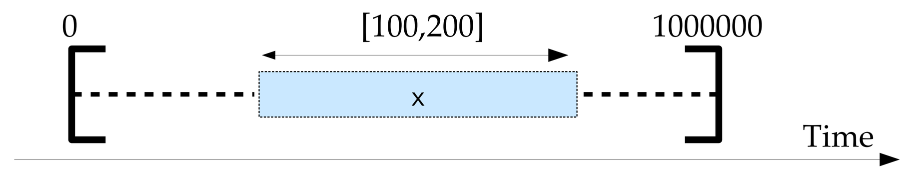
IBM ILOG CP OPTIMIZER FOR SCHEDULING

P. Laborie - IBM - laborie@fr.ibm.com

MODEL

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, *etc.*)
- If desired, intervals can be optional: that is, whether the interval will be present or absent in a solution is part of the optimization problem [1,2]
- Example: `dvar interval x optional in 0..1000000 size 100..200;`



- Logical constraints on interval presence: `presenceOf(x) => presenceOf(x)`
- Precedence constraints: `endBeforeStart(x,y, delay)`
- Step functions for modeling resource time-dependent intensity and breaks
- Integer expressions to get interval attributes: `startOf(x,ValIfAbsent)`

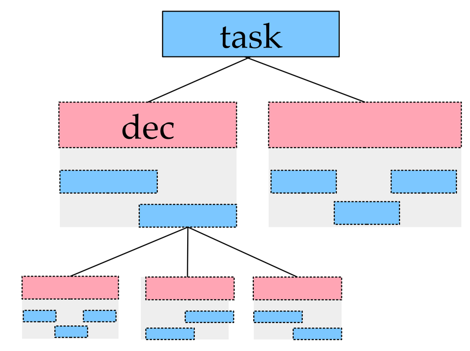
Structural constraints

- **What for?** Structure the problem in the form of an AND/OR graph
- OR nodes = alternative constraint `alternative(x, [y1,…,yn])`
- AND nodes = span constraint `span(x, [y1,…,yn])`
- Example of a work-breakdown structure:

```
using CP;
tuple Dec { int task; {int} subtasks; };
int n = ...;
int compulsory[1..n] = ...;
{Dec} Decs = ...;
int nbDecs[i in 1..n] = card( {d | d in Decs : d.task==i} );
int nbParents[i in 1..n] = card( {d | d in Decs : i in d.subtasks} );

dvar interval task[i in 1..n] optional;
dvar interval dec[d in Decs] optional;

constraints {
  forall(i in 1..n) {
    if (nbParents[i]==0 && 0<compulsory[i])
      presenceOf(task[i]);
    if (nbDecs[i]>0) {
      alternative(task[i], all(d in Decs: d.task==i) dec[d]);
      forall(d in Decs: d.task==i)
        span(dec[d], all(j in d.subtasks) task[j]);
    }
  }
  forall(d in Decs, j in d.subtasks: 0<compulsory[j])
    presenceOf(dec[d]) => presenceOf(task[j]);
}
```



Sequencing

- **What for?** Modeling constraints that enforce a total temporal ordering of a set of interval variables
- A sequence variable represents a permutation of interval variables: `dvar sequence`
- Example: Classical Job-Shop Scheduling Problem

```
dvar interval op[j in Jobs][p in Pos] size Ops[j][p].pt;
dvar sequence mchs[m in Mchs] in
  all(j in Jobs, p in Pos: Ops[j][p].mch == m) op[j][p];

minimize max(j in Jobs) endOf(op[j][nbPos]);
subject to {
  forall(m in Mchs)
    noOverlap(mchs[m]);
  forall(j in Jobs, p in 2..nbPos)
    endBeforeStart(op[j][p-1],op[j][p]);
}
```

- Additional features available for modeling sequence-dependent setup times and constraints on transitions (for instance for VRP-like problems)

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: number of workers of a given type, level of an inventory, *etc.*
- Example: Classical RCPSP

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

- Levels can be fixed or variable
- Constraints are available for limiting the value of a cumul function over some fixed time periods or variable intervals

State functions

- **What for?** The value of a state function represents the time evolution of a value that can be changed/required by interval variables
  - Two interval requiring different states cannot overlap
  - Two interval requiring the same state can (optionally) be batched together (same start and end value)
- Example of a photo-lithography machine:

```
using CP;
int n = ...;
int capacity = ...;
int pt[1..n] = ...;
int nbwafers[1..n] = ...;
int family[1..n] = ...;
tuple triplet { int id1; int id2; int value; };
{triplet} M = ...; // Transition time between pairs of families

dvar interval op[i in 1..n] size pt[i];
stateFunction batch with M;
cumulFunction load = sum (i in 1..n) pulse(op[i], nbwafers[i]);

constraints {
  load <= capacity;
  forall(i in 1..n) {
    alwaysEqual(batch, op[i], family[i], true, true);
  }
}
```

RUN

Automatic search

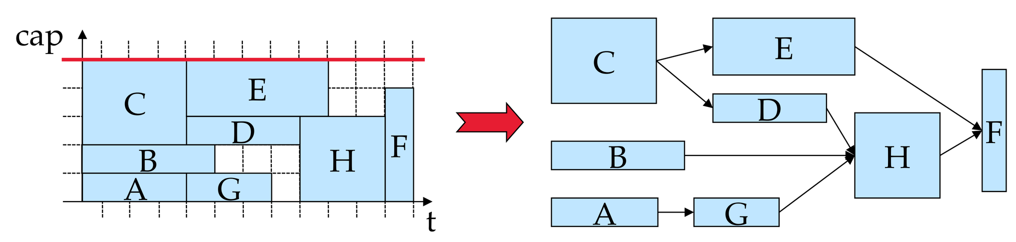
- Properties of the automated search
  - Complete
  - Anytime (usually a first feasible solution is found quickly)
  - Parallel (unless stated otherwise)
  - Randomized (internally some ties are broken using random numbers)
  - Deterministic (solving the same problem twice produces the same result)
- The search can be parametrized
  - Search parameters (time limit, number of workers, control of inference levels, random generator seed, ...)
  - Starting point (injecting a solution)
  - Search phases (partition of the decision variables)
- You can write your own constraints or search in C++, but this is seldom needed in an industrial context

Under the hood

Two iterative methods are interleaved: **LNS** for producing good quality solutions and **FDS** for proving infeasibility

Large-Neighborhood Search (LNS) [4]

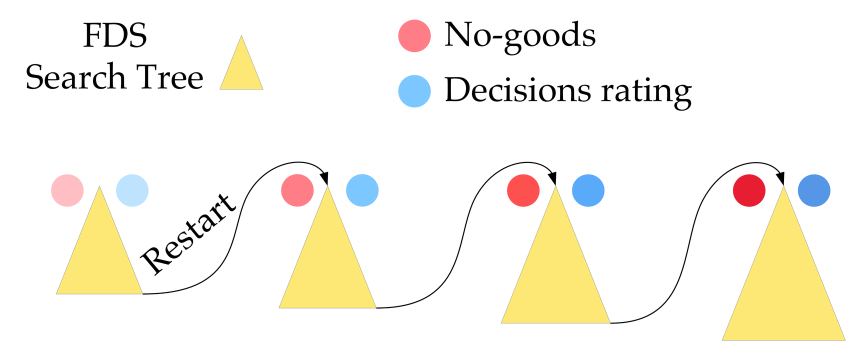
- Based on a Partial Order Schedule (POS) computed from feasible solutions
- A POS is a precedence graph such that the resource constraints (on sequence, cumul and state functions) are necessarily satisfied



- At each iteration, a fragment of the POS is relaxed and re-optimized
- The completion strategy is guided by a linear relaxation of the problem [7] and by exploiting objective landscapes [8]

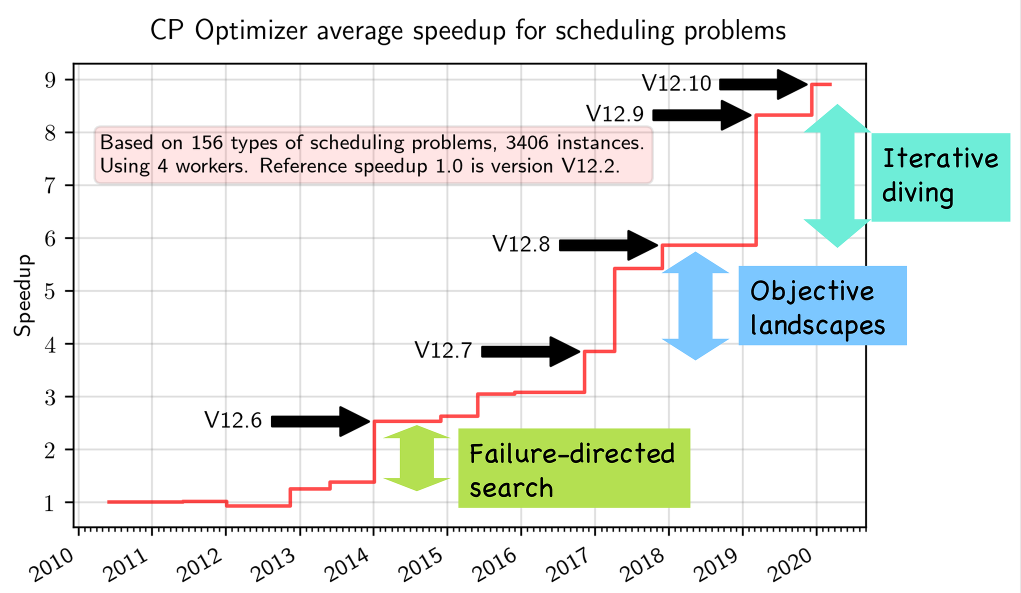
Failure-Directed Search (FDS) [5]

- Uses strong propagation [1,2,6]
- 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 iterations
- FDS uses no-goods to avoid revisiting already explored parts of the search space



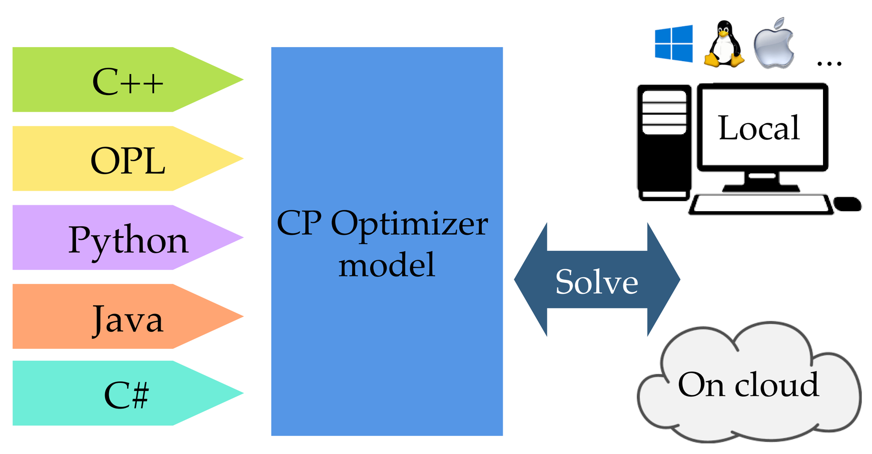
Performance

- Performance is on a par or outperforms state-of-the-art problem specific algorithms for most of classical scheduling problems like different variants / extensions of job-shop and RCPSP problems [3,4,5]
- Recent improvements on large problems (1.000.000 tasks)
- Performance is continuously improved



APIs and Tools

- APIs



- Tools
  - Human readable I/O format
  - Conflict refiner [9]: tells you why a model has no solution
  - Parametrizable search log for understanding the behavior of the search

References

1. Reasoning with Conditional Time-Intervals. FLAIRS-2008.
2. Reasoning with Conditional Time-Intervals, Part II: an Algebraical Model for Resources. FLAIRS-2009.
3. IBM ILOG CP Optimizer for Detailed Scheduling Illustrate on Three Problems. CPAIOR-2009.
4. Self-Adapting Large Neighborhood Search: Application to Single-Mode Scheduling Problems. MISTA-2007.
5. Failure-Directed Search for Constraint-Based Scheduling. CPAIOR-2015.
6. Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources. CPAIOR-2011.
7. Temporal Linear Relaxation in IBM ILOG CP Optimizer. Journal of Scheduling 19(4), 391-400, 2016.
8. Objective Landscapes for Constraint Programming. CPAIOR-2018.
9. An Optimal Iterative Algorithm for Extracting MUCs in a Black-box Constraint Network. ECAI-2014.
10. IBM ILOG CP Optimizer for Scheduling. Constraints Journal 23(2), 210–250, 2018.