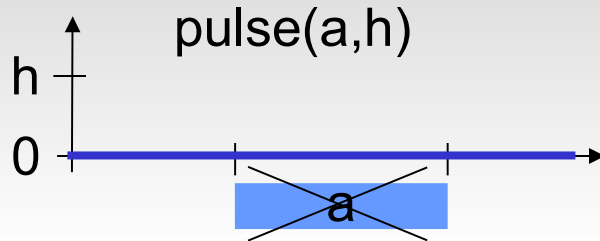
- Objective :
 - Model discrete capacity resources (ILOG Scheduler: discrete resources, discrete reservoirs)
 - and more than that ...
- The value of a **cumul function** represents the time evolution of a quantity (e.g. level of a reservoir) that can be incrementally changed (increased or decreased) by interval variables

- The individual contribution of an interval to a cumul function is called an **elementary cumul function**
- An elementary cumul function is a cumul function

Elementary cumul functions



- If a is absent, the function is the **null function** (null contribution)



- OPL declaration of an **elementary** cumul function:

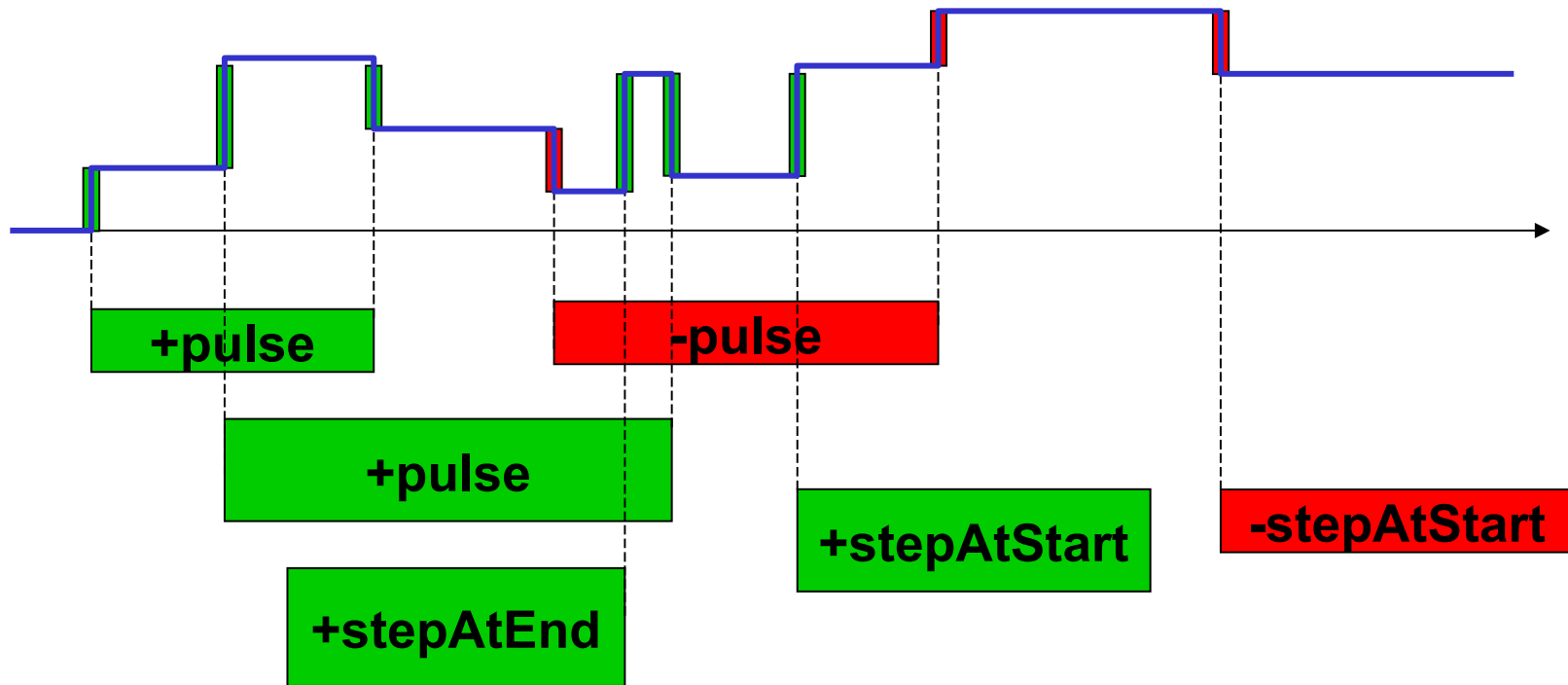
```
dvar interval a ...;  
int h, hmin, hmax, u, v;  
  
cumulFunction f = pulse(a,h) ;  
cumulFunction f = pulse(a,hmin,hmax) ;  
cumulFunction f = stepAtStart(a,h) ;  
cumulFunction f = stepAtStart(a,hmin,hmax) ;  
cumulFunction f = stepAtEnd(a,h) ;  
cumulFunction f = stepAtEnd(a,hmin,hmax) ;  
cumulFunction f = pulse(u,v,h) ;  
cumulFunction f = step(u,h) ;
```

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

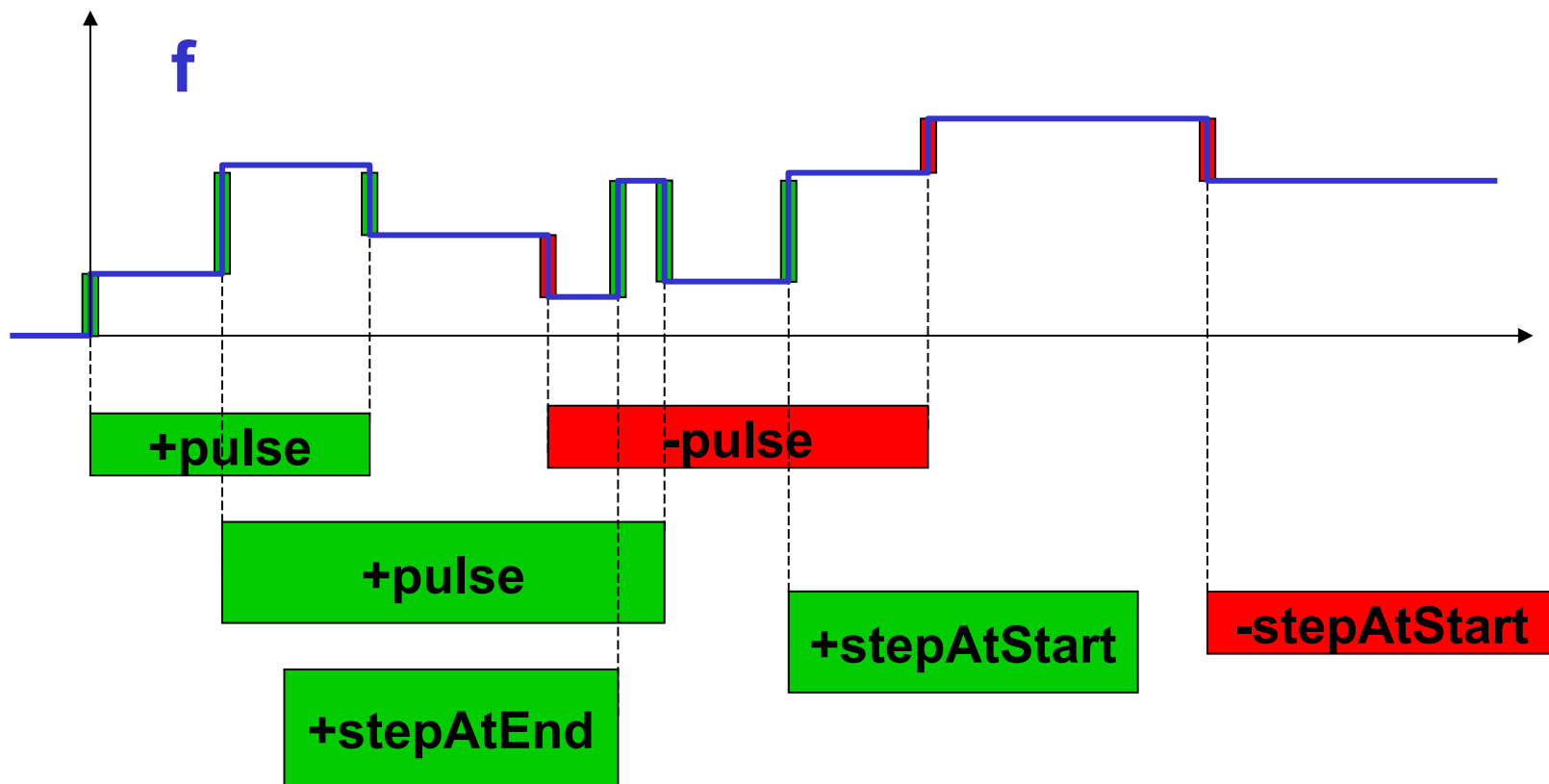
$$\forall t, f(t) = \sum_i \varepsilon_i \cdot f_i(t)$$

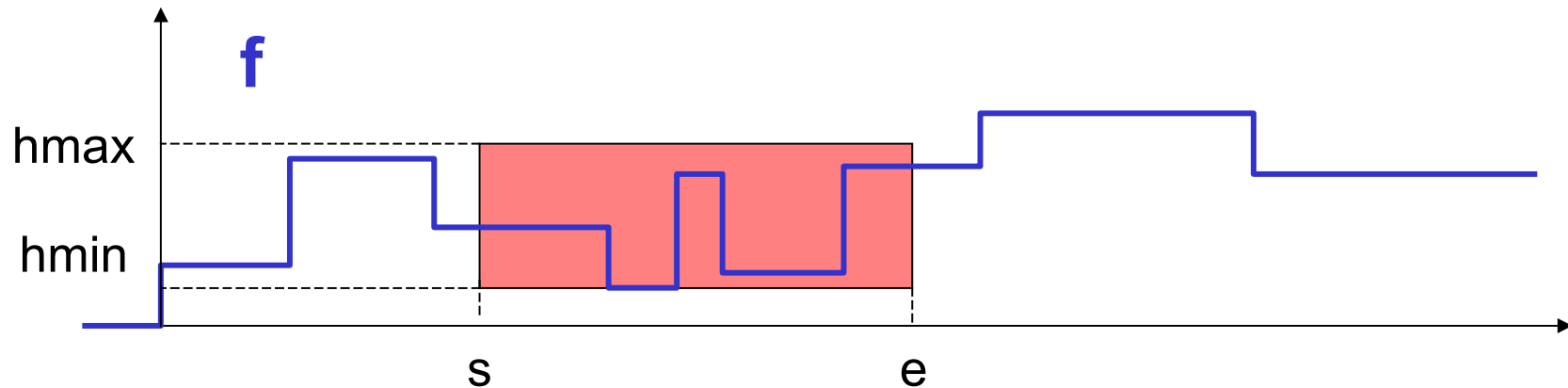
$$\varepsilon_i \in \{-1, +1\}$$

Cumul function



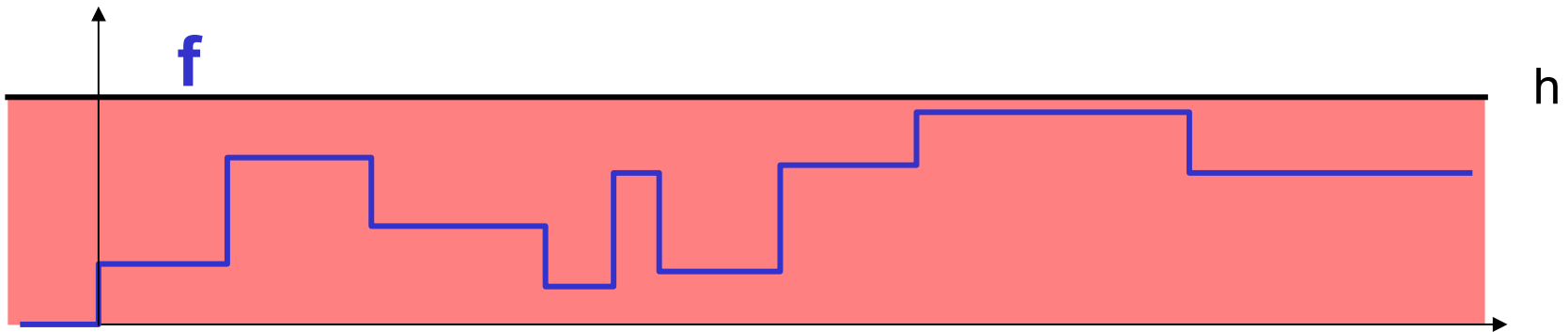
Constraints on cumul functions





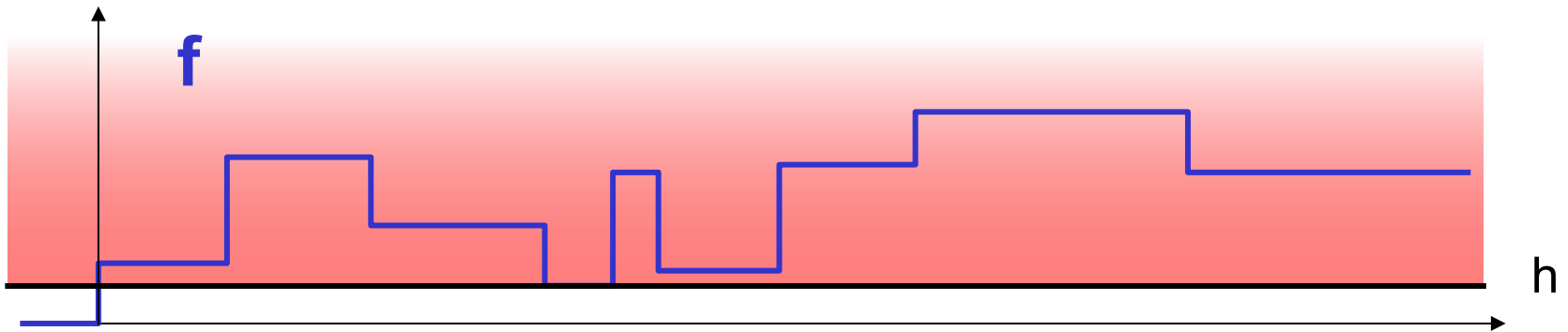
- Constraint: over a fixed interval $[s, e)$, f always takes its value in a fixed range $[hmin, hmax]$

`alwaysIn(f, s, e, hmin, hmax)`



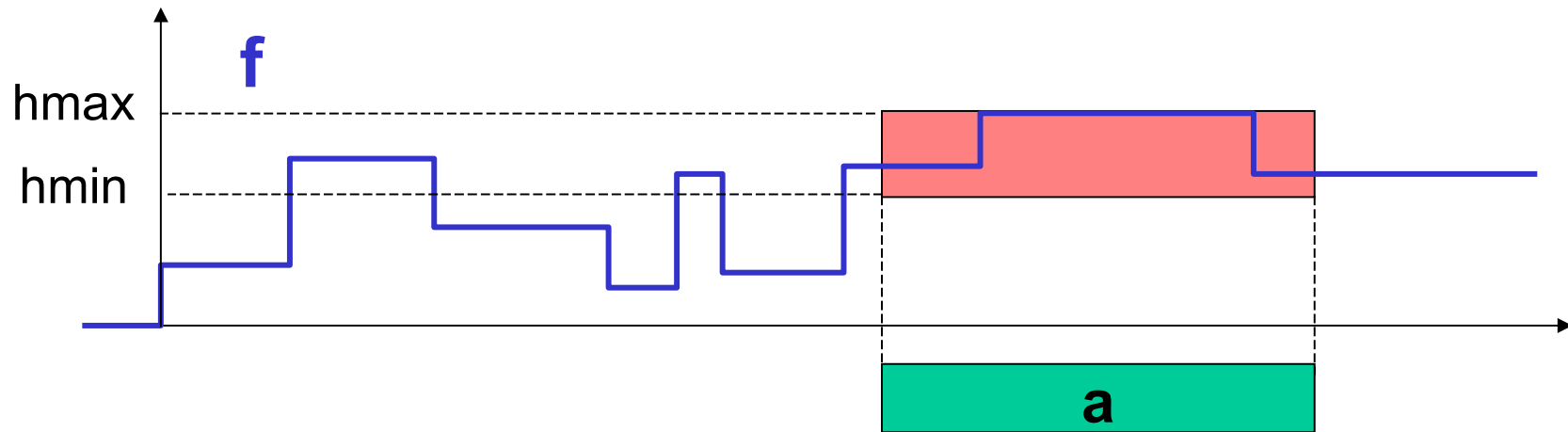
- Constraint: some shortcut for constraints over the complete horizon $[-\infty, +\infty]$

$f \leq h;$



- Constraint: some shortcut for constraints over the complete horizon $[-\infty, +\infty]$

$h \leq f$;



- Constraint: over an interval variable a (if present), f always takes its value in a fixed range $[hmin, hmax]$

`alwaysIn(f, a, hmin, hmax)`

- Example of a discrete (renewable) resource of capacity Q required by n activities (activity i requires $q[i]$ units).

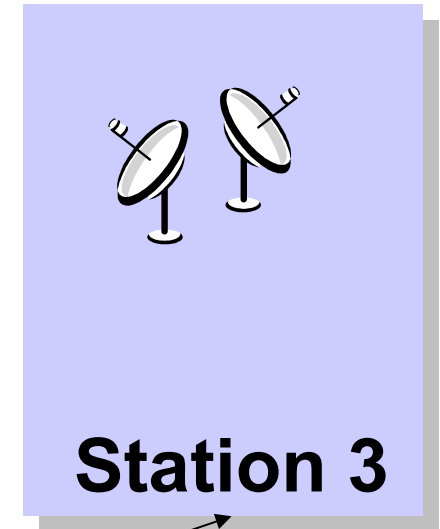
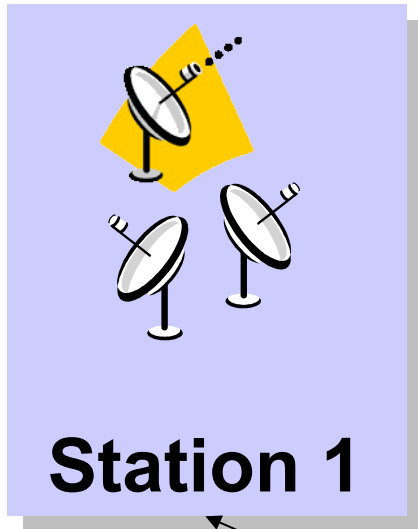
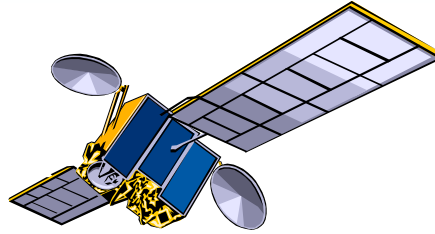
```
dvar interval a[i in 1..n] ...;  
int q[i in 1..n] = ...;  
cumulFunction f = sum(i in 1..n) pulse(a[i],q[i]);  
subject to {  
    f <= Q;  
}
```

- USAF Satellite Control Network scheduling problem [Kramer&al 2007]
- A set of n input communication requests for Earth orbiting satellites must be scheduled on a total of 32 antennas spread across 13 ground-based tracking stations.
- Objective is to maximize the number of satisfied requests

[Kramer&al 2007] L. Kramer, L. Barbulescu and S. Smith. *"Understanding Performance Tradeoffs in Algorithms for Solving Oversubscribed Scheduling"*. Proc. AAAI-07, July, 2007.

- A station S_j is associated a number of antennas C_j
- A request R_i is associated a set of alternative allocations. An allocation specifies:
 - A ground station
 - A time window $[smin_i, emax_i]$
 - A task duration
- When executed on a station, the request will require 1 antenna of the station
- All requests are optional: the objective is to maximize the number of satisfied requests

Oversubscribed satellite scheduling



Duration1

Duration3



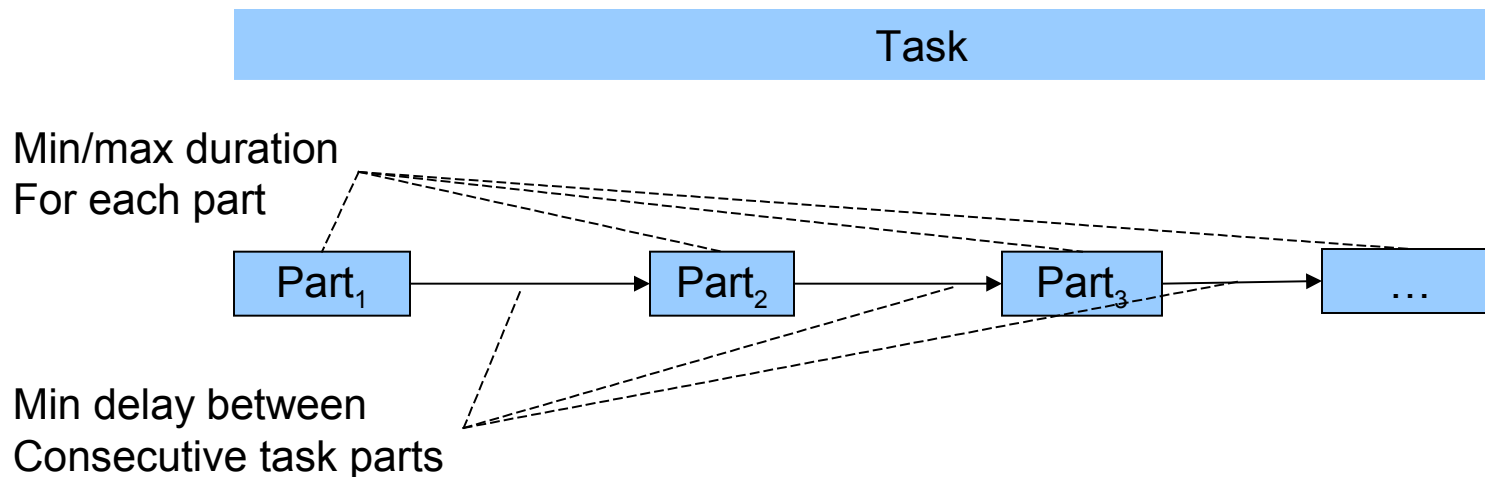
Model 2 - OPL Model for Satellite Scheduling

```
1: using CP;
2: tuple Station { string name; key int id; int cap; }
3: tuple Alternative { string task; int station; int smin; int dur; int emax; }
4: {Station} Stations = ...;
5: {Alternative} Alternatives = ...;
6: {string} Tasks = { a.task | a in Alternatives };
7: dvar interval task[t in Tasks] optional;
8: dvar interval alt[a in Alternatives] optional in a.smin..a.emax size a.dur;
9: maximize sum(t in Tasks) presenceOf(task[t]);
10: subject to {
11:   forall(t in Tasks)
12:     alternative(task[t], all(a in Alternatives: a.task==t) alt[a]);
13:   forall(s in Stations)
14:     sum(a in Alternatives: a.station==s.id) pulse(alt[a],1) <= s.cap;
15: }
```

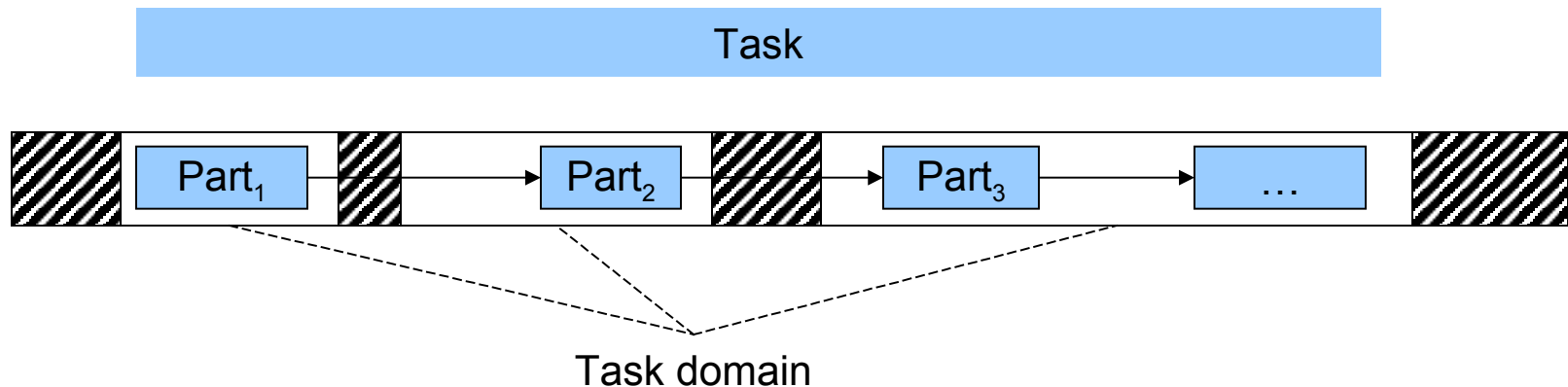
- Personal tasks scheduling [Refanidis 2007]
- Schedule a personal agenda composed of n tasks
- Available online: <http://selfplanner.uom.gr/>

[Refanidis 2007] I. Refanidis. *"Managing Personal Tasks with Time Constraints and Preferences"*. Proc. ICAPS-07, September, 2007.

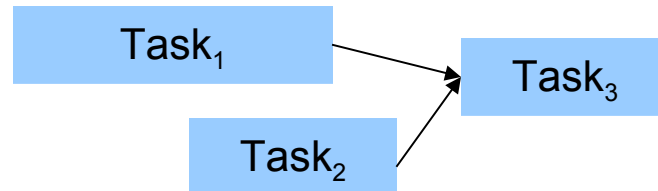
- Tasks are preemptive
 - A task specifies a fixed total processing time
 - It can be split into one or several parts.
 - There is a min/max value for the duration of each individual part
 - There is a minimal delay between consecutive parts of the same task



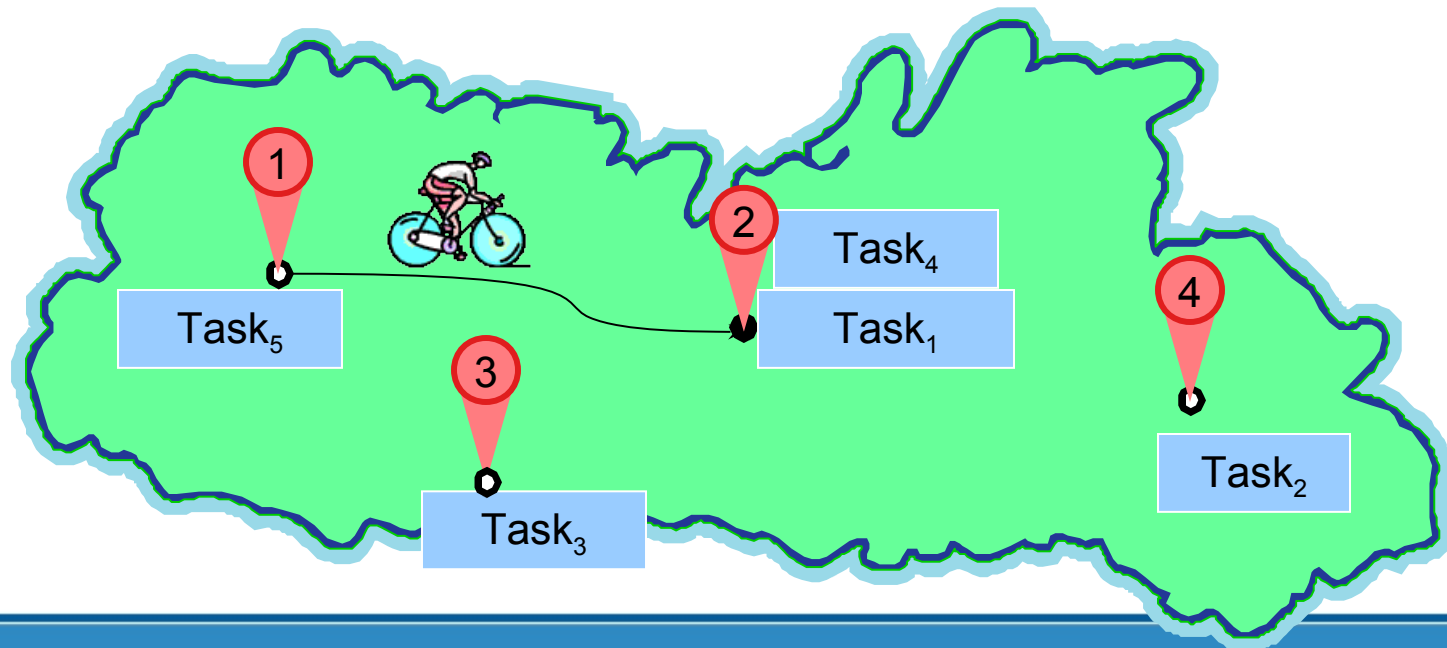
- Each task specifies a set of time-windows where it can be executed



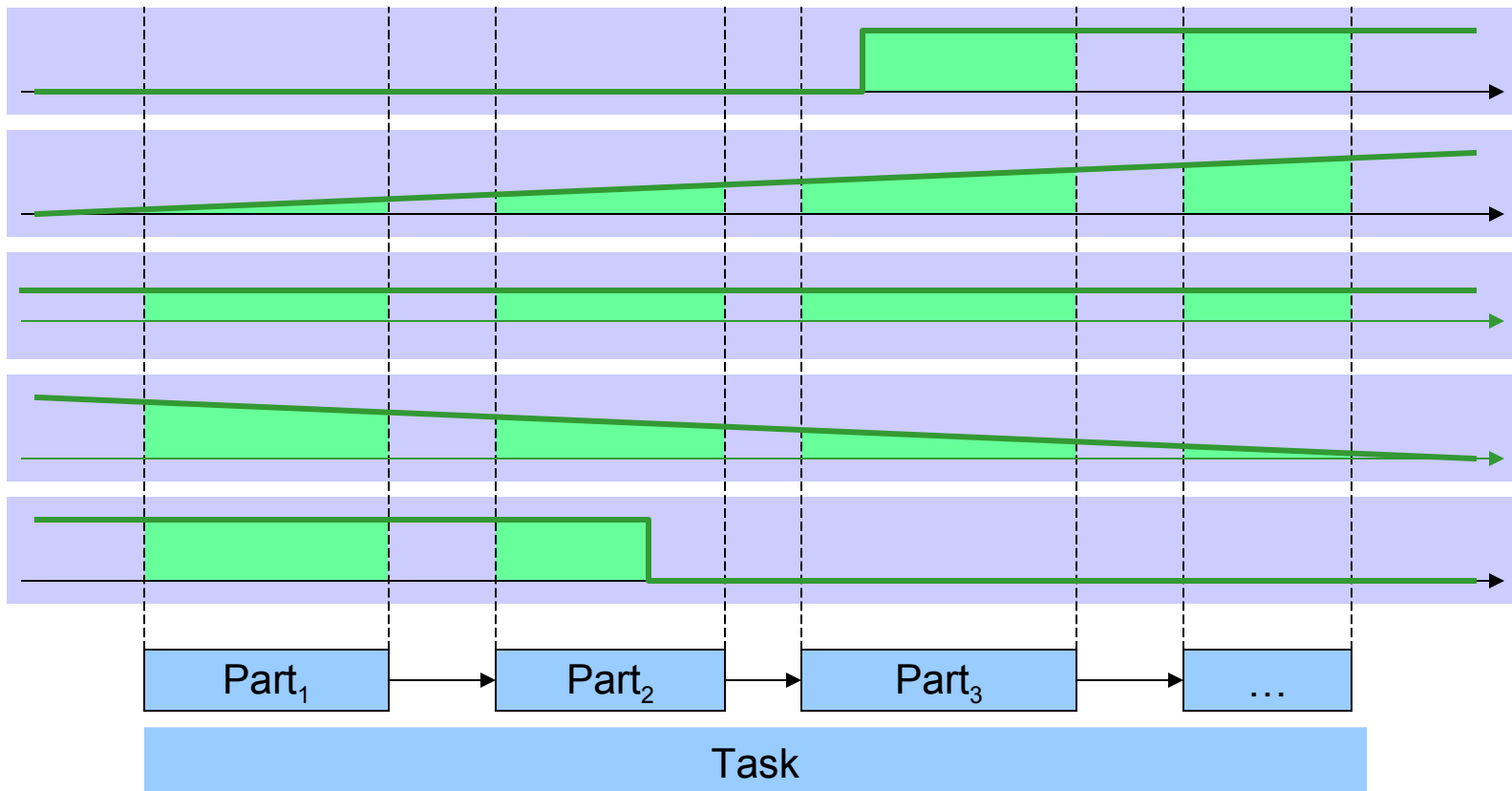
- Precedence constraints



- Locations, distances and transition times



- Objective function: maximize task satisfaction
- 5 types of task-dependent preference functions:



Model 3 - OPL Model for Personal Task Scheduling

```
1: using CP;
2: tuple Task { key int id; int loc; int dur; int smin; int smax; int dmin; int f; int
  date; {int} ds; {int} de; }
3: {Task} Tasks = ...;
4: tuple Distance { int loc1; int loc2; int dist; };
5: {Distance} Dist = ...;
6: tuple Ordering { int pred; int succ; };
7: {Ordering} Orderings = ...;
8: int L[t in Tasks] = min(x in t.ds) x;
9: int R[t in Tasks] = max(x in t.de) x;
10: int S[t in Tasks] = R[t]-L[t];
11: tuple Part { Task task; int id; }
12: {Part} Parts = { <t,i> | t in Tasks, i in 1 .. t.dur div t.smin };
13: tuple Step { int x; int y; }
14: sorted {Step} Steps[t in Tasks] =
15:   {<x,0> | x in t.ds} union {<x,1> | x in t.de};
16: stepFunction holes[t in Tasks] = stepwise(s in Steps[t]) {s.y -> s.x; 0};
17: dvar interval tasks[t in Tasks] in 0..500;
18: dvar interval a[p in Parts] optional size p.task.smin..p.task.smax;
19: dvar sequence seq in all(p in Parts) a[p] types all(p in Parts) p.task.loc;
20: dexpr float satisfaction[t in Tasks] = (t.f==0)? 1 :
21:   (1/t.dur)* sum(p in Parts: p.task==t)
22:   (t.f==2)? maxl(endOf(a[p]),t.date)-maxl(startOf(a[p]),t.date) :
23:   (t.f==1)? lengthOf(a[p])*(R[t]-(startOf(a[p])+endOf(a[p])-1)/2)/S[t] :
24:   (t.f== 1)? lengthOf(a[p])*((startOf(a[p])+endOf(a[p])-1)/2-L[t])/S[t] :
25:   (t.f== 2)? minl(endOf(a[p]),t.date)-minl(startOf(a[p]),t.date) : 0;
26: maximize sum(t in Tasks) satisfaction[t];
27: subject to {
28:   forall(p in Parts) {
29:     forbidExtent(a[p], holes[p.task]);
30:     forall(s in Parts: s.task==p.task && s.id==p.id+1) {
31:       endBeforeStart(a[p], a[s], p.task.dmin);
32:       presenceOf(a[s]) => presenceOf(a[p]);
33:     }
34:   }
35:   forall(t in Tasks) {
36:     t.dur == sum(p in Parts: p.task==t) sizeOf(a[p]);
37:     span(tasks[t], all(p in Parts: p.task==t) a[p]);
38:   }
39:   forall(o in Orderings)
40:     endBeforeStart(tasks[<o.pred>], tasks[<o.succ>]);
41:   noOverlap(seq, Dist);
42: }
```

■ Experimental Results

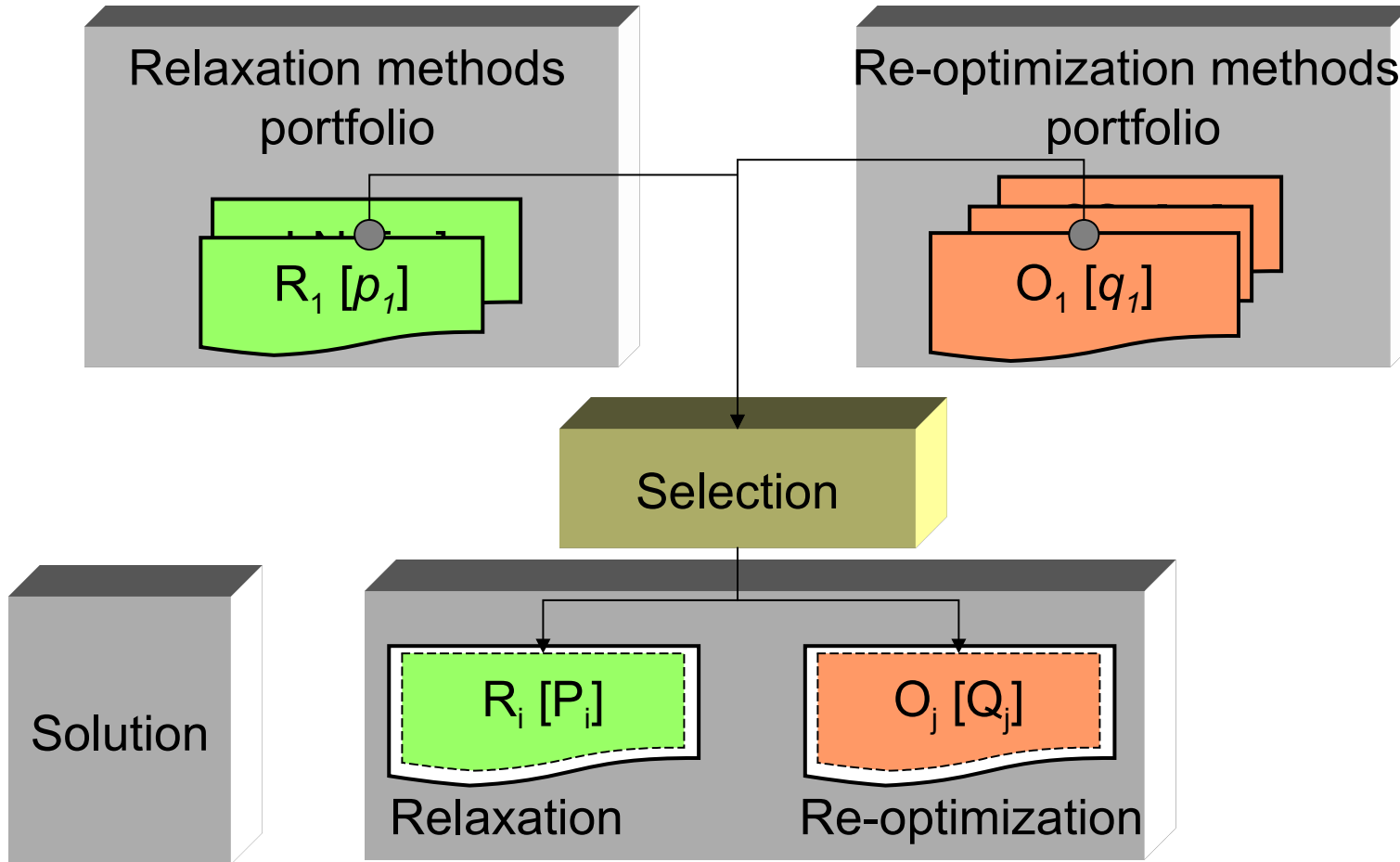
- Flowshop with earliness/tardiness costs
 - Similar results (slight improvement of 2.7% in average) as state-of-the-art problem specific algorithms (GAs, LNS)
- Oversubscribed satellite communication scheduling
 - CPO assigns 5.3% more tasks in average than state-of-the-art problem specific algorithms (Tabu Search, Squeaky Wheel Optimization)
- Personal tasks scheduling with preferences
 - CPO solves more problems than state-of-the-art problem specific algorithms (Squeaky Wheel Optimization) ...
 - ... with better quality (average improvement of 12.5% of task satisfaction)

- Results over 22 benchmarks

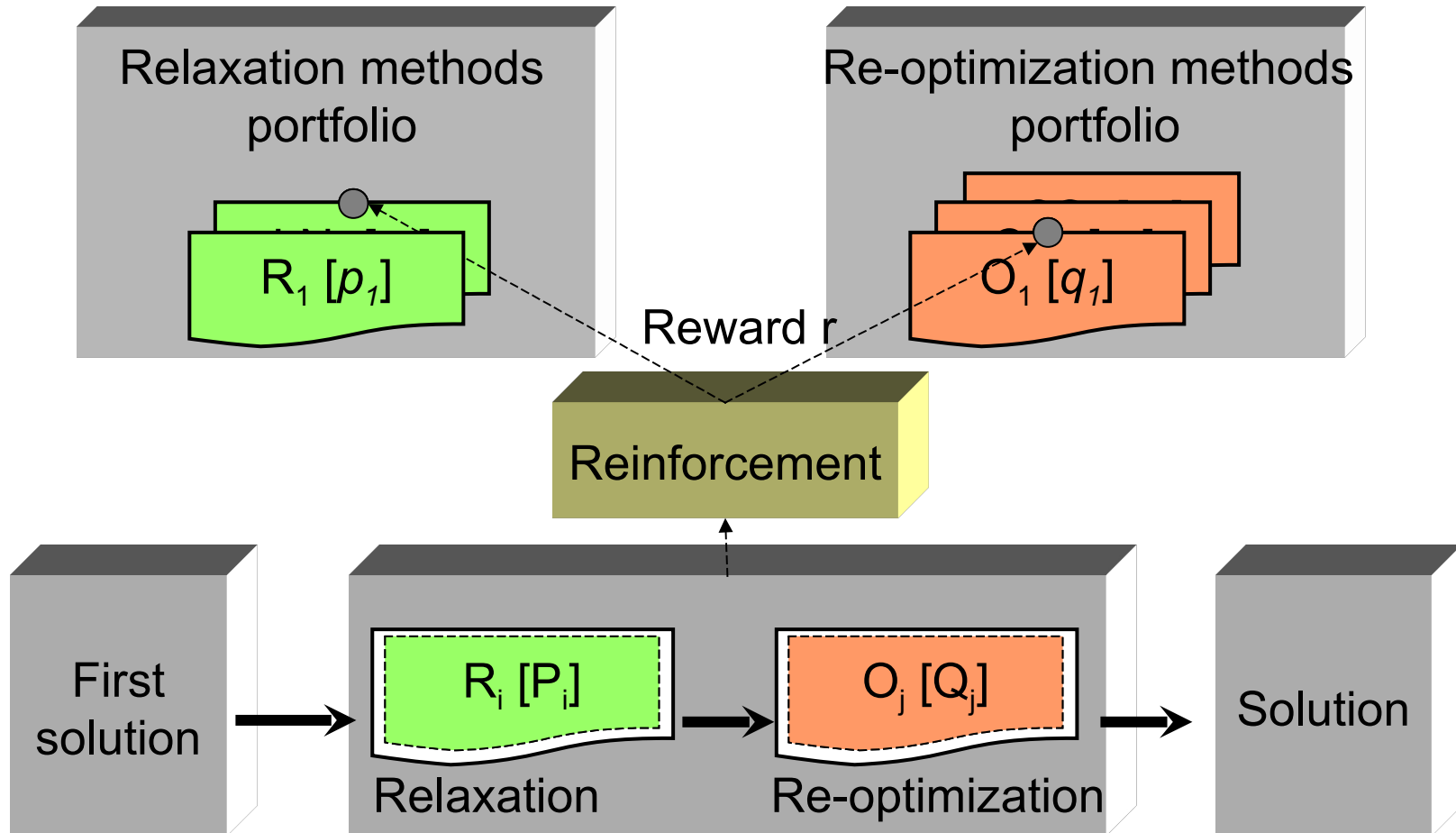
Bench index	Problem type	MRD	# Imp. UBs / # Instances
1	Trolley	-10.2%	15/15
2	Hybrid flow-shop	-11.3%	19/20
3	Job-shop w/ E/T	-6.2%	41/48
4	Air traffic management	-7.0%	1/1
5	Max. quality RCPSP	-2.7%	NA/3600
6	Flow-shop w/ E/T	-1.1%	5/12
7	RCPSP w/ E/T	-2.1%	16/60
8	Cumulative job-shop	-0.1%	15/86
9	Semiconductor testing	-0.3%	7/18
10	Single proc. tardiness	0.3%	0/20
11	Open-shop	0.3%	0/28
12	MaScLib single machine	0.6%	0/60
13	Shop w/ setup times	0.4%	3/15
14	RCPSP	1.2%	2/600
15	Air land	0.0%	0/8
16	Parallel machine w/ E/T	1.6%	4/52
17	Job-shop	1.9%	0/33
18	Flow-shop	0.9%	4/22
19	Flow-shop w/ buffers	3.9%	11/30
20	Single machine w/ E/T	7.4%	0/40
21	Aircraft assembly	8.7%	0/1
22	Common due-date	6.8%	4/20

- Some elements about how it works inside
 - Default search strategy (Restart)
 - Propagation on conditional bounds

Restart for detailed scheduling



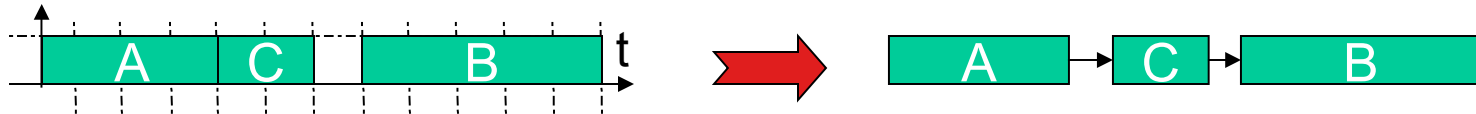
Restart for detailed scheduling



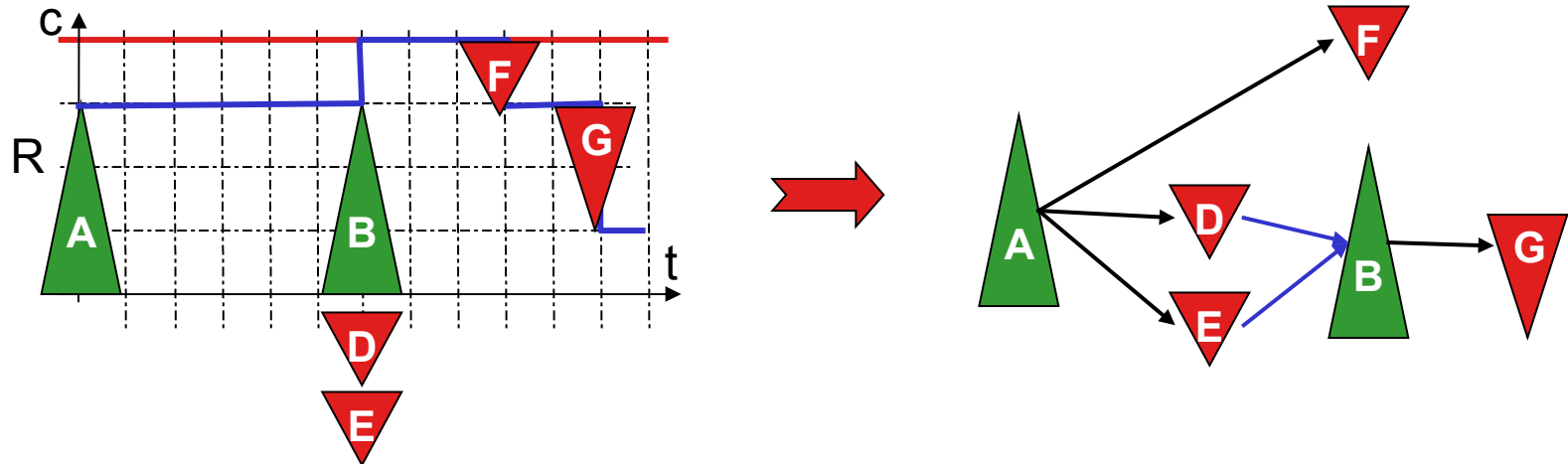
- Partial Order Schedule (POS) definition
 - A temporal network that is sufficient to ensure that all its solutions satisfy the temporal and “resource” constraints of the problem
- All relaxation methods in the portfolio start by computing a POS from the solution and then, relax a subset of activities F (fragment) on this temporal network

$$R_i = \text{Relax}_{F_i} \circ \text{POS}$$

- POS computation on “resources”
 - Sequence/NoOverlap: easy, $O(n \log n)$



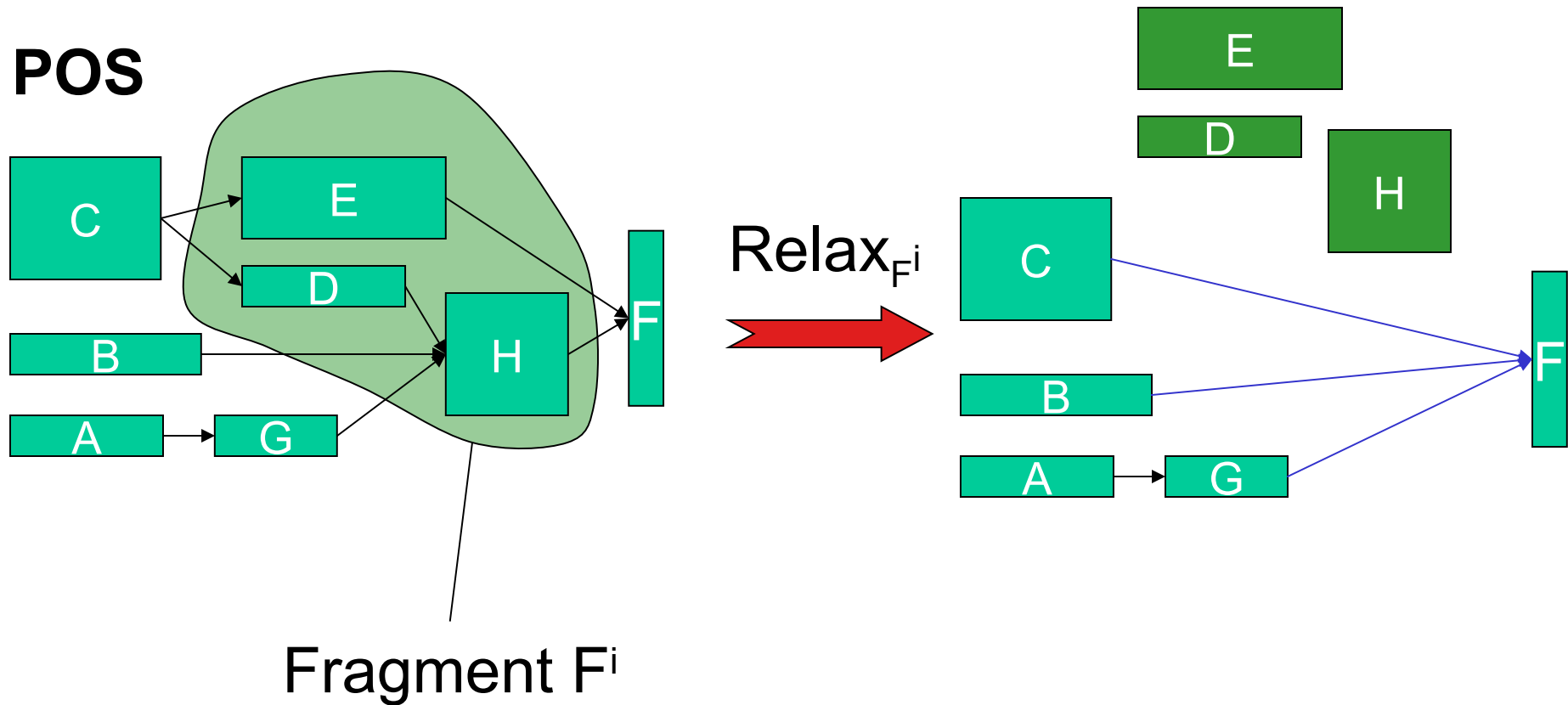
- Cumul: $O(n \log(Cn))$



- Partial Order Schedule (POS) definition
 - A temporal network that is sufficient to ensure that all its solutions satisfy the temporal and resource constraints of the problem
- All relaxation methods in the portfolio start by computing a POS from the solution and then, relax a subset of interval variables F (fragment) on this temporal network

$$R_i = \text{Relax}_{F_i} \circ \text{POS}$$

- Relaxation of a fragment F_i of the POS



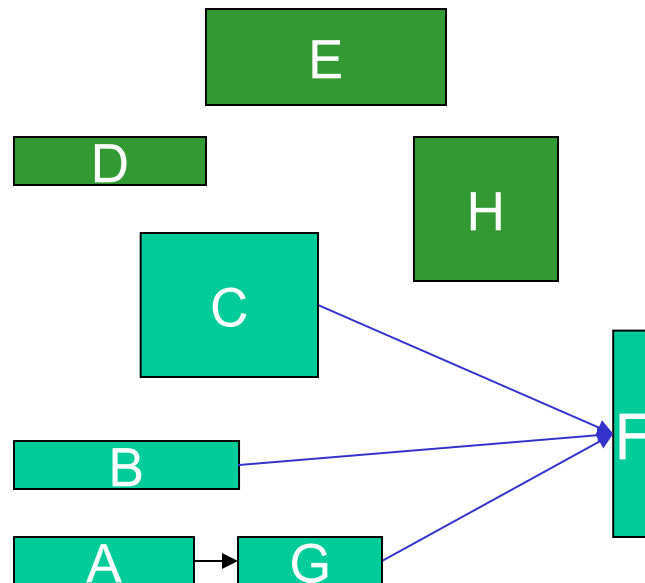
- Portfolio:

- R^1_{α} : Randomized relaxation
- $R^2_{\alpha,\beta}$: Time-window relaxation
- $R^3_{\alpha,\beta}$: Topological relaxation
- $R^4_{\alpha,\beta}$: Slack-based relaxation

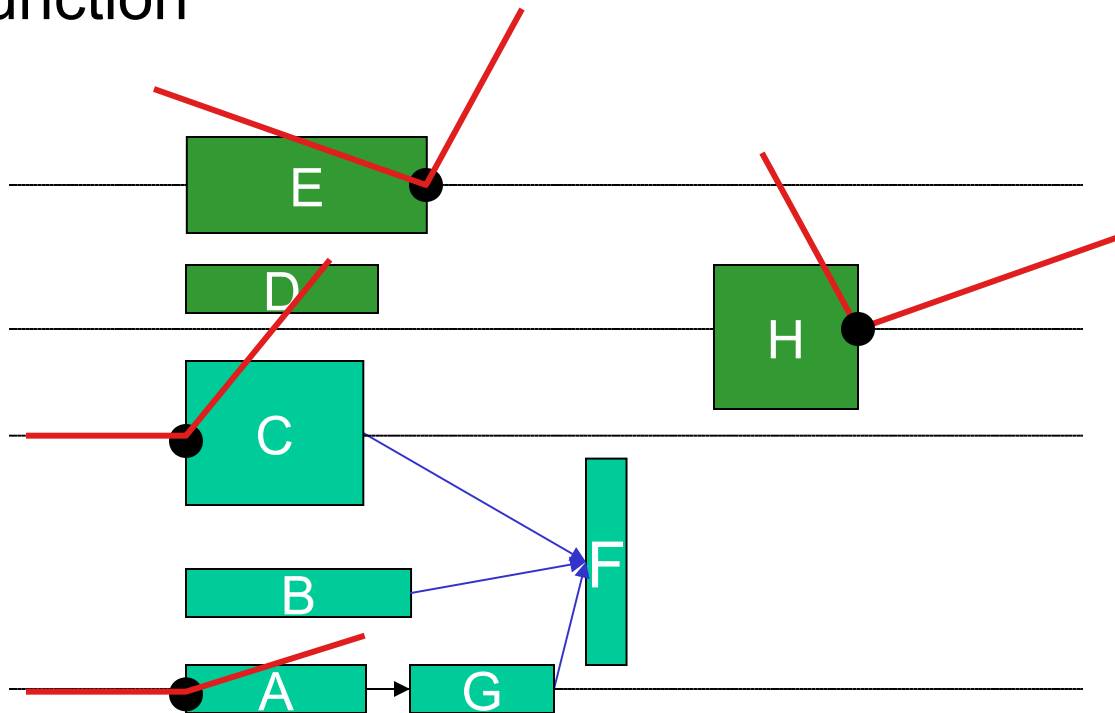
- Currently, a unique optimization method is used: $\text{ScheduleJustInTime}_\alpha$
 - Explores a search tree with a limited number of failures $\alpha.n$
 - α is a self-adapting parameter of the optimization method
 - At the root node, indicative start and end values for interval variables are computed using an LP relaxation of the problem

- Temporal relaxation:
 - Only consider temporal constraints (including the ones of the relaxed POS) and (convexified) cost function

- Temporal relaxation:
 - Only consider temporal constraints (including the ones of the relaxed POS) and (convexified) cost function



- Temporal relaxation:
 - Only consider temporal constraints (including the ones of the relaxed POS) and (convexified) cost function



- Tree search
 - Search considers interval variables by increasing indicative start values and tries to schedule them as close as possible to their indicative values

- Multi-Points:

- Based on Genetic Programming

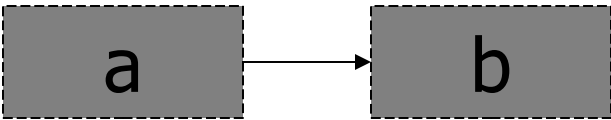
- Depth-First:

- Not really efficient on optimization problems
 - Mostly useful for:
 - Debugging or investigating a model
 - Producing all solutions to a decision problem
 - Proving that there is no solution to a problem

- Interval variable domain representation: tuple of ranges:
 - $[x_{\min}, x_{\max}] \subseteq [0, 1]$: current execution status
 - $[s_{\min}, s_{\max}] \subseteq \mathbb{Z}$: **conditional** domain of start **would the interval be present**
 - $[e_{\min}, e_{\max}] \subseteq \mathbb{Z}$: **conditional** domain of end **would the interval be present**
 - $[l_{\min}, l_{\max}] \subseteq \mathbb{Z}^+$: **conditional** domain of length **would the interval be present**

- Logical constraints are aggregated in an implication graph: all 2-SAT logical constraints $[\neg]x(a) \vee [\neg]x(b)$ are translated as implications $(\neg[\neg]x(a) \Rightarrow [\neg]x(b))$
- **Incremental transitive closure** of the implication graph allows detecting infeasibilities and querying in $O(1)$ whether $x(a) \Rightarrow x(b)$ for any (a,b)

- Precedence constraints are aggregated in a temporal network

- **Conditional reasoning:** *From logical network*

 $x(a) \Rightarrow x(b)$
`endBeforeStart(a,b)`

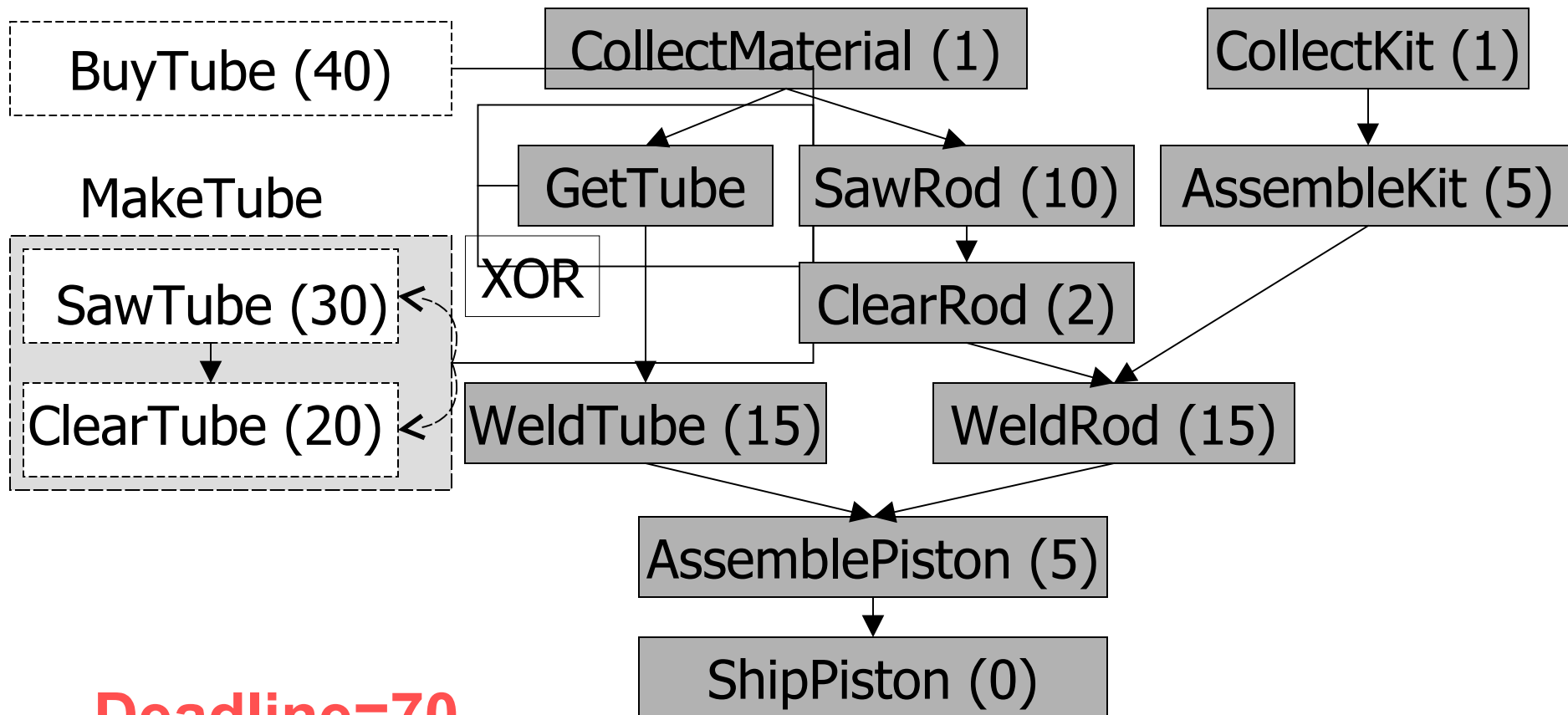
- Propagation on the conditional bounds of a (would a be present) can assume that b will be present too, thus:

$$e_{\max}(a) \leftarrow \min(e_{\max}(a), s_{\max}(b))$$

- **Bounds are propagated even on interval variables with still undecided presence status**

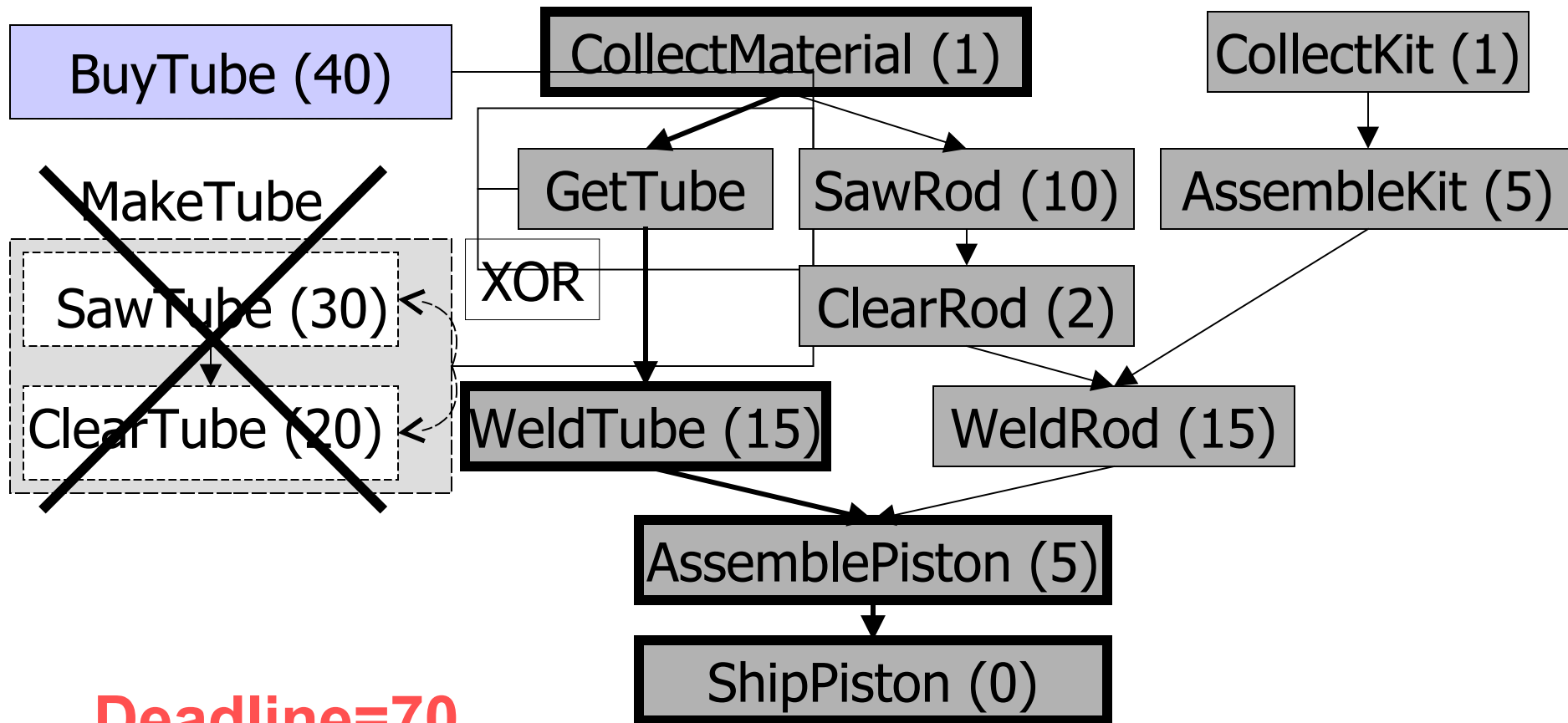
Constraint Propagation: Simple example

- Inspired from [Barták&Čepek 2007]



Constraint Propagation: Simple example

- Inspired from [Barták&Čepek 2007]



Constraint propagation: Inference Levels

Model element	Inference level	Filtering algorithms
Sequence variable	Basic \geq Medium	Light precedence graph Precedence graph
No-overlap constraint	Basic Medium Extended	Timetable + Disjunctive + EF variants
Cumul function expression	Basic Medium Extended	Timetable + Disjunctive + EF variants
State function variable	Basic \geq Medium	Timetable + Disjunctive

- Limited number of concepts
- Naturally fit into a CP paradigm with clearly identified decision variables/expressions and constraints
- Expressive model
 - Optional activities / Oversubscribed problems
 - Alternative processes/modes/routes
 - Complex synchronization between activities
 - Complex cost functions (regular/non-regular, resource costs, etc.)

- Ingredients to the robustness of the approach:
 - LNS: efficient traversal of the search space
 - POS: generality, injects flexibility for re-optimization
 - Relaxation methods:
 - Randomization allows diversity
 - Some methods exploit problem structure
 - Re-optimization methods:
 - Global vision provided by temporal relaxation
 - (Limited) Tree search allows exploiting CP and powerful propagation on conditional bounds
 - Learning

- <http://www.ilog.com/products/cpoptimizer>
 - White papers
 - Presentations
 - Data sheet
- <http://www2.ilog.com/techreports> has some technical reports adapted from papers
 - TR-07-001: Large neighborhood search (MISTA-07)
 - TR-08-001: Reasoning with conditional time intervals (FLAIRS-08)
 - TR-08-002: Scheduling model exhaustive & formal description
 - TR-09-001: Reasoning with conditional time intervals (II) (FLAIRS-09)
 - TR-09-002: CP Optimizer illustrated on 3 problems (CPAIOR-09)