TNO innovation for life

Production Line Performance Optimisation

TNO Public ⟩ TNO 2025 R11609
August 2025

ICT, Strategy & Policy
www.tno.nl
+31 88 866 50 00
info@tno.nl

TNO 2025 R11609 – August 2025

Production Line Performance Optimisation

Author(s)	Jacques Verriet
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Appendices	TNO Public
Number of pages	38 (excl. front and back cover)
Number of appendices	1
Programme name	AIMS
Project name	AIMS 2024
Project number	060.59493/01.01

The research is carried out as part of the AIMS program under the responsibility of TNO-ESI in cooperation with Canon Production Printing. The research activities are co-funded by Holland High Tech | TKI HSTM via the PPP Innovation Scheme (PPP-I) for public-private partnerships.

Contents

Abstract 4

1 Introduction 5
1.1 Research questions 5
1.2 Contribution 6
1.3 Outline 6

2 Domain model 7
2.1 Running example 8
2.2 Material language 9
2.3 Equipment language 11
2.4 Job language 13
2.5 Allocation language 15

3 Performance analysis 17
3.1 Constraint graphs 17
3.2 Constraint graph analysis 20

4 Performance optimisation 23
4.1 Constraint program 23
4.2 Allocation optimisation 30

5 Conclusion 32

References 33

Appendix A 34

Domain model 34

Abstract

To assess the practical performance of a system, one needs to consider the system in its operating environment. This report presents an approach for analysing and optimising the performance of High-Mix Low-Volume (HMLV) automated production lines. The approach comprises three methods: (1) a method to specify a production line's equipment, workload and allocation, (2) a method based on constraint graphs to compute the latency of a specified allocation, and (3) a method based on constraint programming to find the optimal allocation of a production line's workload on its equipment. Experiment results show that allocation analysis is feasible for large instances, but that allocation optimisation lacks scalability, i.e. it is only feasible for small instances. The report suggests alternative methods to improve this allocation optimisation scalability.

1 Introduction

Traditionally, the high-tech equipment industry is optimising the performance of the equipment they are developing. However, the actual performance of the equipment depends on the systems-of-systems context in which it operates. In the context of high-tech equipment, these systems of systems often involve manufacturing systems with equipment from various suppliers.

The AIMS program is a collaboration of TNO-ESI and Canon Production Printing to address analysing and optimising the (timing) performance of manufacturing system of systems. This report considers a specific type of manufacturing systems, namely automated production lines. We defined a *production line* as a composition of multiple systems that are coupled mechanically, i.e. material handling and transports do not involve human operators.

The AIMS program is particularly studying production systems with a highly variable workload. Such systems are called High-Mix Low-Volume (HMLV) production systems, as they produce a high variety of products in small series.

1.1 Research questions

The AIMS 2024 project addresses three research questions:

- RQ1. What is an appropriate way to describe a HMLV production line's equipment, its workload, and the allocation of this workload onto the line's equipment?
- RQ2. How can the latency of a specific allocation of a HMLV production line's workload onto its equipment be computed in an fast and accurate manner?
- RQ3. How can a (near-)optimum allocation of a HMLV production line's workload into its equipment be found (in reasonable time)?

Research questions RQ1 and RQ2 have partially been answered in the AIMS 2023 project. In AIMS 2023, a domain model was developed to describe a (HMLV) production line's equipment and workload and the allocation of the workload on the equipment. This domain model allows fast and accurate analysis of the timing of the specified allocation [1].

Unfortunately, the AIMS 2023 domain model [1] has several limitations with respect to addressing research questions RQ1 and RQ2:

1. The AIMS 2023 domain model does not support inline assembly and disassembly operations. For instance, one cannot describe inline cutting of material.
2. The job specifications of the AIMS 2023 domain model are quite long, because one needs to specify the operations of every individual piece of material.

With respect to RQ3, the AIMS 2023 domain model [1] does not provide any support; it can only be used for manual optimisation. Because the domain model does not include dependencies between job parts, the domain model's user is responsible for guaranteeing the feasibility of allocations of a job onto the available equipment.

1.2 Contribution

This report presents an update of the AIMS 2023 domain model, which addresses the mentioned limitations with respect to RQ1 and RQ2 and supports performance optimisation, i.e. provides a solution for RQ3.

Compared to the AIMS 2023 domain model, the AIMS 2024 domain model has the following improvements:

1. It has concepts for (inline) assembly and disassembly operations.
2. It has concepts to express the sequence dependencies of parts of jobs.
3. It allows shorter job specifications.
4. It supports automatic identification of the optimum allocation of a job set onto available optimisation.

A fifth contribution involves a visual tool to address research questions RQ1 and RQ2. PLOT (Production Line Optimization Tool) [2] has been developed to provide a graphical frontend for the textual domain model developed in AIMS 2024. Besides a graphical representation of the domain model, it also provides a frontend for the corresponding performance analysis.

1.3 Outline

The updated domain model is described in Chapter 2, which shows both its textual syntaxes in Xtext and the corresponding visual representations in PLOT. Chapter 3 explains how the domain model can be used to analyse specific allocations of a job set onto available equipment; this includes the graphical visualisation using PLOT. Chapter 4 does the same for identifying the optimum allocation, but it lacks graphical support. The report's summary and conclusions can be found in Chapter 5. Appendix A presents the syntax of the domain model's languages.

2 Domain model

This chapter describes the AIMS 2024 domain model. The domain model consists of four languages (see Figure 2.1):

1. The material language describes a production line's materials and the operations that can be performed on these materials.
2. The equipment language describes the production line's modules and their connections. The modules' descriptions include the time they need to prepare for operations and to execute operations.
3. The job language describes the work that a production line needs to perform.
4. The allocation language describes how the work is performed by the production line. This involves sequences of operations assigned to the production line's modules.

Figure 2.1 shows the two main analyses supported by the domain model.

1. *Single allocation analysis:* The left path from the allocation model involves the analysis of a fully specified allocation using constraint graphs. This, typically very fast, single allocation analysis is explained in Chapter 3.
2. *Allocation optimisation:* The right path from the allocation model corresponds to automatically finding the optimum allocation of a job set onto a production line's modules using constraint programming. This, typically lengthy, allocation optimisation is explained in Chapter 4.

Both analyses lead, if given sufficient time, to a schedule describing the execution of a production line's jobs on its equipment modules.

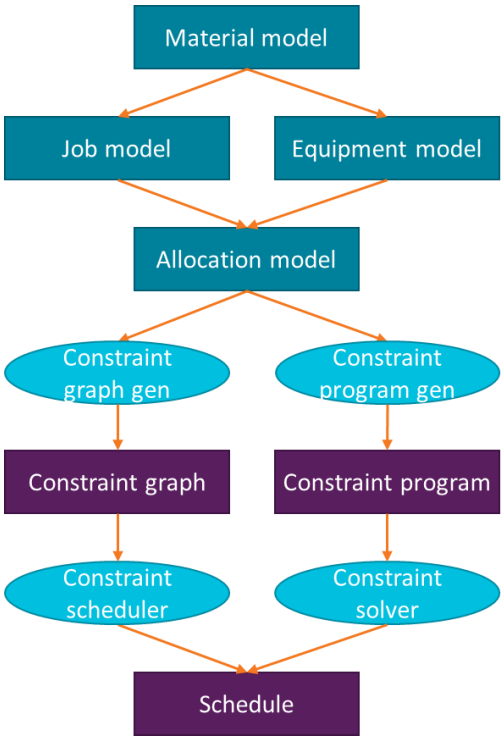

Figure 2.1: Production line optimisation context

Most elements in Figure 2.1 come with both a textual and a graphical representation. The (textual) languages have been defined using Xtext [3] and have a visual representation in PLOT [2]. The detailed syntax of the languages can be found in Appendix A. The analyses involve transformations defined using Xtend [4]; they can be run from the Eclipse IDE. PLOT [2] provides a graphical frontend for the single allocation analysis: it allows performing the analysis and visualising the analysis results.

2.1 Running example

We will describe domain model's languages using a running example, which involves a fictitious cake bakery and decoration system. The corresponding production line consists of seven modules (see Figure 2.2):

1. An *oven* which produces batches of cakes.
2. An *input buffer* in which baked cakes are stored.
3. A *slicer* in which a cake in sliced into a top and bottom half.
4. A *top decorator* where a cake's top half gets decorated.
5. A *bottom decorator* where a cake's bottom half gets decorated.
6. An *assembler* where decorated tops and bottoms are reassembled into a cake.
7. An *output buffer* where decorated cakes are stored.

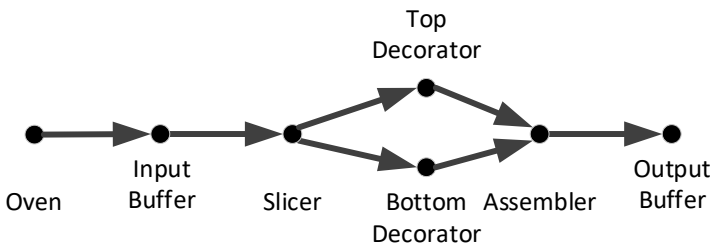

Figure 2.2: Cake decoration running example

For the running example, we will use a cake decoration scenario. The scenario assumes that a batch of cakes have already been baked and these need to be decorated before they get shipped. Decorating a cake involves cutting it into a top and a bottom slice, which are decorated separately. After the slices have been decorated, the decorated slices are combined to obtain a decorated cake, which can be shipped to a customer. As shown in Figure 2.1, four elements are needed to describe a scenario. Sections 2.2, 2.3, 2.4, and 2.5 describe these four elements for the cake decoration scenario.

2.2 Material language

Material models consist of three elements, which are specified in a single file with a .machine extension: material types, material instances, and operations. The material language distinguishes basic material types and composite material types. The latter consist of other material types.

The material types of the running example only involve basic materials: cakes and slices. The syntax in Appendix A shows how composite materials can be defined. The running example's material types are defined using the following (textual) specification:

```
basic material type Cake

basic material type Slice
```

The bold words **`basic`**, **`material`** and **`type`** represent the language's (generic) keywords. The normal text `Cake` and `Slice`, corresponds to the (specific) parameter of the running example.

Material types describe a class of material instances, which are identical with respect to some characteristics, but different with respect to others. In the running example, we consider cakes (and slices) of three sizes. The following specification describes them.

```
basic material Cake20 type Cake
basic material Cake24 type Cake
basic material Cake28 type Cake
```

The different slices sizes are defined similarly. Optionally, one can specify the dimensions and weight of the material instances, but these are not used for the analyses explained in this report.

The operations in the material language are defined based on the material types that they take as input and the material types they produce as output. We distinguish types of operations based on their number of inputs and outputs:

1. *Storage operations* store material in a buffer. They have one input and no outputs.
2. *Retrieval operations* retrieve material from a buffer. They have no inputs and one output.
3. *Assembly operations* combine multiple materials into one. They have at least two inputs and one output.
4. *Disassembly operations* decompose one material into multiple. They have one input and multiple outputs.
5. *Normal operations* transform one material. They have one input and one output.

The following specification shows how the running example's operations are defined in the material language.

```
retrieval operation RetrieveCake
  input
  output Cake

disassembly operation SliceCake
  input Cake
  output Slice, Slice

normal operation DecorateSlice
  input Slice
  output Slice

assembly operation AssembleCake
  input Slice, Slice
  output Cake

storage operation StoreCake
  input Cake
  output
```

Besides the textual representation using Xtext In the graphical editor PLOT (Production Line Optimization Tool) [2], the material model is visualised as shown in Figure 2.3. The top part shows the operations, the bottom part the materials.

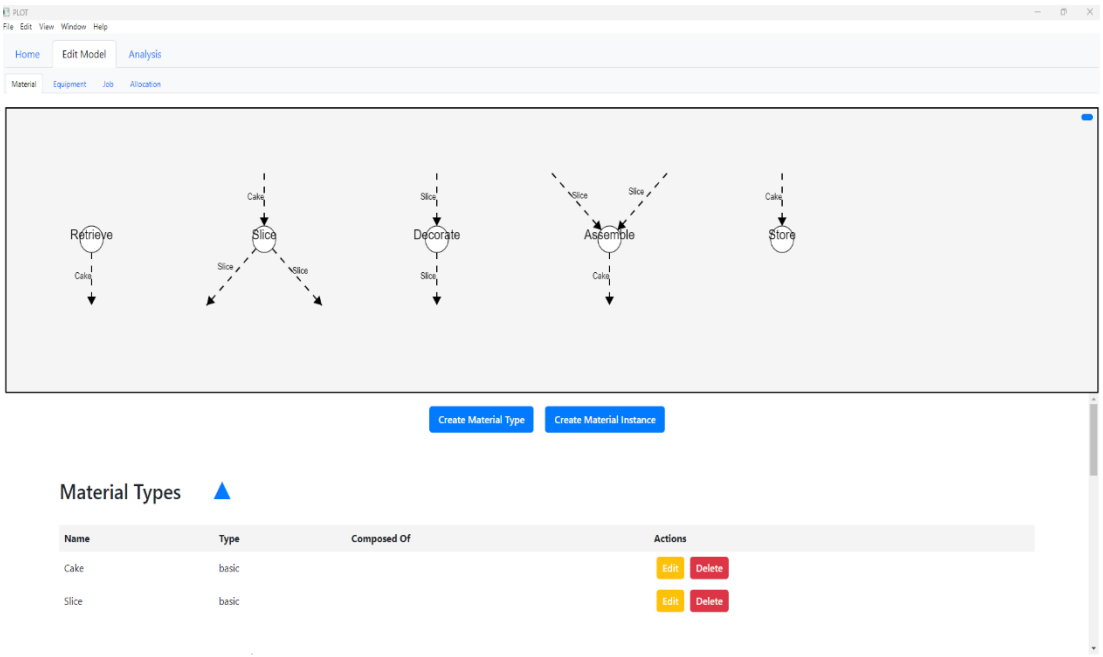

Figure 2.3: Material model in PLOT

2.3 Equipment language

The equipment language can be used to specify available equipment modules and their connections. The equipment is described as a directed graph, in which the nodes correspond to the equipment modules and the arcs to the (mechanical) connections between them.

Per module, one needs to specify the duration of its supported operations and the (sequence-dependent) setup times between its operations. The equipment language distinguishes four types of modules/nodes:

1. *Input nodes* correspond to points where material enters a production line.
2. *Output nodes* correspond to points where material leaves a production line.
3. *Buffer nodes* are locations where multiple pieces of material can be stored.
4. *Operation nodes* are locations where operations are performed and only the materials of the operation being performed can be stored.

The following specification describes two of the modules of the running example in the equipment language. Materials arrive at a node's inputs and leaves via its outputs; the operations perform the transformation from input material(s) to output material(s).

```
buffer node A_Input
  output out type Cake
  operation Retrieve
    output Cake20 duration 20 s
      after Retrieve output Cake20 setup 10 s
      after Retrieve output Cake24 setup 10 s
      after Retrieve output Cake28 setup 10 s
    output Cake24 duration 24 s
      after Retrieve output Cake20 setup 10 s
      after Retrieve output Cake24 setup 10 s
      after Retrieve output Cake28 setup 10 s
    output Cake28 duration 28 s
      after Retrieve output Cake20 setup 10 s
      after Retrieve output Cake24 setup 10 s
      after Retrieve output Cake28 setup 10 s

operation node B_Slicer
  input in type Cake
  output top type Slice
  output bottom type Slice
  operation Slice
    input Cake20 output Slice20, Slice20 duration 30 s
      after Slice input Cake20 output Slice20, Slice20 setup 5 s
      after Slice input Cake24 output Slice24, Slice24 setup 60 s
      after Slice input Cake28 output Slice28, Slice28 setup 60 s
    input Cake24 output Slice24, Slice24 duration 34 s
      after Slice input Cake20 output Slice20, Slice20 setup 60 s
      after Slice input Cake24 output Slice24, Slice24 setup 5 s
      after Slice input Cake28 output Slice28, Slice28 setup 60 s
    input Cake28 output Slice28, Slice28 duration 38 s
      after Slice input Cake20 output Slice20, Slice20 setup 60 s
      after Slice input Cake24 output Slice24, Slice24 setup 60 s
      after Slice input Cake28 output Slice28, Slice28 setup 5 s
```

The specification of an equipment node involves two types of durations: operation durations and setup times. *Operation durations* depend on the material being handled. The example above shows that the time of slice a cake depends on its size: slicing the smallest cake takes 30 seconds and slicing the largest takes 38 seconds. *Setup times* represent time that an equipment node to prepare for the next material. They depend on the equipment node's previous operation and the corresponding materials handled. In the running example, 5 seconds of setup is needed if the next operation involves the cake size, but 60 seconds of setup is needed if the cake size changes.

Per segment, one needs to specify the nodes that it connects and the travel time between these nodes. Below is a specification of one segment of the running example in the equipment language.

```
segment InputSlicer type Cake
  source node A_Input output out
  target node B_Slicer input in
  duration 10 s
```

Segments may have multiple sources and/or multiple targets. Such nodes are called *switches*, which may require time to change source or target node. The running example does not include switches.

The corresponding graphical visualisation in PLOT is shown in Figure 2.4. Note that the visualisation in PLOT is similar to the one shown in Figure 2.2. The black circles in Figure 2.4 allow visualisation of switches: they may have multiple incoming and multiple outgoing arcs.

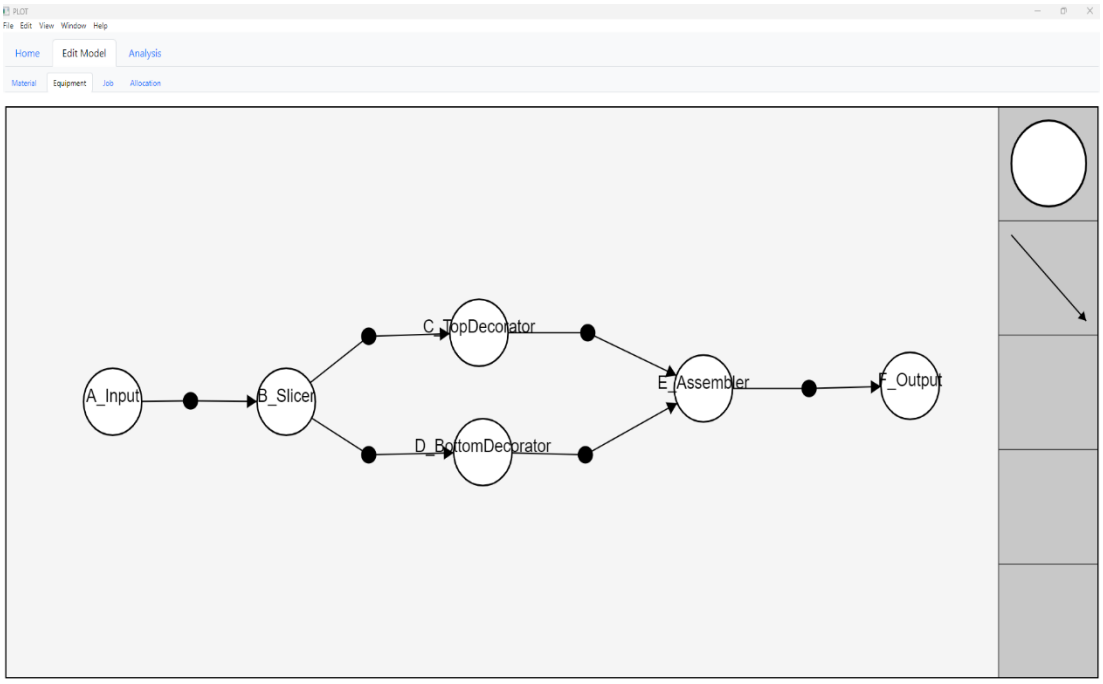

Figure 2.4: Equipment model in PLOT

A part of the equipment model, which is not shown in the running example, is the initial state of the equipment. This state is used to specify a starting point for timing analysis. It consists of the initial position of the switches and the operations that have completed and the time at which these were completed. This is used to keep the performance analysis described in Chapter 3 scalable.

2.4 Job language

The domain model's job language specifies the batch of work that is to be done. A batch consists of products consisting of parts. The job language specifies the recipe that leads to the production of the product parts. The following specification specifies the recipes needed to decorate one cake in the running example.

```
product cake1
  parts
    part preslice consisting of 1 Cake20
      operations Retrieve, Slice

    part top consisting of 1 Slice20
      operations Decorate

    part bottom consisting of 1 Slice20
      operations Decorate

    part assemble consisting of 1 Cake20
      operations Assemble, Store
```

One may need many parts to produce a product. To keep short job specifications, identical parts requiring the same recipe can be grouped. This is indicated by the multiplicity of the parts.

Except for the part recipes, the job specification also specifies the precedence dependencies between these recipes. To not overcomplicate the job language, part recipes are sequences of operation. Using the part dependencies, one can create more complex recipes. There are five types of recipe precedence constraints:

1. *Assembly dependencies* specify the dependency of inline assembly of multiple parts into one part.
2. *Disassembly dependencies* specify the dependency of inline disassembly of one part into multiple parts.
3. *Storage-retrieval dependencies* specify the dependencies between storing parts in a buffer and retrieving them later.
4. *Strict sequence dependencies* specify logical sequence of part execution with strict timing constraints, e.g. the time between the end of the source part and the start of the target part is fixed by the transportation times between the corresponding equipment nodes.
5. *Loose sequence dependencies* specify logical sequence of part execution with loose timing constraints, e.g. the time between the end of the source part and the start of the target part is unbounded.

As products can be produced independently of each other, dependencies between (parts of) different products are not supported by the job language. The following specification illustrates the dependencies between the cake decoration recipes.

```
part dependencies

  disassembly preslice -> top, bottom

  assembly top, bottom -> assemble
```

The corresponding visualisation in PLOT is shown in Figure 2.5. For two cakes, it shows the parts, i.e. the grey boxes with sequences of operations, and the sequence dependencies, i.e. the arcs, between these parts. The types of dependency are not shown explicitly.

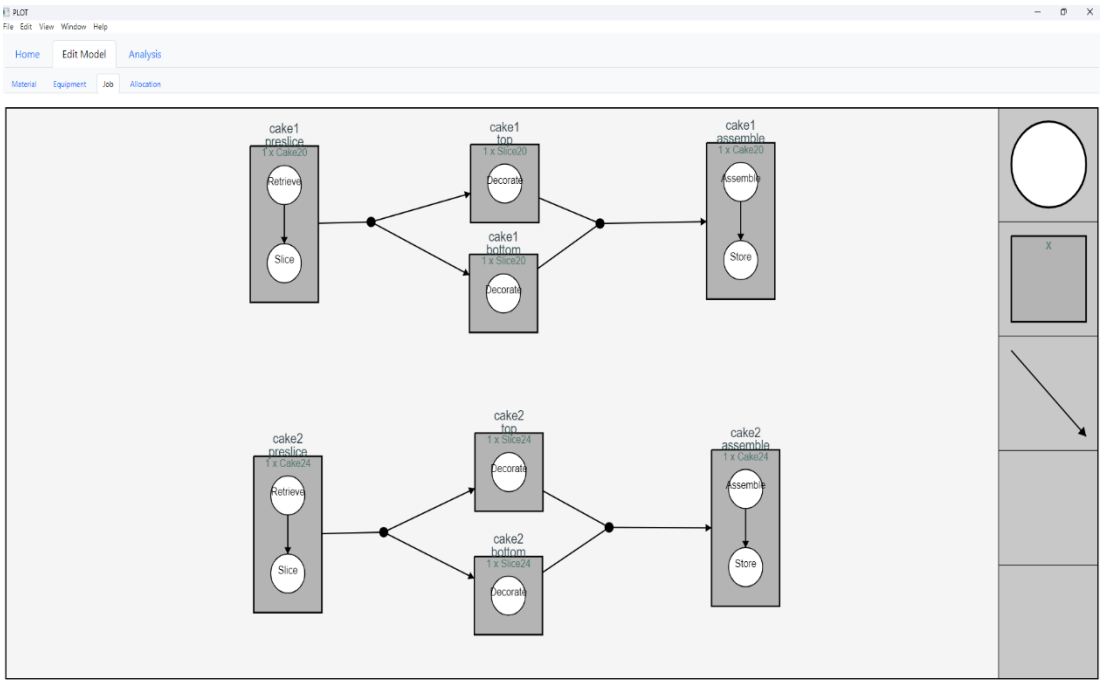

Figure 2.5: Job model in PLOT

2.5 Allocation language

The allocation language specifies which equipment that is used to execute a job's recipe and the sequence in which the recipes are to be executed. The following specification illustrates the allocation of the decoration recipes of a single cake.

```
product cake1
  part preslice
    A_Input:           Retrieve
    B_Slicer:          Slice

  part top
    C_TopDecorator:    Decorate

  part bottom
    D_BottomDecorator: Decorate

  part assemble
    E_Assembler:       Assemble
    F_Output:          Store
```

The corresponding allocation model in PLOT is shown in Figure 2.6. It (partially) shows the allocation of two cake decorations. Just as in the textual representation, the allocation is assumed to be done from top to bottom. PLOT allows easy reordering using drag-and-drop.

Figure 2.6: Allocation model in PLOT

If a job involves parts consisting of multiple sub-parts, i.e. part with a multiplicity of at least two, these are assumed to be allocated consecutively. This keeps the allocation specification short. If one would like to allocate the sub-parts differently, the individual sub-parts can be specified and allocated individually. This will, however, lead to longer job and allocation specifications.

3 Performance analysis

In Chapter 2, we explained the domain model, with which one can specify production line allocation scenarios. This chapter explains the timing performance analysis of specified scenarios. Section 3.1 explains constraint graphs, the formalism that is used for this analysis. Section 3.2 describes how constraint graphs are used to analyse a scenario.

3.1 Constraint graphs

The performance analysis entails the generation of a so-called constraint graph [5] [6]. A *constraint graph* is a directed graph whose nodes correspond to events. An arc between the events e_1 and e_2 represent temporal constraints between these events. The temporal constraints are of the form $t_2 \geq t_1 + \Delta$, where t_1 and t_2 represent the times at which events e_1 and e_2 occur and Δ represents the minimum time between the events.

Δ may have both positive and negative values. Positive values of Δ represent (relative) event *release dates* for the target event of an arc: event e_2 must occurs at least Δ time units after e_1. By specifying negative values of Δ, one can also specify maximum times between events. As $t_1 \leq t_2 - \Delta$, e_1 must occur at most $-\Delta$ time units before e_2. In other words, negative values of Δ specify *due dates* for the source event of a constraint graph arc.

This dual effect can be used to capture the duration of a operation. An operation of (positive) duration Δ gets represented by a start event e_1 and an end event e_2 and two arcs: an arc from e_1 to e_2 with weight Δ and an arc from e_2 to e_1 with weight $-\Delta$. Suppose e_1 occurs at time t_1 and e_2 at time t_2. The constraints of the arcs state that $t_2 \geq t_1 + \Delta$ and $t_1 \geq t_2 - \Delta$, which guarantees that $t_2 = t_1 + \Delta$.

The arc constraints along a path in a constraint graph add up: an arc of weight Δ_1 from e_1 to e_2 and an arc with weight Δ_2 from e_2 to e_3 specify a minimum delay $\Delta_1 + \Delta_2$ of between e_1 and e_3. As constraint graphs may be cyclic, events may have constraints with themselves. If a constraint graph does not have cycles with a positive total weight, the timing constraints can be satisfied. On the other hand, constraint graphs with positive-weight cycles represent constraints that cannot be satisfied.

A constraint graph for an instance of Chapter 2's running example involving a 20-centimetre cake followed by a 24-centimetre cake is shown in Figure 3.1. In this graph, the nodes represent events, which can be either the start or end of an equipment operation or the start or end of a transportation.

- The *green nodes* represent the starts of operations and the ends of transports.
- The *yellowish nodes* represent the ends of operations and the starts of transports.
- The *blue node* labelled ZERO represents an artificial event corresponding to the absolute time 0. It is used to correctly deal with the equipment's initial state.

Not shown in Figure 3.1 are nodes representing the history of the equipment, i.e. the end time of the last operations performed by the equipment nodes. Examples of such nodes can be found in the report describing the original domain model [1].

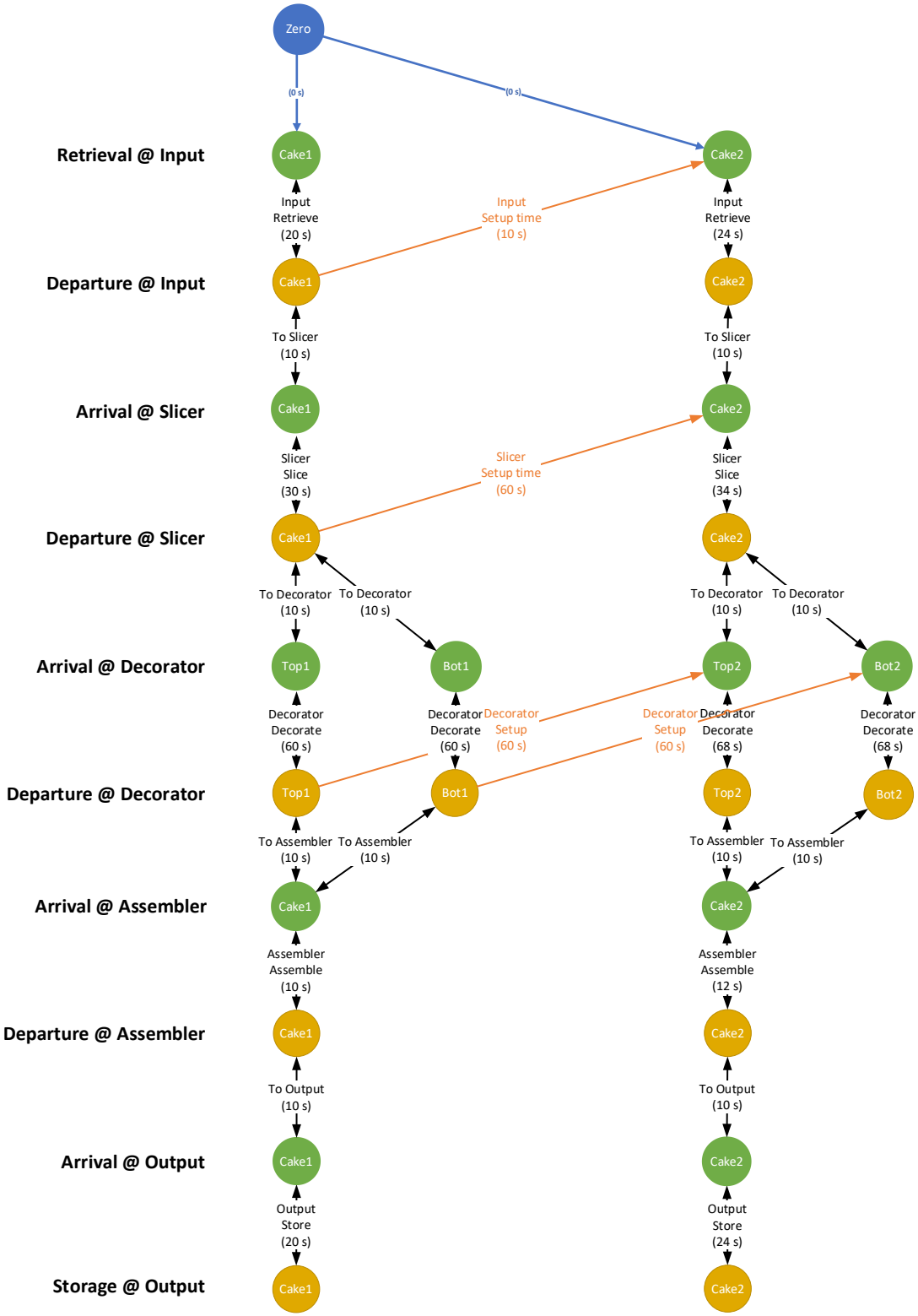

Figure 3.1: Running example constraint graph

The arcs represent the timing constraints between the corresponding events. Unidirectional arcs indicate that the target node's event must occur at least the arc weight time units after the source node's event. Bidirectional arcs indicate that the target node's event must occur

exactly the arc weight time units after the source node's event. The generated constraint graph contains seven types of arcs:

1. The timing constraints that originate from the tightly coupled equipment, which does not have buffer capacity, are captured by bidirectional arcs. These arcs represent either the start and end of an operation, or the start and end of a transportation. Examples are the black bidirectional arcs in Figure 3.1.
2. The timing constraints of storage-retrieval constraints are captured by unidirectional arcs. They specify that a retrieval operation must start after the completion of a storage operation. Figure 3.1 does not contain such arcs; examples can be found in the report describing the AIMS 2023 domain language [1].
3. Setup timing constraints are captured by unidirectional arcs capture the setup timing constraints. They specify the minimum time between the completion of one operation and the start of the next one on the same equipment node. The orange arcs in Figure 3.1 are examples of these arcs.
4. The timing constraints of switch segments is captured by unidirectional arcs: if two pieces of material follow a different path though a switch, then the time between completion of the first piece's path and the start of the second piece's must be at least the switch's switch time. Figure 3.1 does not contain such arcs; examples can be found in the report describing the AIMS 2023 domain language [1].
5. Bidirectional arcs between the ZERO nodes and the history nodes are used to exactly specify the times at which the (last) operations of the equipment node ended. Figure 3.1 does not contain such arcs; examples can be found in the report describing the AIMS 2023 domain language [1].
6. Unidirectional arcs from the ZERO node to the nodes corresponding to the first events of a part specify that no events may happen before the ZERO event. Examples are the blue arcs in Figure 3.1.

A visualisation of the same constraint graph in PLOT is shown in Figure 3.2. Note that the graph's structure is the same as that of the (manually drawn) graph in Figure 3.1.

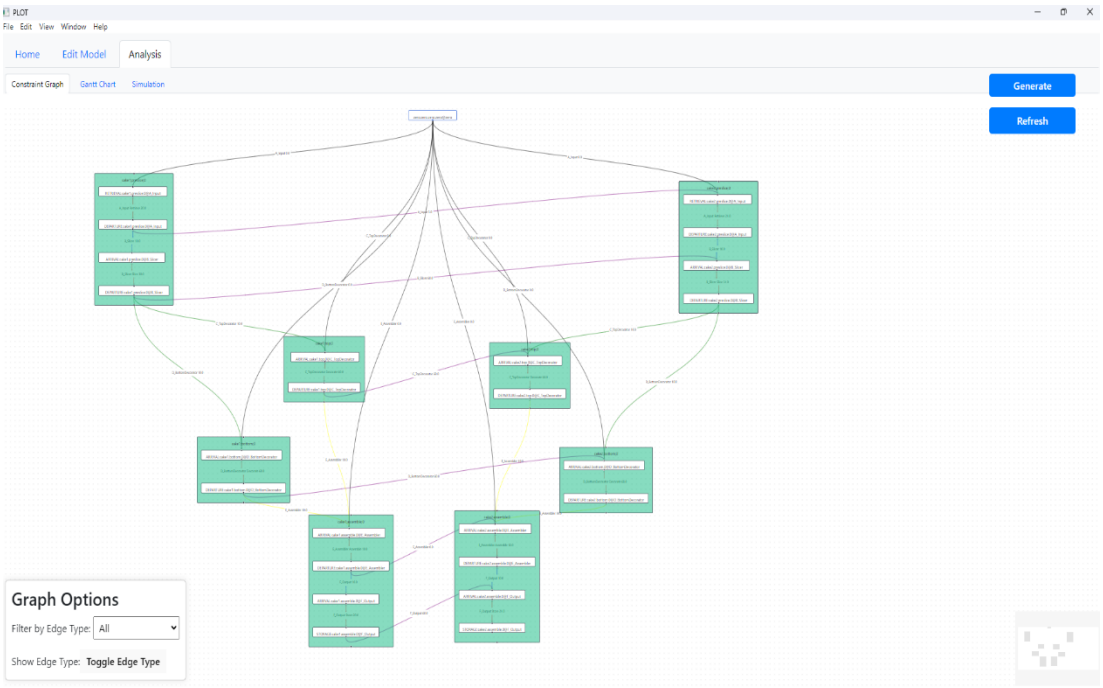

Figure 3.2: Constraint graph in PLOT

3.2 Constraint graph analysis

From the generated constraint graph, one can compute a schedule. This involves computing the longest paths from the ZERO node to all other nodes. In its general form, finding the longest (simple) path in a (weighted) graph is an NP-hard problem [7]. The longest path can however be found in polynomial time when a graph does not have positive cycles [8].

In constraint graphs, positive cycles describe infeasible constraints, and the absence of positive cycles means a feasible solution exists [5]. The fastest feasible solution can be found by the Bellman-Ford algorithm [8]. This algorithm either finds the shortest path from one node to all other nodes in a weighted graph (without negative cycles), or it detects the presence of a negative cycle. To make this algorithm find the longest paths from the ZERO node, all arc weights need to be negated.

The lengths of the paths found by the Bellman-Ford algorithm correspond to the earliest times at which events may happen. This can be translated into a Gantt chart showing the equipment's operations and the travel between these operations. The Gantt chart in Figure 3.3 shows the timing of consecutively decorating six cakes. The operations for one cake all have the same colour; one can clearly see the process that the cakes follow during decoration.

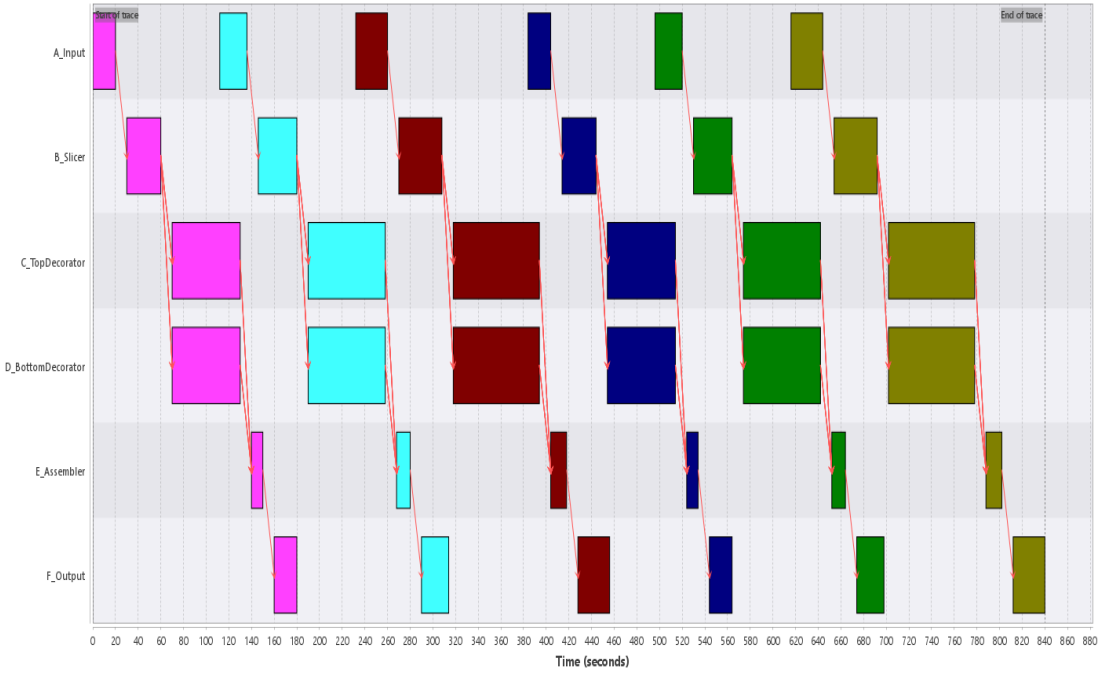

Figure 3.3: Cake decoration schedule

PLOT allows a Gantt to be animated: PLOT's animation shows the material flow over time and the corresponding active machines. A screenshot is shown in Figure 3.4.

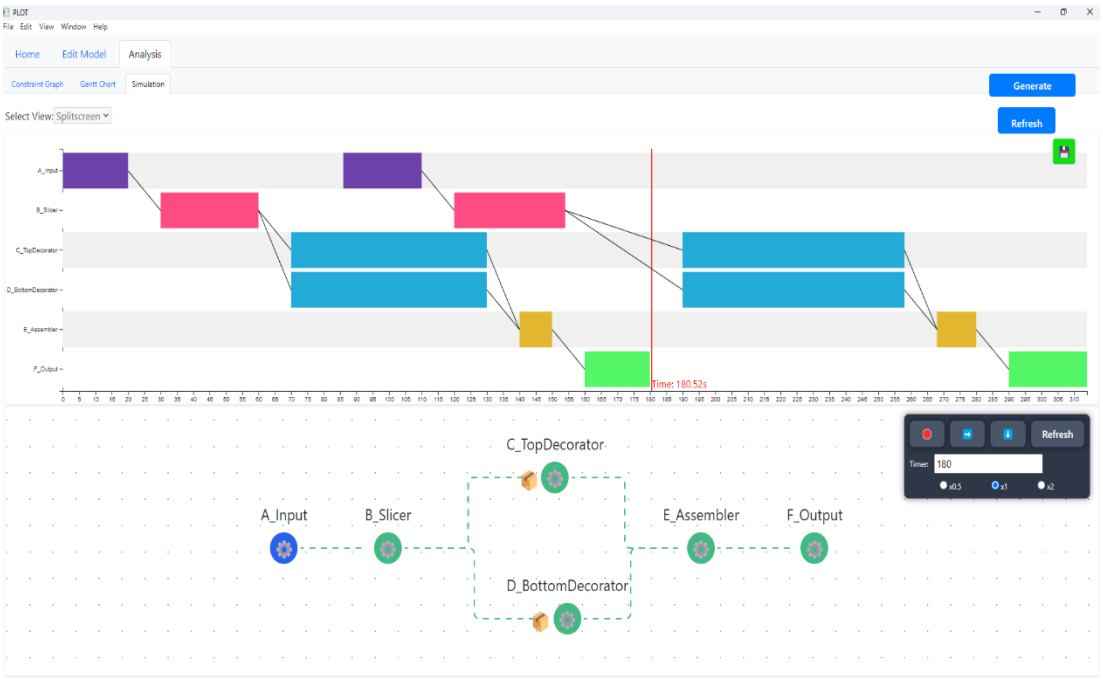

Figure 3.4: Gantt chart animation in PLOT

The Gantt chart in Figure 3.3 has a lot of idle time between the different cakes. This is because the cakes have different sizes, and the equipment nodes need setup to adapt to a new size. By reordering the allocation sequence such that cakes (and slices) of the same size are handled consecutively, one obtains the much shorter Gantt chart in Figure 3.5, which uses the same colours as Figure 3.3.

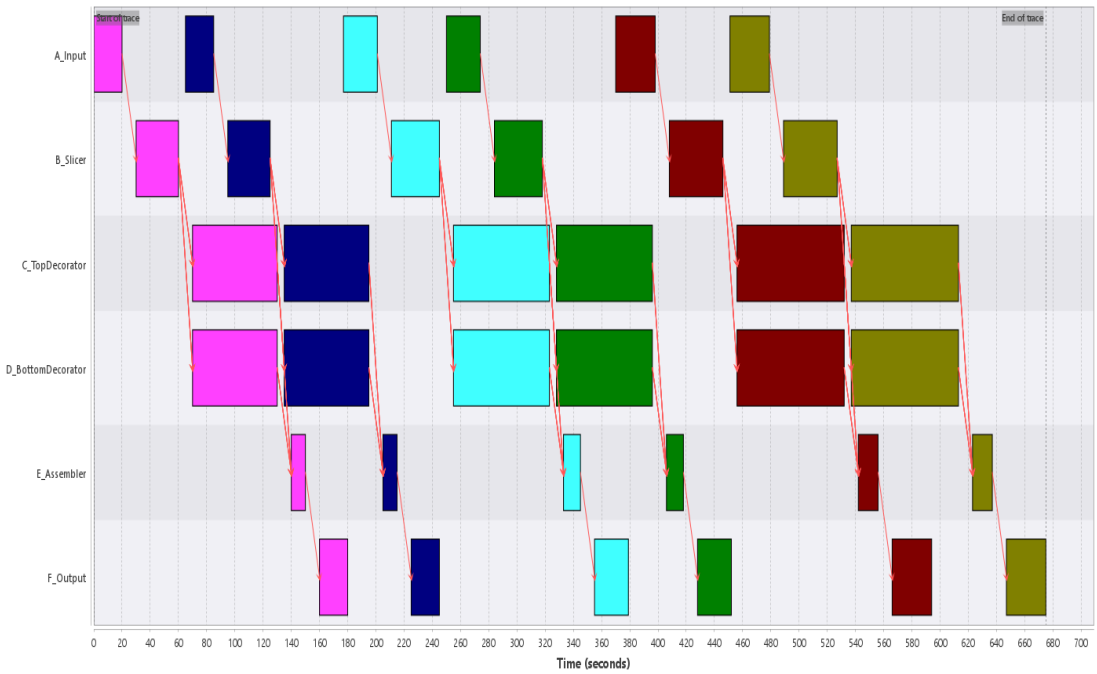

Figure 3.5: Optimised cake decoration schedule

Next, we assess the scalability of performance analysis using constraint graphs. We do this by increasing numbers of cakes in the running example introduced in Chapter 2. For the scalability assessment, we use laptop PC with an Intel Core i7-1255U processor and 16 GB of RAM running Windows 11 version 24H2. Table 3.1 shows the time needed to compute a schedule using constraint graph analysis and to generate the corresponding Gantt chart (file).

Table 3.1: Constraint graph analysis scalability assessment

Number of cakes	Constraint graph analysis time	Gantt chart writing time	Total time
100	0.04 s	0.6 s	0.6 s
200	0.2 s	2.3 s	2.5 s
300	0.3 s	5.4 s	5.7 s
400	0.6 s	10 s	11 s
500	1.0 s	16 s	17 s
600	1.4 s	23 s	24 s
700	2.0 s	31 s	33 s
800	2.7 s	49 s	52 s
900	3.6 s	64 s	68 s
1000	5.2 s	89 s	94 s

The results in Table 3.1 show that the creating a Gantt chart in the TRACE4CPS format [9] is the most time-consuming step. Computing a schedule using the Bellman-Ford algorithm [8] is quite fast for instances with at most a few hundred cakes, but then the analysis time starts to increase non-linearly. This is due to the fact the Bellman-Ford algorithm is a worst-case quadratic-time algorithm. The non-linear time increase may be avoided by using the recently developed expected linear-time algorithms for computing shortest paths for graphs with positive and negative arc weights [10] [11].

4 Performance optimisation

In Chapter 3, we have explained how the duration of production scenarios can be analysed using constraint graphs. As there are typically very many possible allocations, it is challenging to find an optimum or near-optimum allocation manually. In this chapter, we will address automatic allocation optimisation using constraint programming. The generation of a constraint program is explained in Section 4.1. Experimental optimisation results are presented in Section 4.2.

4.1 Constraint program

Constraint programming (CP) is a declarative programming paradigm used to solve complex computational problems, particularly combinatorial ones, by defining constraints that must be satisfied by the solution [12].

We will use the running example introduced in Chapter 2 to illustrate how an allocation optimising constraint program can be generated from the domain model also introduced in Chapter 2. The constraint program is specified in the MiniZinc syntax [13]. MiniZinc is a well-known frontend for multiple open-source and commercial constraint solvers.

The generation of the constraint program ignores most of the content of the input allocation model. The generation only uses the allocation model's batch and its equipment; all allocations specified by the user are ignored.[1]

The generated MiniZinc constraint program starts by defining an upper bound for the makespan/schedule length:[2]

```
% Maximum makespan
int: max = 1000000;
```

Next, it defines a data structure for the equipment nodes:

```
% Equipment nodes
enum NODE = {A_Input, B_Slicer, C_TopDecorator, D_BottomDecorator,
E_Assembler, F_Output};
```

The next part is an enumeration of all combinations of operations and their possible inputs and outputs and an array with the corresponding durations. The value 1,000,000 indicates that an operation cannot be performed by an equipment node.

[1] Including the allocations specified by the user is a straightforward extension of the generation process explained in this chapter.

[2] The maximum makespan is set to a constant value of 1,000,000 milliseconds or 1,000 seconds. This should be based on the domain model instance, e.g. by setting the maximum to the sum of all operation, setup and travel times.

```
% Operations
enum OPERATION = {Assemble__I_Slice20_Slice20__O_Cake20,
Assemble__I_Slice24_Slice24__O_Cake24,
Assemble__I_Slice28_Slice28__O_Cake28, Decorate__I_Slice20__O_Slice20,
Decorate__I_Slice24__O_Slice24, Decorate__I_Slice28__O_Slice28,
Retrieve__I__O_Cake20, Retrieve__I__O_Cake24, Retrieve__I__O_Cake28,
Slice__I_Cake20__O_Slice20_Slice20, Slice__I_Cake24__O_Slice24_Slice24,
Slice__I_Cake28__O_Slice28_Slice28, Store__I_Cake20__O, Store__I_Cake24__O,
Store__I_Cake28__O};

% Operation durations
array [NODE, OPERATION] of int: duration =
[|1000000, 1000000, 1000000, 1000000, 1000000, 1000000, 20000, 24000,
28000, 1000000, 1000000, 1000000, 1000000, 1000000, 1000000|
1000000, 1000000, 1000000, 1000000, 1000000, 1000000, 1000000, 1000000,
1000000, 30000, 34000, 38000, 1000000, 1000000, 1000000|
1000000, 1000000, 1000000, 60000, 68000, 76000, 1000000, 1000000, 1000000,
1000000, 1000000, 1000000, 1000000, 1000000, 1000000|
1000000, 1000000, 1000000, 60000, 68000, 76000, 1000000, 1000000, 1000000,
1000000, 1000000, 1000000, 1000000, 1000000, 1000000|
10000, 12000, 14000, 1000000, 1000000, 1000000, 1000000, 1000000, 1000000,
1000000, 1000000, 1000000, 1000000, 1000000, 1000000|
1000000, 1000000, 1000000, 1000000, 1000000, 1000000, 1000000, 1000000,
1000000, 1000000, 1000000, 1000000, 20000, 24000, 28000|];
```

The travel times are captured in a matrix with the equipment nodes on both axes. The value 1,000,000 is used for unconnected equipment nodes.

```
% Travel times
array [NODE, NODE] of int: travel =
[|0, 10000, 1000000, 1000000, 1000000, 1000000|
1000000, 0, 10000, 10000, 1000000, 1000000|
1000000, 1000000, 0, 1000000, 10000, 1000000|
1000000, 1000000, 1000000, 0, 10000, 1000000|
1000000, 1000000, 1000000, 1000000, 0, 10000|
1000000, 1000000, 1000000, 1000000, 1000000, 0|];
```

A three-dimensional array is used to capture the equipment nodes' setup times. As this array is large, only the setup times of one equipment are shown. If no setup times are specified in the equipment model, we assume no setup time, i.e. a value of 0. Below a part of a setup table is shown; it involves the setup times for the A_Input node.

```
% Setup times
array [NODE, OPERATION, OPERATION] of int: setup =
[||
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 5000, 5000, 5000, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 5000, 5000, 5000, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 5000, 5000, 5000, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0|,
…
```

The following data structures define the operations that are performed on all the job model's parts. The subpart operations have two indices, both starting from 0.

1. The first index is used for part consisting of multiple subparts. This does not apply to the running example; hence only the value 0 is used.
2. The second index corresponds to the position of an operation in the part's sequence. It allows multiple occurrences of the same operation in the same sequence.

```
% Subpart operations
enum SUBPART = {cake1_assemble_0_0_Assemble, cake1_assemble_0_1_Store,
cake1_bottom_0_0_Decorate, cake1_preslice_0_0_Retrieve,
cake1_preslice_0_1_Slice, cake1_top_0_0_Decorate,
cake2_assemble_0_0_Assemble, cake2_assemble_0_1_Store,
cake2_bottom_0_0_Decorate, cake2_preslice_0_0_Retrieve,
cake2_preslice_0_1_Slice, cake2_top_0_0_Decorate,
cake3_assemble_0_0_Assemble, cake3_assemble_0_1_Store,
cake3_bottom_0_0_Decorate, cake3_preslice_0_0_Retrieve,
cake3_preslice_0_1_Slice, cake3_top_0_0_Decorate,
cake4_assemble_0_0_Assemble, cake4_assemble_0_1_Store,
cake4_bottom_0_0_Decorate, cake4_preslice_0_0_Retrieve,
cake4_preslice_0_1_Slice, cake4_top_0_0_Decorate,
cake5_assemble_0_0_Assemble, cake5_assemble_0_1_Store,
cake5_bottom_0_0_Decorate, cake5_preslice_0_0_Retrieve,
cake5_preslice_0_1_Slice, cake5_top_0_0_Decorate,
cake6_assemble_0_0_Assemble, cake6_assemble_0_1_Store,
cake6_bottom_0_0_Decorate, cake6_preslice_0_0_Retrieve,
cake6_preslice_0_1_Slice, cake6_top_0_0_Decorate};
```

Another array is used to specify the input and output materials of these part operations.

```
array [SUBPART] of OPERATION: operation =
[Assemble__I_Slice20_Slice20__O_Cake20, Store__I_Cake20__O,
Decorate__I_Slice20__O_Slice20, Retrieve__I__O_Cake20,
Slice__I_Cake20__O_Slice20_Slice20, Decorate__I_Slice20__O_Slice20,
Assemble__I_Slice24_Slice24__O_Cake24, Store__I_Cake24__O,
Decorate__I_Slice24__O_Slice24, Retrieve__I__O_Cake24,
Slice__I_Cake24__O_Slice24_Slice24, Decorate__I_Slice24__O_Slice24,
Assemble__I_Slice28_Slice28__O_Cake28, Store__I_Cake28__O,
Decorate__I_Slice28__O_Slice28, Retrieve__I__O_Cake28,
Slice__I_Cake28__O_Slice28_Slice28, Decorate__I_Slice28__O_Slice28,
Assemble__I_Slice20_Slice20__O_Cake20, Store__I_Cake20__O,
Decorate__I_Slice20__O_Slice20, Retrieve__I__O_Cake20,
Slice__I_Cake20__O_Slice20_Slice20, Decorate__I_Slice20__O_Slice20,
Assemble__I_Slice24_Slice24__O_Cake24, Store__I_Cake24__O,
Decorate__I_Slice24__O_Slice24, Retrieve__I__O_Cake24,
Slice__I_Cake24__O_Slice24_Slice24, Decorate__I_Slice24__O_Slice24,
Assemble__I_Slice28_Slice28__O_Cake28, Store__I_Cake28__O,
Decorate__I_Slice28__O_Slice28, Retrieve__I__O_Cake28,
Slice__I_Cake28__O_Slice28_Slice28, Decorate__I_Slice28__O_Slice28];
```

The following enumeration specifies the types of part dependencies. They are defined for the subparts. For subparts of the same part, the new dependency type CONSECUTIVE is introduced.

```
% Dependencies between operations
enum DEPENDENCY = {NONE, STRICT, STORAGERETRIEVAL, LOOSE, CONSECUTIVE};
```

The following two-dimensional array captures the part dependencies as specified in the job model. To save space, only the dependencies of three subparts are shown.

```
array [SUBPART, SUBPART] of DEPENDENCY: dependency =
[|NONE, STRICT, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE,
NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE,
NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE|
NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE,
NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE,
NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE|
STRICT, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE,
NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE,
NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE, NONE|
```

What the constraint solver needs to do is find an optimum allocation. An allocation is defined as an assignment of subparts to equipment nodes. These are combined with the start and end times of this assignment.

```
% Allocation of operations to equipment nodes
array [SUBPART] of var NODE: alloc;

% Operation start and end times
array [SUBPART] of var 0..max: start;
array [SUBPART] of var 0..max: end;

% Makespan (maximum end time)
var 0..max: makespan;
```

The last part of the generated constraint program defines the constraints that constraint solver must satisfy. The first constraints specify the relation between the start and end time of subpart operations and the makespan.

```
% Start and end time constraints
constraint
  forall (t in SUBPART) (
    end[t] == start[t] + duration[alloc[t], operation[t]]
);

constraint
  forall (t in SUBPART) (
    start[t] >= 0
);

constraint
  forall (t in SUBPART) (
    makespan >= end[t]
);
```

Dependency constraints specify the timing constraints between subpart operations. For STRICT dependencies, this includes travel time; for other dependencies, a simple start-after-end constraint suffices.

```
% Dependency constraints
constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if dependency[t1, t2] == STRICT
    then
      end[t1] + travel[alloc[t1], alloc[t2]] == start[t2]
    endif
);

constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if dependency[t1, t2] == STORAGERETRIEVAL
    then
      end[t1] <= start[t2]
    endif
);
```

```
constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if dependency[t1, t2] == LOOSE
    then
      end[t1] <= start[t2]
    endif
);
```

The equipment constraints specify that an equipment node can perform only one operation at a time and that there are setup times between operations performed by the same equipment node. Note that these setup times may be 0.

```
% Equipment constraints
constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if allocation[t1] == allocation[t2]
    then
      start[t2] >= end[t1] + setup[alloc[t1], operation[t1], operation[t2]]
\/
      start[t1] >= end[t2] + setup[alloc[t2], operation[t2], operation[t1]]
    endif
);
```

Subpart allocation constraints specify how subpart operations must be allocated given their dependencies. Some operations must be performed by the same equipment node, whereas others must be performed by different equipment nodes. The last subpart allocation constraint specifies the consecutive execution of the subparts of one part. The sequence constraints are captured by the dependency constraints defined earlier.

```
% Subpart allocation constraints
constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if dependency[t1, t2] == STRICT
    then
      alloc[t1] != alloc[t2]
    endif
);

constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if dependency[t1, t2] == STORAGERETRIEVAL
    then
      alloc[t1] == alloc[t2]
    endif
);


constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if dependency[t1, t2] == CONSECUTIVE
    then
      alloc[t1] == alloc[t2]
    endif
);
```

```
constraint
  forall (t1, t2 in SUBPART where t1 != t2) (
    if dependency[t1, t2] == CONSECUTIVE
    then
      start[t2] >= end[t1] + setup[alloc[t1], operation[t1], operation[t2]]
    endif
);

constraint
  forall (t1, t2, t3 in SUBPART where t1 != t2 /\ t1 != t3 /\ t2 != t3) (
    if dependency[t1, t2] == CONSECUTIVE /\
      alloc[t1] == alloc[t3] /\ alloc[t2] == alloc[t3]
    then
      end[t3] <= min(start[t1], start[t2]) \/
        start[t3] >= max(end[t1], end[t2])
    endif
);
```

The last part of the constraint program defines the makespan minimisation objective.

```
% Makespan constraints
constraint makespan <= max;

solve minimize makespan;
```

When running the constraint solver for this instance with six cakes, one gets the schedule in Figure 4.1 with a makespan of 669 seconds. By decorating the largest cakes in the middle of the schedule, one obtains a schedule which is slightly shorter than the manually optimised schedule in Figure 3.5.

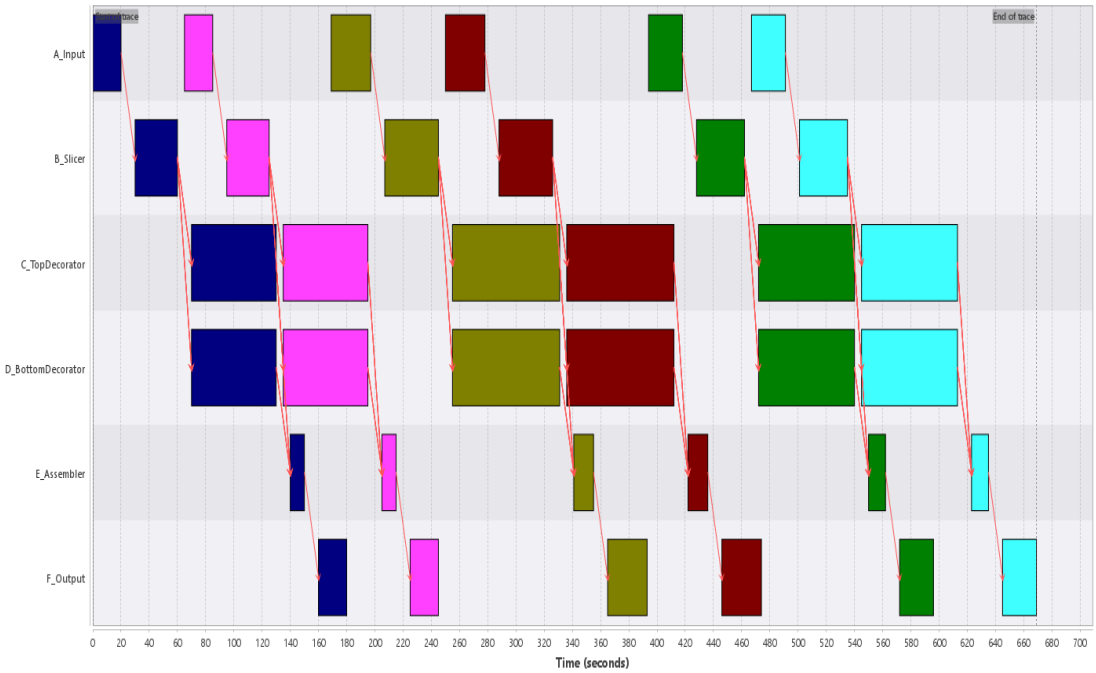

Figure 4.1: Automatically optimised cake decoration schedule

4.2 Allocation optimisation

The optimised cake decoration schedule was quickly found by running OR-Tools [14] from MiniZinc [13]. This is not surprising as the instance is small, and the constraint solver only must find the best sequence to decorate the cakes. By increasing the number of products to be manufactured or introducing a greater variety of products, the optimisation challenge becomes larger.

In this section, we assess the scalability of constraint solving. We do this by increasing the number of cakes to be decorated in the running example introduced in Chapter 2. For the scalability assessment, we run OR Tools CP-SAT 9.10.4067, a state-of-the-art constraint solver, from MiniZinc 2.9.3 on a PC with an Intel Core i7-1255U processor and 16 GB of RAM running Windows 11 version 24H2. As CP-SAT applies different strategies when using multiple worker threads [15], we will use the solver both using its default mode, i.e. with a single worker thread (see Table 4.1), and with 16 parallel worker threads (see Table 4.2).

The first column specifies the size of the instance being evaluated. The second column contains the makespan of the first solution found and the time needed to find this first solution. The third column contains the same values for the last, possibly optimum, solution found. The fourth columns contains the total analysis time, which was maximised at 15 minutes. If the fourth column contains a value, the last solution found is guaranteed to be the optimum. If a fourth column is empty, the last found solution may be suboptimal. If the second and third column are empty, no solution has been found within 15 minutes.

Table 4.1: Constraint solving scalability assessment (1 worker thread)

Number of cakes	First solution	Last solution	Total analysis time
1	180 s / 0.1 s	180 s / 0.1 s	0.2 s
2	314 s / 0.1 s	314 s / 0.1 s	0.2 s
3	456 s / 0.2 s	450 s / 0.2 s	0.3 s
4	564 s / 0.5 s	515 s / 19 s	-
5	698 s / 0.4 s	594 s / 628 s	-
6	840 s / 0.9 s	718 s / 395 s	-
7	948 s / 1.3 s	844 s / 493 s	-
8	978 s / 785 s	978 s / 785 s	-
9	- / -	- / -	-
10	- / -	- / -	-

Table 4.2: Constraint solving scalability assessment (16 worker threads)

Number of cakes	First solution	Last solution	Total analysis time
1	180 s / 0.1 s	180 s / 0.1 s	0.2 s
2	348 s / 0.2 s	314 s / 0.2 s	0.2 s
3	456 s / 0.3 s	450 s / 0.3 s	0.4 s
4	576 s / 0.3 s	515 s / 0.3 s	0.4 s
5	908 s / 0.4 s	588 s / 0.4 s	0.5 s

Number of cakes	First solution	Last solution	Total analysis time
6	936 s / 0.6 s	669 s / 0.7 s	0.8 s
7	945 s / 1.4 s	734 s / 1.5 s	2.1 s
8	1,000 s / 1.3 s	807 s / 1.7 s	6.0 s
9	997 s / 1.7 s	888 s / 2.1 s	53 s
10	998 s / 2.9 s	953 s / 3.8 s	621 s
11	- / -	- / -	-
12	- / -	- / -	-
13	- / -	- / -	-
14	- / -	- / -	-
15	- / -	- / -	-

The results in Table 4.1 and Table 4.2 clearly show the benefits of using multiple search strategies. With 16 worker threads, CP-SAT finds more and better solutions and in less time than with a single worker thread, even with compensating for the extra workers. Unfortunately, the tables also show that CP-SAT is not very scalable: no solution can be found for an instance with more than ten cakes.

The research of Boonstoppel [16] confirms the lack of scalability of Constraint Programming (CP), especially if there are sequence-dependent setup times. He has compared OR-Tools CP-SAT [14] and IBM ILOG CP Optimizer [17]. Both solvers suffer from (long) setup times as well as from long task execution times.

Boonstoppel [16] has also looked at a formalism that does not suffer from large execution and setup times: Multi-valued Decision Diagrams (MDDs) [18]. MDDs are directed acyclic graphs that describe the solution space of optimisation problems. The nodes in this graph correspond to a partial solution, i.e. an assignment of values to some decision variables. The outgoing arcs of a node correspond to the possible values of one (undecided) decision variable. An MDD has a source and a sink node. In the source node, none of the decision variables have received a value; in the sink node, all decision variables have received a value. Any path from the source to the sink corresponds to a solution, whose cost can be computed from the arcs on the path.

MDDs can be large, but there are strategies to limit their size and speed up the search, which is not possible for CP. For instance, the width of MDDs can be maximised; if an MDD layer exceeds the maximum width, the most promising partial solutions are selected and the other partial solutions are merged [18]. Boonstoppel [16] shows that MDDs sometimes outperform CP. He has shown that a solution can be found very quickly if the maximum MDD width is kept small. Unfortunately, this initial solution is not always near optimal. This could be overcome by defining better heuristics for selecting and merging partial solutions. MDD solutions can also be used as a starting point for further searching. This could be continuation of MDD exploration from the discarded partial solutions, but an MDD solution could also be used to warm start CP. In Boonstoppel's experiments, the latter did not allow CP to find better solutions or to find solutions faster.

5 Conclusion

In this report, we have analysed the optimisation of high-mix low-volume (HMLV) production systems, i.e. production systems which produce a high variety of products in small series, in the context of the AIMS 2024 project.

This report addresses three research questions:

- RQ1. What is an appropriate way to describe a HMLV production line's equipment, its workload, and the allocation of this workload onto the line's equipment?
- RQ2. How can the latency of a specific allocation of a HMLV production line's workload onto its equipment be computed in an fast and accurate manner?
- RQ3. How can a (near-)optimum allocation of a HMLV production line's workload into its equipment be found (in reasonable time)?

Chapter 2 of this report provides an answer to RQ1; it describes domain languages to describe the equipment of a (HMLV) production system, a workload and the allocation of this workload to the equipment. Compared to the earlier AIMS 2023 results [1], the updated domain languages allow more systems to be specified, and the specifications are shorter. In particular, the new domain languages allows specification of assembly and disassembly operations.

RQ2 is addressed by Chapter 3, which presents a transformation of the domain languages introduced in Chapter 2 to constraint graphs. Constraint graphs provide a fast analysis of fully specified allocations. Compared to the earlier AIMS 2023 results [1], the transformation to constraint graphs generates smaller graphs, which may be beneficial for the analysis of large instances. Experiments show that allocations of a few hundred products can be analysed within a few seconds, but that the analysis times increases in a non-linear manner. This non-linear scaling could be circumvented by using an alternative algorithm for the underlying longest path computation.

Chapter 4 is concerned with RQ3. It presents a transformation from the domain languages introduced in Chapter 2 to a constraint programming specification in MiniZinc notation. This specification can be solved to optimality for workloads involving a few products. For slightly larger instances, one runs into the lack of scalability of constraint programming: either a suboptimal solution is found, or no solution at all. So the studied transformation to constraint programs does not provide a sufficient answer to RQ3. To find near-optimum allocations for realistic workloads in reasonable time, a heuristic approach is needed. A promising formalism for such an approach would be Multi-valued Decision Diagrams, as they have means to quickly find a solution and their search of the optimisation space can be tailored using heuristics for combining and selecting partial solutions.

References

[1] J. Verriet, "Compositional Performance Prediction for Tightly Coupled Manufacturing Systems," TNO, Eindhoven, 2023.

[2] M. J. Vassalo, E. P. Y. Dekker, F. A. Bosneag, M. J. v. Bokhoven, P. Kostadinov, A. G. Anton, D. Manolev, G. R. Ratolla, A. E. Mangos, Y. Lin and E. W. P. J. Kuppens, "PLOT Software User Manual," Eindhoven University of Technology, Eindhoven, 2024.

[3] Eclipse Foundation, "Xtext," 2025. [Online]. Available: https://eclipse.dev/Xtext/.

[4] Eclipse Foundation, "Xtend," 2025. [Online]. Available: https://eclipse.dev/Xtext/xtend/.

[5] S. E. Elmaghraby and J. Kamburowski, "The Analysis of Activity Networks Under Generalized Precedence Relations (GPRs)," *Management Science,* vol. 38, no. 9, pp. 1245-1263, 1992.

[6] D. C. Ku and G. De Micheli, "Relative Scheduling under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 11, no. 6, pp. 696-718, 1992.

[7] R. M. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, New York, Plenum Press, 1972, pp. 85-103.

[8] R. E. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics,* vol. 16, no. 1, pp. 87-90, 1958.

[9] Eclipse Foundation, "Eclipse TRACE4CPS," 2025. [Online]. Available: https://eclipse.dev/trace4cps/.

[10] A. Bernstein, D. Nanongkai and C. Wulff-Nilsen, "Negative-Weight Single-Source Shortest Paths in Near-Linear Time," *Communications of the ACM,* vol. 68, no. 2, pp. 87-94, 2025.

[11] K. Bringmann, A. Cassis and N. Fischer, "Negative-Weight Single-Source Shortest Paths in Near-Linear Time: Now Faster!," in *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, Santa Cruz, CA, 2023.

[12] Google OR-Tools, "Constraint Optimization," 2025. [Online]. Available: https://developers.google.com/optimization/cp.

[13] MiniZinc, "MiniZinc," 2025. [Online]. Available: https://www.minizinc.org/.

[14] Google OR-Tools, "CP-SAT Solver," 2025. [Online]. Available: https://developers.google.com/optimization/cp/cp_solver.

[15] D. Krupke, "The CP-SAT Primer: Using and Understanding Google OR-Tools' CP-SAT Solver," 2025. [Online]. Available: https://d-krupke.github.io/cpsat-primer/.

[16] S. Boonstoppel, "Solving the Flexible Job Shop Scheduling Problem with Alternative Process Plans," Utrecht University, Utrecht, 2025.

[17] IBM, "IBM ILOG CP Optimizer," 2024. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer.

[18] W.-J. van Hoeve, "An Introduction to Decision Diagrams for Optimization," *Tutorials in Operations Research,* vol. 2024, pp. 117-145, 2024.

[19] X. Gillard, P. Schaus and V. Coppé, "DDO a generic and efficient framework for MDD-based optimization.," 2025. [Online]. Available: https://github.com/xgillard/ddo.

Appendix A
Domain model

The domain model described in this report consists of four languages: a material language, an equipment language, a job language and an allocation language. The syntaxes of these languages are visualised in Figure 5.1, Figure 5.2, Figure 5.3, and Figure 5.4, respectively.

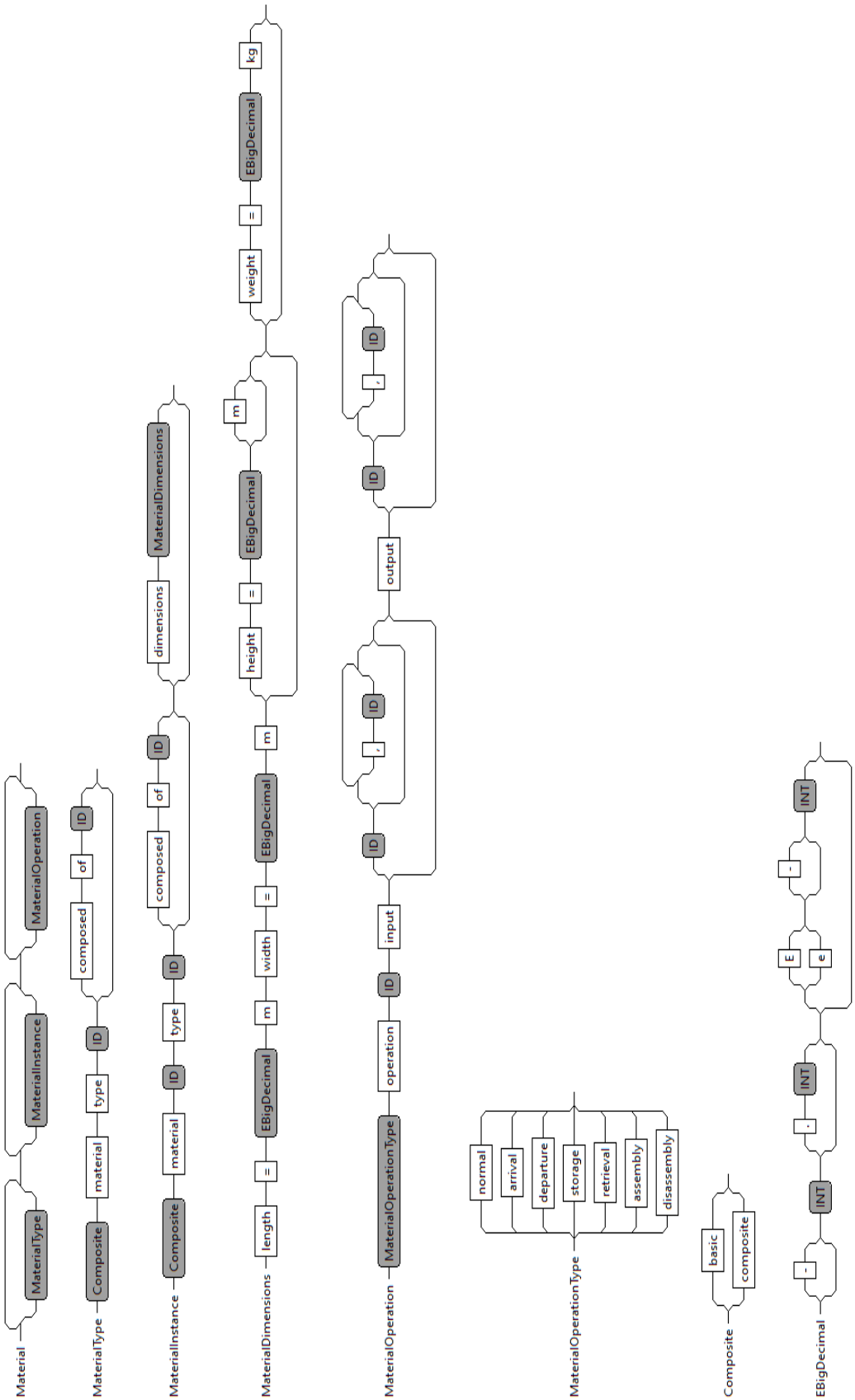

Figure 5.1: Material language syntax

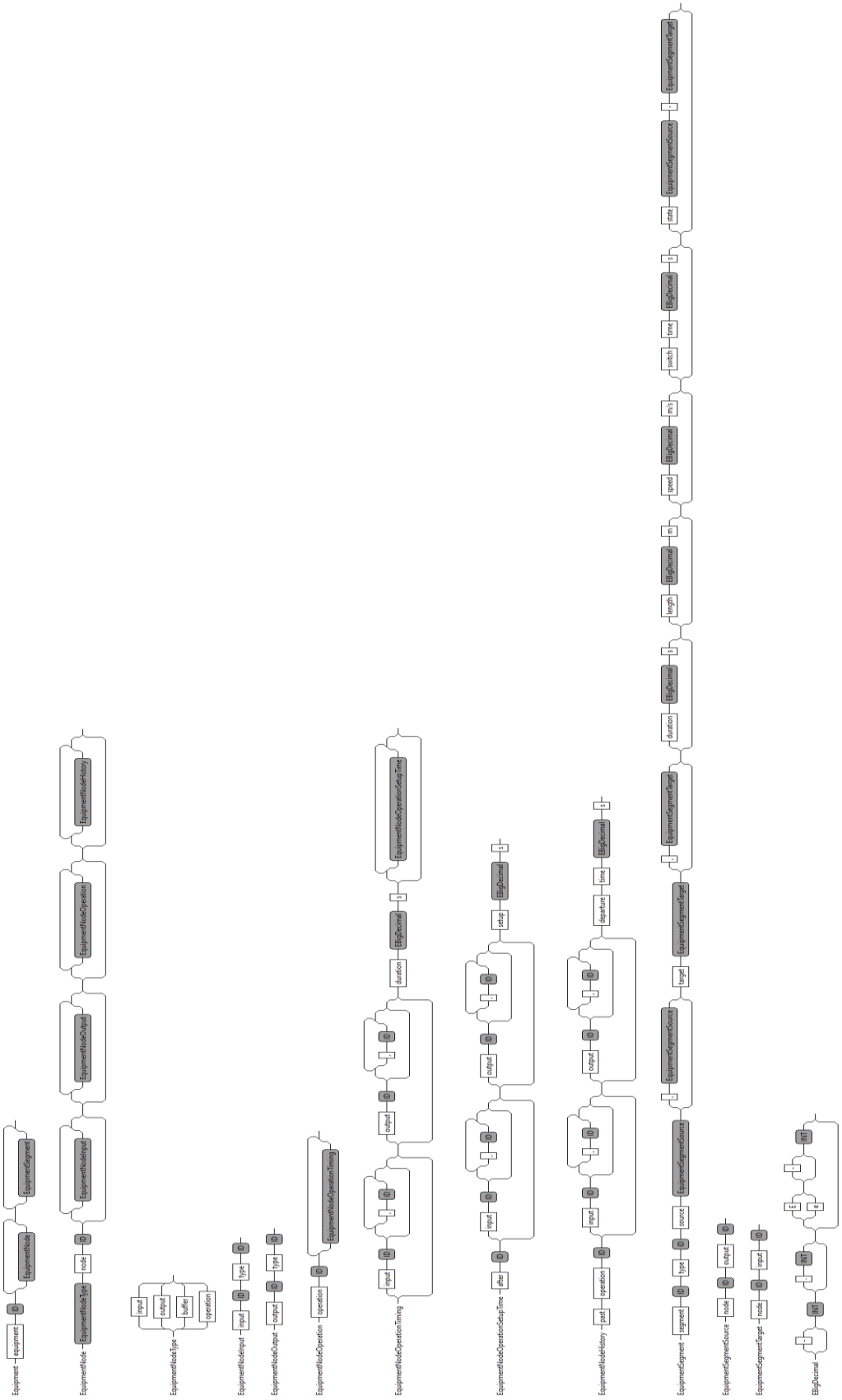

Figure 5.2: Equipment language syntax

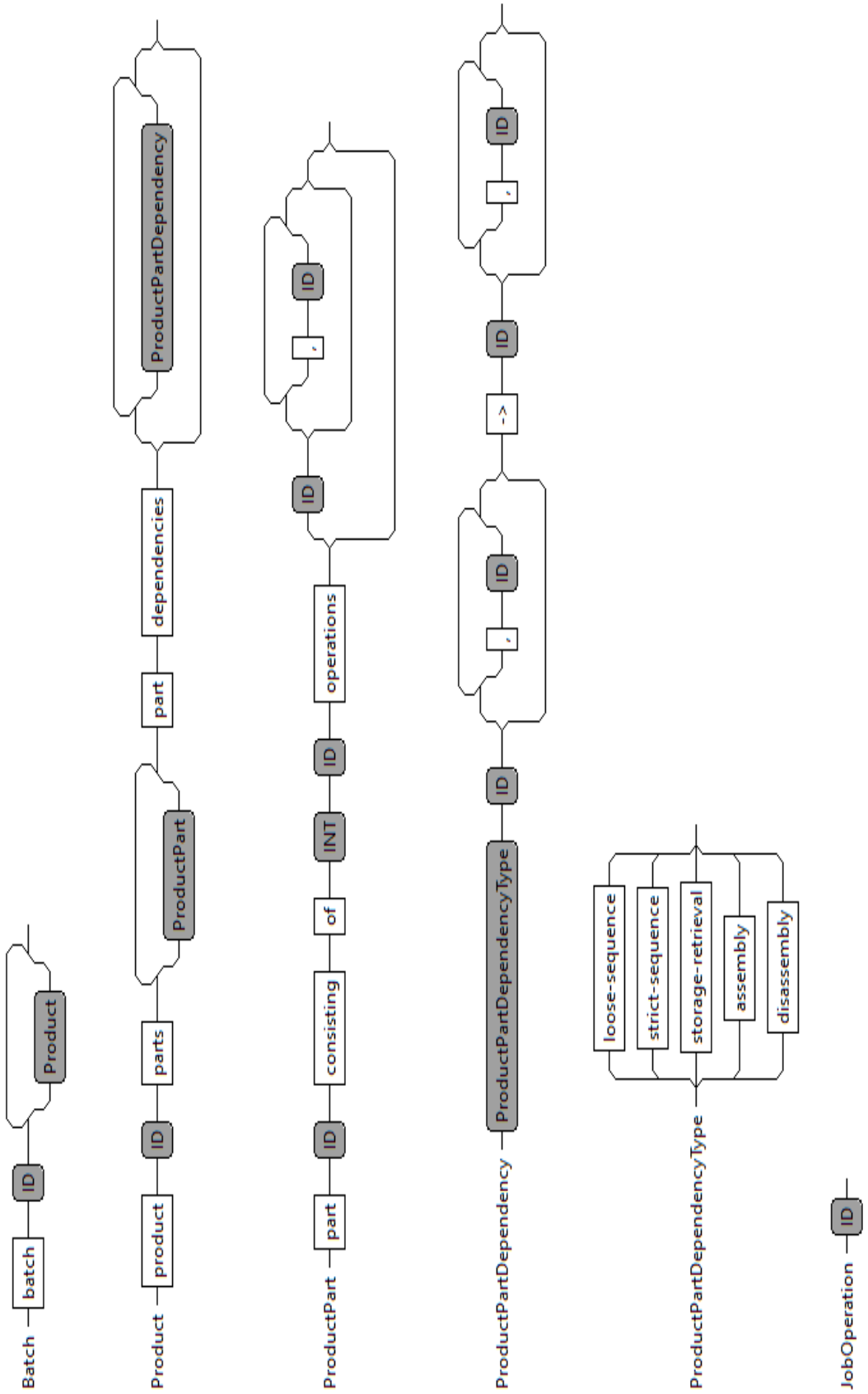

Figure 5.3: Job language syntax

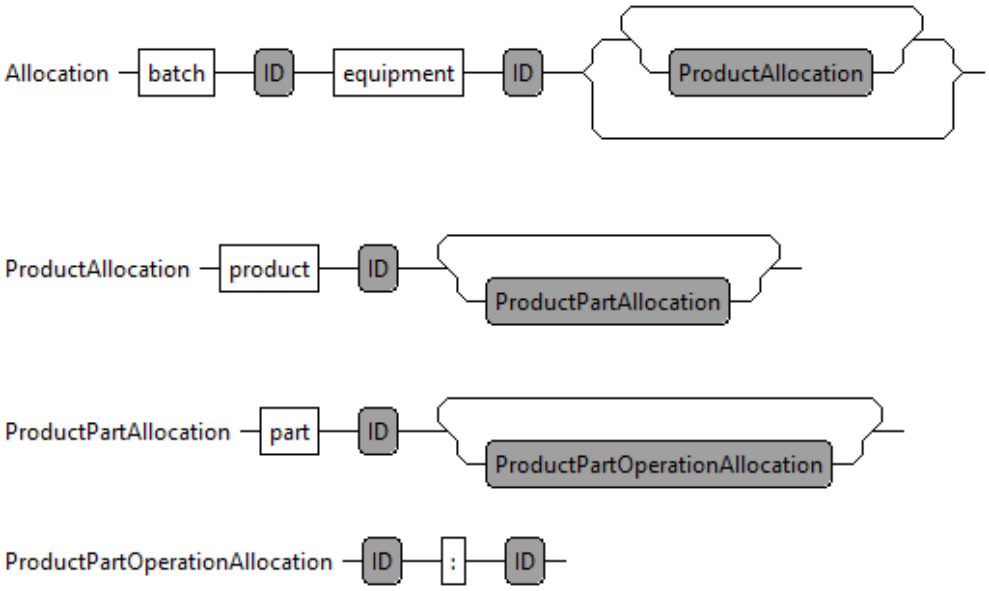

Figure 5.4: Allocation language syntax

ICT, Strategy & Policy

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl