Design Choices and Functionality Definition for ENTIRE EDIH Test before Invest Demonstrator Scheduling

Liam O'Toole and Helmut Simonis
School of Computer Science and Information Technology
University College Cork
Cork, Ireland
helmut.simonis@insight-centre.org

September 17, 2024

Abstract

This document describes the design alternatives and choices made for the demonstrator software used in the "test before invest" service for scheduling in the ENTIRE EDIH. We compare four possible system architectures, and justify our decision or a web-based application with a Java based back-end. We also describe the required functionality, first a minimal viable product definition to test the overall design and obtain some user feedback, then the full functionality intended for the actual service delivery.

1 Introduction

2 Architecture Design Alternatives

- Standalone MiniZinc
- Jupyter Notebook
- Desktop application
- Pure REST based solver
- Web-based aapplication

2.1 Standalone Minizinc Program

- very limited user experience

- target audience will not be familiar

- needs installation in customer machines

+ JSON Support in MiniZinc

- no support for dates/times
- Not all scheduling constraints available
- No support for commercial solvers

2.2 Jupyter Notebook

+ widely used for prototyping
+ very goodd tool support
- commercial tool support is an issue
- target audience probably is not familiar
- unfamiliar with Python interfaces of solvers/ MiniZinc Python interface
- may require installation on customer machine

2.3 Desktop Java Application

+ based on over ten years of experience developing prototypes
+ full featured JSON interfaces
+ sophisticated reporting/ presentation features available
- requires installation in customer machines
- licence management for commercial tools would be challenging
- in-app visualization not well supported
- testing issues when used by end-users

2.4 Pure REST-based Solver

+ no installation
- very limited user experience
+ close to a deployable application
- required integration into workflow to be practical
+ existing extension of framework as REST server
- scalability issue: computing power provided by UCC
- difficult to debug: not an issue for customers

2.5 Web-based app

+ no installation

+ potential to run on non PC platforms

+ easy to use commercial solvers

- security concerns/ access control

+ existing extension of framework for backend solver

- requires development/interface of visualization components

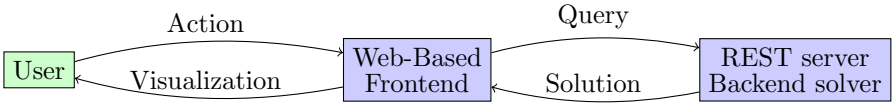
- scalability issue: computing power provided by UCC

- difficult to debug: not an issue for customers

3 Chosen Design

1. Back-end solver
    - Define JSON formats
    - Build data model to read/produce JSON files
    - Test harness : Problem generator
    - Write back-end solver based on MiniZinc
    - Write back-end solver based on CPOptimizer
2. Front-end
    - Load/Modify input data from file
    - Call solver
    - Consume results
    - Allow browsing of results
3. Integration
    - Test framework on developer machines
    - Production system on UCC/Insight servers
    - Access control mechanism for deployment

Figure 1: Chosen Design



4 MVP: Minimal viable product functionality

4.1 Constraints

- Product, order, job, task
- end-to-start temporal constraints
- Release date, soft due date
- intra-job linear sequence of tasks
- disjunctive machines
- cumulative machines
- machine choice
- WiP
- planned shutdown

4.2 Objective

- makespan
- total Lateness

4.3 Visualization

- Data lists
- Job Gantt Chart
- Machine Gantt Chart
- Resource Profiles

4.4 Solver Support

- MiniZinc (includes CP-SAT)
- CPOptimizer

5 Planned Full Functionality

5.1 Constraints

- alternative process paths
- producer/consumer (component stock, intermediate products
- calendars
- skill levels

5.2 Objective

- add earliness
- add flowtime
- multi-objective

5.3 Visualization

- add KPIs
- add Multi solution
- add solution comparison

5.4 Solver Support

- add CP SAT direct support (if feasible in Java)
- add OptalCP (if licence can be arranged)

6 Conclusion