Brunel BSEL/IDA seminars, 6 November 2019

Brunel University London

Constraint Satisfaction Problems and Constraint Programming

Gabriel Scali

Constraint Satisfaction Problems

Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is a tuple (X,D,C) where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is the set of variables
- $D = \{d_1, d_2, \ldots, d_n\}$ is the set of domains (d_i is a finite set of potential values for x_i)
- $C = \{c_1, c_2, \ldots, c_m\}$ is a set of constraints restricting the values that the variables can simultaneously take.

Values need not be a set of consecutive integers, they just need to be a discrete set.

A solution to a CSP is an **assignment** of a value from its domain to every variable, in such a way that all constraints are satisfied at once.

Just any solution vs all solutions vs a good/optimal solution.

In this last case, we have a **constrained optimisation problem** (COP), defined by some objective function of some (or all) of the variables

Complexity of CSPs

CSPs are in general NP-complete.

This easily follows by a number of other NP-complete problems being expressible as constraint satisfaction problems: for example the propositional satisfiability (SAT) problems.

Research has shown that some subclasses of CSPs with certain properties are polynomial if they follow certain restrictions. For example an instance where all the domains are binary and all the constraints are binary, can be solved in polynomial time.

COPs are harder than their corresponding CSPs: knapsack and TSP are reducible to COPs.

Puzzles such as the n-queens problem or sudoku are often used as toy examples.

Why are CSPs and CP interesting?

CSPs and COPs appear in all sorts of fields: scheduling, planning, resource allocation, transport and logistics, network management, economics, biology and many more.

Unlike it happens for other areas of AI, this is one where computers, while struggling, are still much better than people!

In the holy wars between different programming paradigms, declarative and constraint programming are often unjustly neglected - but there are strong arguments in favour of them, especially in terms of the *software engineering* practices they enable.

Solving CSPs : backtracking

Brunel University London

Finding a satisficing assignment is a search problem on (possibly infinite) tree.

The simplest CSP-solving technique is backtracking, that consists in incrementally extending a partial solution that specifies consistent values for some of the variables, towards a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution.

This is often not very effective for a few reasons:

- repeated failures due to the same reason;
- conflicting values of variables are not remembered;
- conflicts are not detected before they really occur.

Solving CSPs : constraint propagation

A better method for solving CSPs is based on removing inconsistent values from variables' domains as new values are assigned. These *consistency techniques* were introduced first in the scene labelling problem (1975) and are based on the concept of constraint propagation.

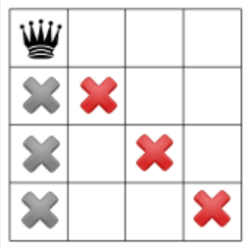
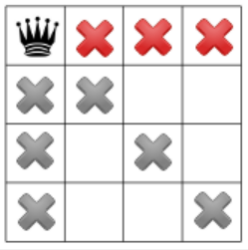
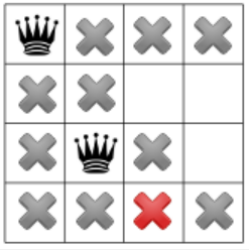
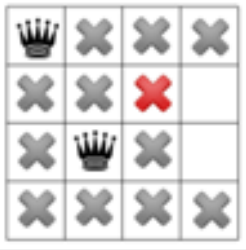
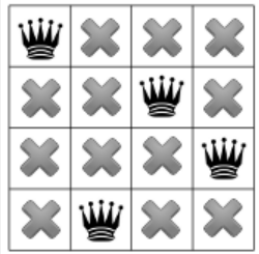
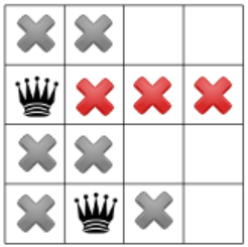
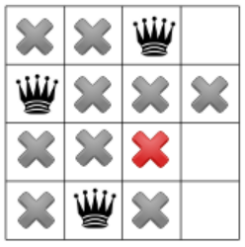
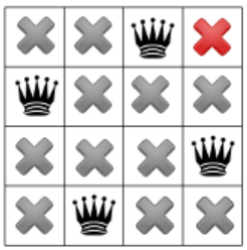
CSP can be represented as a constraint graph where nodes correspond to variables and edges are labelled by constraints.

The most widely used consistency technique is called **arc consistency** (AC). It removes inconsistent values from variables' domains based on binary constraints. In particular, the arc (V_i,V_j) is arc consistent if and only for every value x in the current domain of V_i which satisfies the constraints on V_i there is some value y in the domain of V_j such that V_i=x and V_j=y is permitted by the binary constraint between V_i and V_j.

Other, more sophisticated algorithms have been introduced over the years, based on things like path consistency, lookahead, and look back techniques such as backjumping or backmarking.

N-Queens problem and constr. propag.

Brunel University London

Solving CSPs : beyond systematic search

Brunel University London

In most problems encountered in real-world applications systematic searches are unfeasible, even using the mentioned techniques.

There are three alternatives:

Construction heuristics: policies for choosing which variable to assign next and which available value to assign next based on the nature of the problem.

Local search: start with an unfeasible solution and use a *repair* metaphor to improve it. E.g. hill-climbing or tabu search. LS usually has some form of randomisation to get out of local minima.

Global search: e.g. genetic algorithms, look at the space of assignments as a whole.

Constraint Optimisation Problems

Brunel University London

- COPs are even harder than CSPs.
- The most famous systematic algorithm is Branch and Bound.
- B&B needs a heuristic function that assigns a numerical value to a partial labelling. The value represents an underestimate (for minimisation) of the objective function for the best complete labelling obtainable from the partial labelling. Solutions are searched depth first like with backtracking, but as soon as a value is assigned to a variable, the value of heuristic function is computed. If it exceeds the bound, then the sub-tree under the current partial labelling is pruned. Initially, the bound is set to infinity and during the computation it records the value of best solution found so far.
- The efficiency of B&B is determined by two factors: the quality of the heuristic function and whether a good bound is found early.
- Again, systematic search not usually feasible in real problems, so a number of heuristic and metaheuristic techniques are the subject of much research.

Constraint Programming

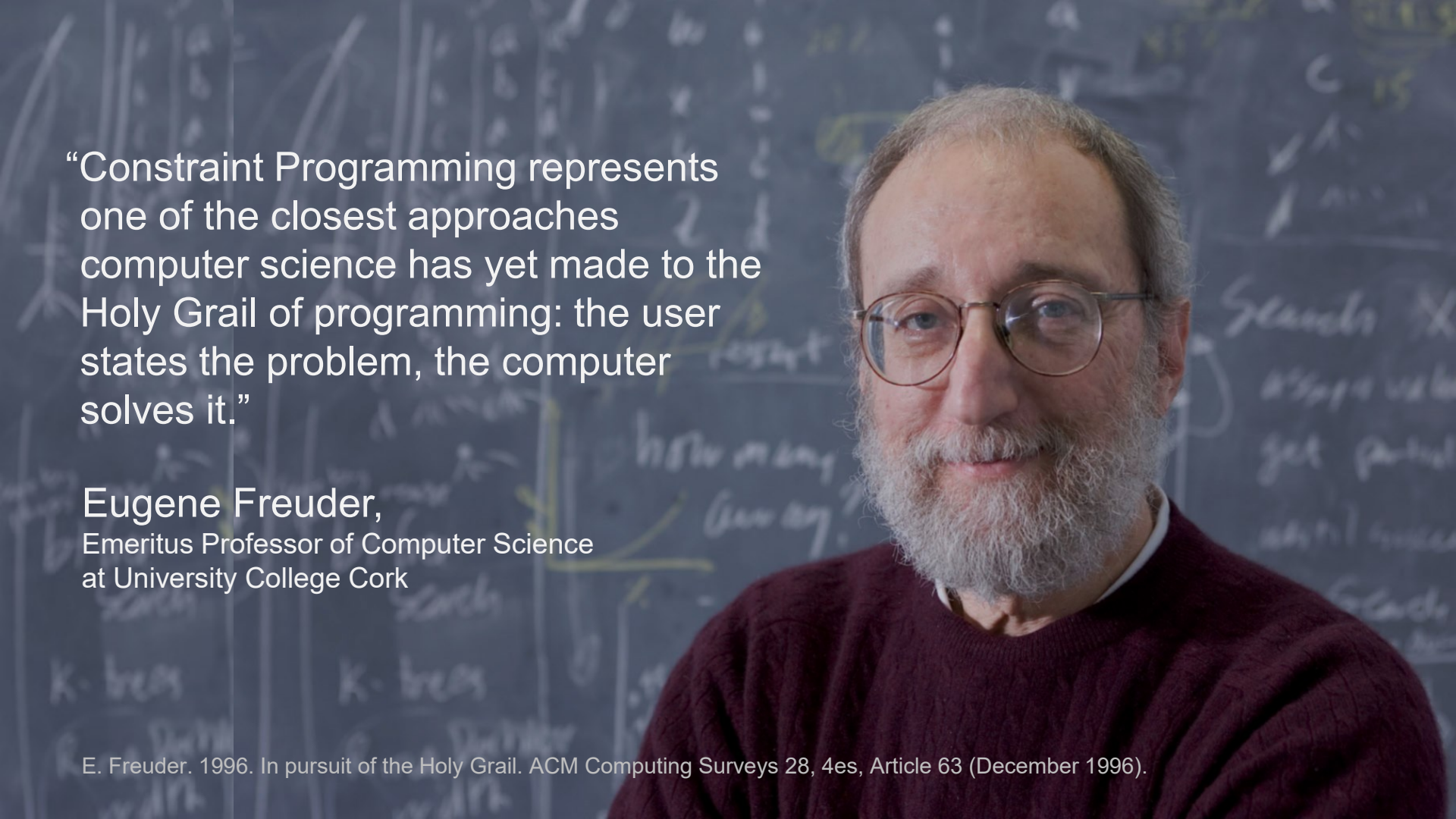
Google or-tools and other solvers

Constraint programming is the study of computational systems based on constraints.

The idea of constraint programming is to solve CSPs and COPs by just stating variables with their domains, and constraints (requirements) about the problem area.

Then letting a generalised solver find solution satisfying all the constraints.

This *paradigm* has been first incorporated as an extension to logic programming languages, originating Constraint Logic Programming, but adding some sort of constraint declaration extension to all sorts of languages of various paradigms.



E. Freuder. 1996. In pursuit of the Holy Grail. ACM Computing Surveys 28, 4es, Article 63 (December 1996).

Google or-tools and other solvers

Tools available today include:

- Or-tools
- IBM CPLEX
- Gecode
- Chocosolver
- ECLiPSe CLP
- JaCoP
- SICStus Prolog
- Oz
- MiniZinc and FlatZinc

N-Queens problem in or-tools

Brunel University London

```
def main(n=8):
  # Create the solver.
  solver = pywrapcp.Solver("n-queens")

  #
  # data
  #
  # n = 8 # size of board (n x n)

  # declare variables
  q = [solver.IntVar(0, n - 1, "x%i" % i) for i in range(n)]

  #
  # constraints
  #
  solver.Add(solver.AllDifferent(q))
  for i in range(n):
    for j in range(i):
      solver.Add(q[i] != q[j])
      solver.Add(q[i] + i != q[j] + j)
      solver.Add(q[i] - i != q[j] - j)

  # for i in range(n):
  #     for j in range(i):
  #         solver.Add(abs(q[i]-q[j]) != abs(i-j))
```

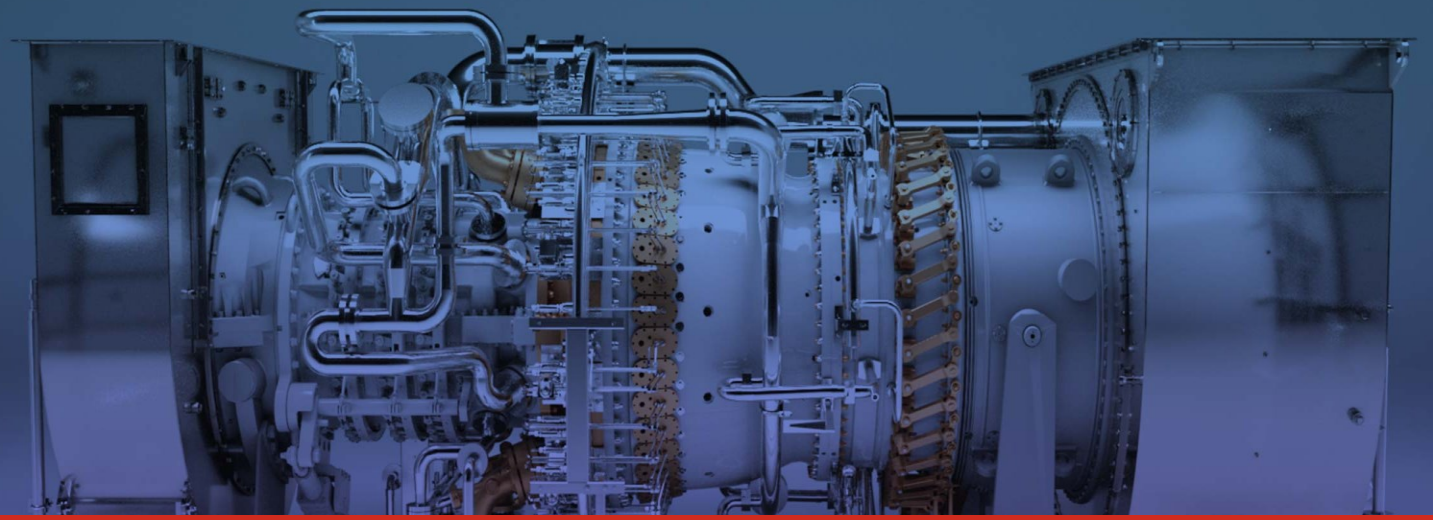
```
#
# solution and search
#
solution = solver.Assignment()
solution.Add([q[i] for i in range(n)])

collector = solver.AllSolutionCollector(solution)
# collector = solver.FirstSolutionCollector(solution)
# search_log = solver.SearchLog(100, x[0])
solver.Solve(solver.Phase([q[i] for i in range(n)],
                          solver.INT_VAR_SIMPLE,
                          solver.ASSIGN_MIN_VALUE),
             [collector])
```

The reality of CP

Most real problems are subject to ulterior complications, and things aren't easy as Freuder hoped:

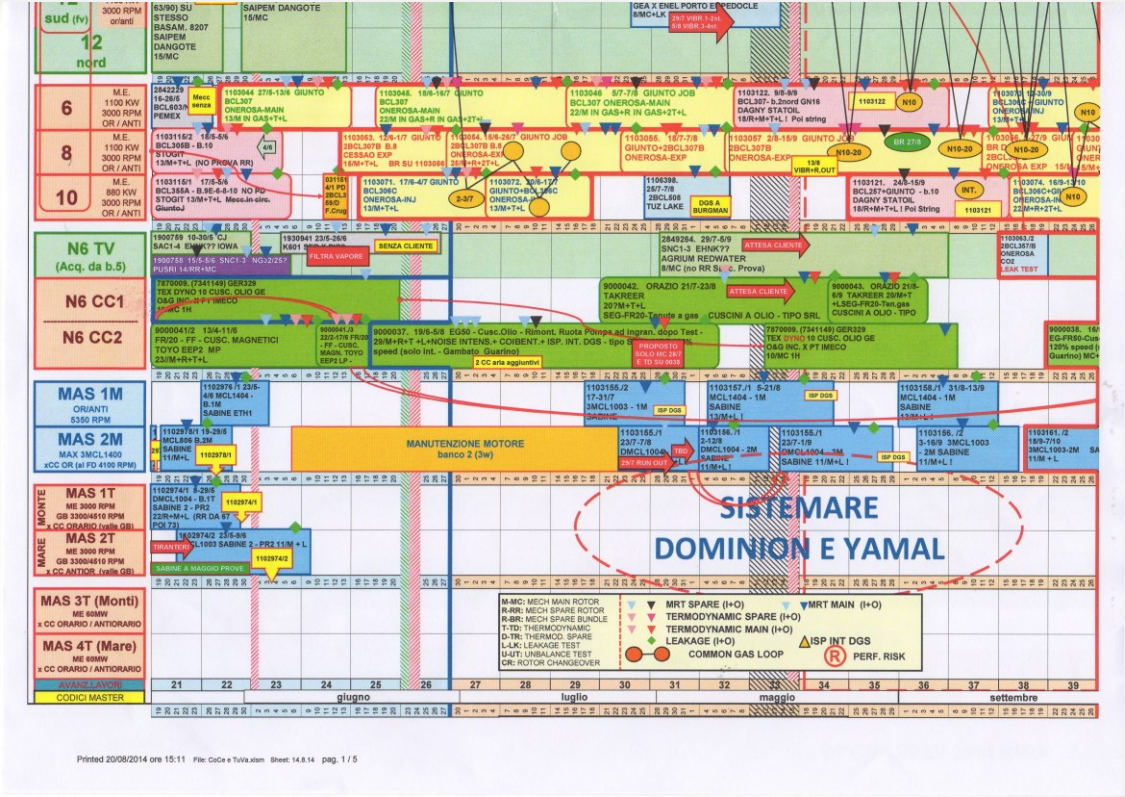
- soft constraints
- multi-objective optimisation
- infinite domains such as time
- complex discrete types, like intervals

- Modelling is actually very hard and different models determine how effective a search method or heuristic can be.



A case study: General Electric Turbines

General Electric – planning the tests



Tests were planned with an excel file (left)

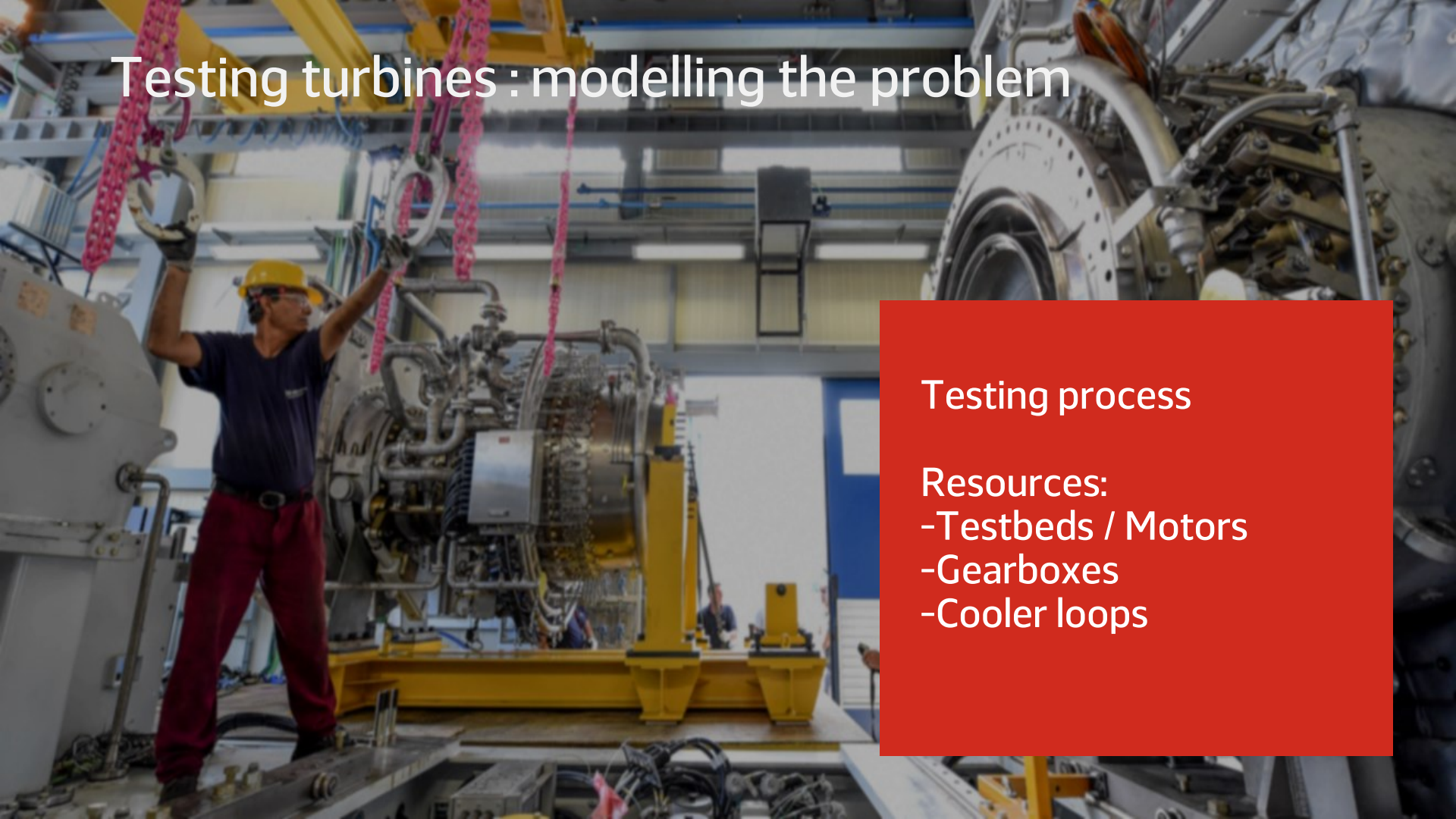
Changes were very difficult to accommodate

There was no warranty of correctness

Trial and error and what-if scenarios were impossible

Operators were not able to optimise solutions, and a lot of time and resources were wasted

Testing turbines : modelling the problem



Testing process

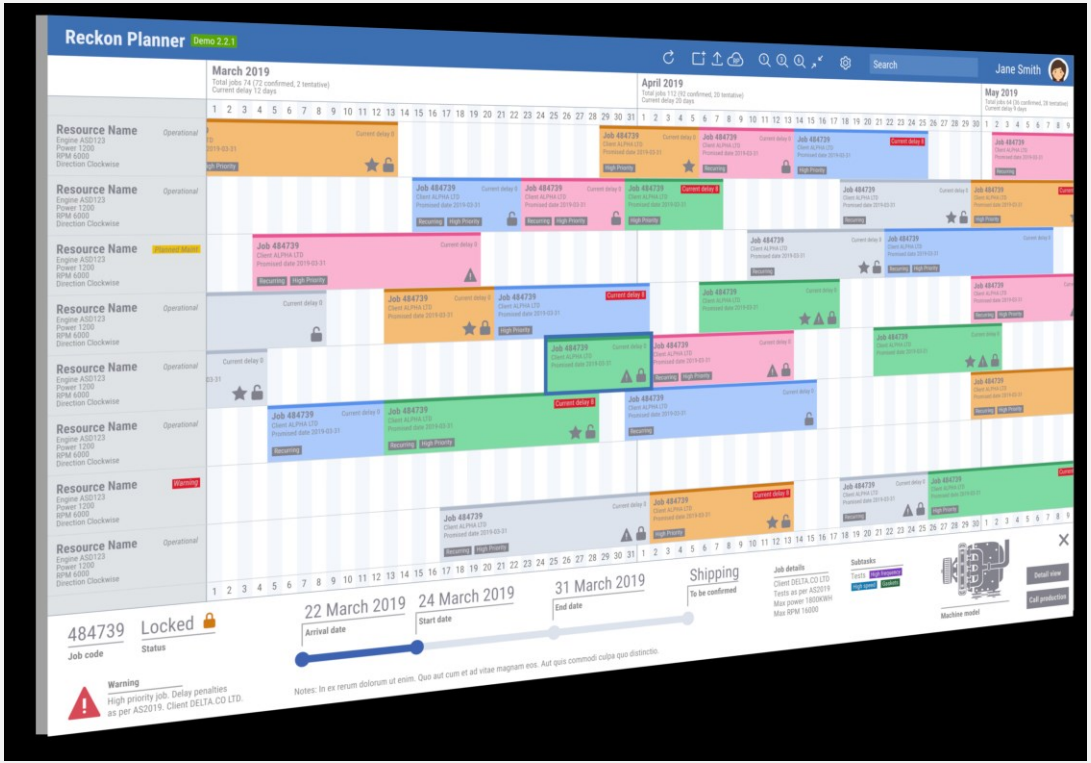
Resources:
-Testbeds / Motors
-Gearboxes
-Cooler loops

Testing turbines : modelling the problem

Matching Constraints

A parallel machines, offline, non-preemptive multi-machine with processing sets scheduling problem with resource eligibility, release dates and due dates. (using Graham's notation)

Stratagem



Already doing much better than humans and in use at GE for the past 4 years.

Uses a mix of or-tools and problem-specific construction heuristics.

Two areas of intense research:

- Local search : problem of efficiently constructing neighbours of an assignment
- Global search : problem of constructing viable operators on an assignment

To know more

Apt, Krzysztof (2003). Constraint Programming. Cambridge

Frühwirth & Abdennadher (2003). Essentials of Constraint Programming. Springer

Pinedo, Michael L. (2012). Scheduling. Theory, Algorithms, and Systems. Springer

Brucker, P & Knust, S. (2006). Complex Scheduling. Springer

Google or-tools: https://developers.google.com/optimization

Thanks.

@gabscali
gab@acm.org