D3: A Turnaround Model and Prototype based on Decoupling

CARE INO III: The Co-ordinated Airport through Extreme Decoupling

Pim van Leeuwen (NLR)
Lian Ien Oei (TUD)
Cees Witteveen (TUD)
Henk Hesselink (NLR)
CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING
Summary

This document offers a description of the turnaround model and corresponding prototype as it has been developed in the context of the Co-ordinated Airport through Extreme Decoupling (CAED) project. First, the turnaround model introduced in deliverable D2 of the project is described in more detail, listing its goal, objectives, user group and intended operational application. Furthermore, the scope of the model is defined and requirements are listed for its implementation. Second, a prototype is described that has been developed to demonstrate the feasibility of the decoupling approach. A description is given of the prototype’s design, its input and output specifications, its operability, and its implementation. Moreover, an elementary test is described that has been conducted to evaluate whether the prototype meets its requirements. Third, a description is given – starting from two typical delay scenarios – of how tactical replanning of ground handling processes can benefit from this result. Finally, some conclusions are drawn about the advantages and disadvantages of the proposed model and its prototype implementation.
# Contents

1 Introduction 6
   1.1 CAED Project Context 6
   1.2 Deliverable D3 Document Structure 6

2 Objectives and Users 8
   2.1 Introduction 8
   2.2 Operational Application of the Model 10
   2.3 Main Objective 11
   2.4 User Group 11

3 Scope and Requirements 13
   3.1 Scope 13
   3.2 High-level Requirements 14
      3.2.1 Requirements for the Pre-tactical Phase 14
      3.2.2 Requirements for the Tactical Phase 15

4 Prototype Design Description 16
   4.1 Introduction 16
   4.2 The input 17
   4.3 Pre-processing the input 20
   4.4 Internal processes of the prototype 20
   4.5 The output 20

5 Prototype Implementation and User Manual 22
   5.1 Implementation 22
   5.2 User Manual 25

6 Prototype Testing and Graphical Extension 27
   6.1 Prototype Input 27
      6.1.1 Test Scenario 27
      6.1.2 Airline Service Times 29
      6.1.3 Aircraft Manufacturer Service Times 29
      6.1.4 Stand Matrix 30
      6.1.5 Task-actor assignments 30
   6.2 Prototype Test 31
CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Performance</td>
<td>33</td>
</tr>
<tr>
<td>6.4</td>
<td>Graphical Input and Output Add-on</td>
<td>33</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Macro ProduceStandPlanGraph.xls</td>
<td>34</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Macro ProduceStandPlanAndDelayGraph.xls</td>
<td>35</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Macro SelectAndFilterFlight.xls</td>
<td>36</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Macro SelectAndFilterFlight.xls</td>
<td>38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Using the Prototype’s Output in the Tactical Phase</td>
<td>40</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>40</td>
</tr>
<tr>
<td>7.2</td>
<td>Delayed service provider</td>
<td>43</td>
</tr>
<tr>
<td>7.3</td>
<td>Delayed arrival</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>References</td>
<td>51</td>
</tr>
<tr>
<td>9.1</td>
<td>Document References</td>
<td>51</td>
</tr>
<tr>
<td>9.2</td>
<td>Online references</td>
<td>52</td>
</tr>
</tbody>
</table>

**Appendix A**  
The Decoupled Fuelling Service  
53

**Appendix B**  
Original Example Planning for the Fuelling Service  
54

**Appendix C**  
Prototype Implementation Documentation  
55

C.1 Decoupler Class  
55
C.2 Edgematrix Class  
56
C.3 MainData Class  
59
C.4 Output Class  
60
C.5 ProcessInput Class  
62
C.6 STP Class  
64

(68 pages in total)
1 Introduction

This document describes the turnaround model and implemented prototype developed within the Co-ordinated Airport through Extreme Decoupling (CAED) project. The turnaround model is developed to support the integration of local ground handling processes into an encompassing, total turnaround plan. Central to this approach is the assumption that local parties are in the best position to plan their resources and activities. Based on this assumption, a methodology called ‘extreme decoupling’ has been investigated to ensure that – given an initial seasonal stand plan – local ground handling plans can be established and merged again into a conflict-free stand and services plan.

1.1 CAED Project Context

The CAED project is executed in the context of Collaborative Decision Making (CDM) and Total Airport Management (TAM). CDM has been identified as an important enabler of capacity and efficiency in air transport [CDM Handbook, 2003]. TAM can be seen as the following step, encapsulating CDM and bringing a planning and decision support component to it [TAM, 2001].

In the project, two deliverables have been previously produced. In [D1, 2007], a literature study has been conducted, giving an overview of research on Simple Temporal Problems (STPs), the Temporal Decoupling Problem, and methods for solving STPs and the Temporal Decoupling Problem. This report described the theoretical background for modelling the ground handling domain.

Deliverable [D2, 2007] focussed on the operational background of the project. In this document, the turnaround process and underlying ground handling processes have been described. Additionally, an outline for a general turnaround model has been presented, as well as a detailed example to illustrate how the model can be applied in practice.

The current document constitutes the last deliverable of the project in Phase 1 (Year 1). In this document D3, the turnaround model first introduced in [D2, 2007] is elaborated in further detail. Also, a prototype implementation of the model is described that has been developed to demonstrate and test the feasibility of the ideas exposed in [D1, 2007] and [D2, 2007].

1.2 Deliverable D3 Document Structure

The outline of this document is as follows. In chapter 2, the main objective and user group of the turnaround model are described. In chapter 3, the scope and requirements following from this objective are detailed. In chapter 4, a high-level design description including input and
output specifications for the prototype is given. Chapter 5 details the prototype’s implementation and offers a short user manual. In chapter 6, a scenario is described that has been used to test the prototype; test results are provided to demonstrate that the prototype meets its requirements. Chapter 7 then discusses how the implemented model should be operated in the tactical phase. Finally, in chapter 8, some conclusions are drawn.
2 Objectives and Users

In this chapter, the objectives and users for the turnaround model are identified. Section 2.1 starts with an introduction on the ground handling domain. Section 2.2 explains in more detail how the turnaround model will be applied and what its main advantages are. Section 2.3 describes the main objective of the model, and of the prototype that implements this model. In Section 2.4 the user group of the model and corresponding prototype will be identified.

2.1 Introduction

In 2004, EUROCONTROL predicted flight traffic in Europe to double by 2020 [PR, 2004]. Given this increase in air traffic, airports will become a major bottleneck in the air transport system. Expansion of airports, however, is expensive and often impossible due to safety and environmental constraints. Therefore, airport authorities are seeking methods to increase airport capacity by making more efficient use of existing resources.

Currently, one of the most important factors preventing a more efficient use of resources is the occurrence of disruptions. At any medium to large size airport dozens of disturbances occur daily. Without re-planning, these disturbances may lead to grave consequences: flights delaying other flights (reactionary delays), provoking gate changes, affecting the in- or outbound capacity of the airport, or even hampering flights at neighbouring airports. Therefore, re-planning has become a daily necessity.

Given the complex and distributed nature of the airport domain, however, re-planning is not an easy task. It typically involves a large group of parties, each having their own (commercial) interests, resources, and planning constraints. Between these different parties, a large number of dependencies exist. For instance, between ground handlers, many temporal constraints apply (e.g., the fuelling company may not start fuelling unless all passengers have deboarded). Any disruption typically affects a large number of activities, since any one activity forms part of a network of related activities (related temporally or by resources used). If a change occurs in one planning domain, it may have repercussions for all.

In particular, these complications have become visible in European Union projects such as LEONARDO [LEO, 2004]. Although fully acknowledging the distributed nature of the domain (by implementing a multi-agent system to support all actors), this project ran into difficulties when trying to model real-time scenarios. The majority of conflicts dealt with necessarily involved all parties. To solve any type of realistic disruption to the original plan, an enormous amount of communication was required to co-ordinate local planning functions. This led to a
large co-ordination overhead when trying to solve the distributed planning problem. As a result, a general tool assisting all planners in solving real-time disturbances at airports turned out not to be feasible.

Airport planning is typically subdivided into a number of domains: arrival management, departure management, stand allocation management, taxi planning. In all of these areas, extensive research has been conducted to improve planning and assist planners by means of decision support tools (e.g., [LEO, 2004], [Buzing et al., 2004], [Jonker et al., 2005], [Van Leeuwen et al., 2003]. Ground handling, denoting all processes that take place when an aircraft is at the stand\footnote{The term ‘stand’ will be used for both gates and remote stands throughout this document.} between flights, is a notable exception. Up until now, not much research has been conducted in this area.

This is surprising, since ground handling is recognized as a common and important source of delays in the air transport system. According to research conducted at London Gatwick Airport, ground handling services are the second largest contributor to flight delays, right after air traffic control (ATC) related delays. In this research, ground handling services proved to be responsible for 25 percent of all delays at London Gatwick [Wu et al. 1, 2000; Wu et al. 2, 2004]. Ground handling delays typically lead to delays in other airport processes – not only for the delayed aircraft itself, but also for other aircraft, whether inbound, outbound or docked at other stands (knock-on delays). Thus, the performance of aircraft turnaround operations has a strong impact on the punctuality of the totality of airport operations. For this reason, NLR has recently – in parallel with the CAED project – intensified its research efforts on the turnaround process as a key chain in airport operations (e.g., [SSM, 2007]).

In [D2, 2007] of CAED, a turnaround model has been introduced to support airport authorities in the establishment of a robust pre-tactical ground handling plan for servicing scheduled aircraft. To this end, a method called ‘extreme decoupling’ has been developed to break up an initial stand plan into several subplans, each of which corresponds to the actions and constraints one single agent involved in the turnaround process has to complete. These subplans can be solved separately and merged together into a conflict-free pre-tactical stand and services plan. Thus, the decoupled domain representations allow parties to solve many tactical disruptions locally, making plan co-ordination and negotiation between parties largely superfluous. Given the large number of plan disruptions occurring daily at airports, and the increase expected in air traffic, such a planning tool will be a valuable asset.
2.2 Operational Application of the Model

The turnaround model introduced in [D2, 2007] aims to assist planners in the establishment of a pre-tactical ground handling plan for a given day of operations. It produces a stand and services plan\(^2\), based on the flights an airline intends to perform, the airport's stand availability, and the various temporal and resource constraints for the ground services. Currently, such a plan is also produced in practice. After all, during the strategic and pre-tactical planning, a rudimentary seasonal stand allocation plan is refined until at the day of operation a fully functional stand and services plan is ready for use.

A plan produced by the turnaround model differs from such a plan, produced at airports today, in the aspect that in the new plan individual agents are in control of their own section of the total plan. A method called 'extreme decoupling' has been developed to break up an initial stand plan into several subplans, which can be solved separately and merged again into a conflict-free pre-tactical plan. Each of these subplans corresponds to the actions and constraints one single agent involved in the turnaround process has to complete. Extreme decoupling ensures that whatever plan execution scheme is applied by an individual agent, as long as it satisfies the decoupled plan, the feasibility of the total plan execution is guaranteed. This implies that, whenever a disruption appears at the day of operation, if such a disruption can be resolved by adapting one or more local plans, decoupling will guarantee us that the total plan is still feasible.

It should be noted that the turnaround model does not yield a solution that is necessarily better in terms of efficiency, costs, or throughput than the solutions offered by current modes of operation. In fact, since the decoupling algorithm may cut away a small portion of the search space when partitioning the domain (see [D1, 2007], chapter 5), it may - at least in theory – exclude solutions (plans) that actually do satisfy the originally given constraints.

The model's main benefit comes from the support it offers during the tactical or operational phase. In this phase, typically a number of disruptions will occur that threaten the pre-tactical stand and services plan. These disruptions need to be dealt with by re-planning part of the original plan. Of course, one does not want to make a new planning for all turnaround activities for that entire day. It is important to keep these re-planning activities as local as possible - affecting an absolute minimum of parties, aircraft, personnel, and resources. As mentioned in the introduction, current re-planning methods involve all parties, which leads to large co-ordination overhead. In the planning created by extreme decoupling, however, the original planning representation has already been distributed into independently constrained local plans.

\(^2\) The term ‘stand and services plan’ is used throughout the document to indicate that a stand allocation plan is complemented by a services plan. This services plan lists for each ground handling service a time interval within which the service can be executed without being in conflict with the stand allocation plan and all other constraints in the domain.
This enables parties to solve many tactical disruptions locally, thus drastically reducing the co-ordination overhead and saving time and money.

The decoupling methodology therefore lends itself naturally to solving disruptions locally. Every actor can first try to solve a delay it suffers locally, for its planning problem is represented in isolation – without any further dependencies to other parties. Thus, any party can quickly see whether it can solve a disruption (either caused by itself, or by others) itself, by updating some of its constraints and re-planning only the affected part of its own planning. Only if this re-planning does not yield any solutions will it be necessary to bother others.

### 2.3 Main Objective

In [D2, 2007], page 26-27, two goals were defined for Phase I of the project. The first goal reads as follows:

*The turnaround model should minimize the required co-ordination in planning between actors as much as possible – whilst preventing that their plans are in conflict with one another.*

As argued above, this minimization of the required co-ordination will typically be beneficiary in the tactical phase, when re-planning is required to solve disruptions and little time is available to come up with solutions. In order to measure whether the model accomplishes this goal, a comparison needs to be made with current modes of solving tactical disruptions. The turnaround model should therefore demonstrate that the decoupled stand allocation plan it produces offers important tactical re-planning advantages over the nominal stand allocation plan.

The second goal states that the decoupling should not lead to plans conflicting with the original stand allocation, or other parties’ plans. This goal will be reached if the decoupling algorithm is properly implemented and intelligently applied to the airport domain – it follows directly from the Mergeable Solutions Property of Hunsberger’s algorithm (see [D1, 2007], chapter 5). In other words: if the prototype is properly implementing Hunsberger’s algorithm, this goal will be automatically satisfied. Therefore, the second goal will be our main objective.

### 2.4 User Group

An important question can be raised at this point. Who will use the prototype, when, and how? To answer this question, the prototype’s role in the subsequent planning phases should be clarified.
In the strategic phase, it is the responsibility of the airport authorities to establish a seasonal stand allocation plan, allocating the expected flights for a given day at the airport to its available stands [D2, p. 17]. This seasonal stand allocation plan, called the stand plan from now on, is established about half a year before the actual day of operation (with ‘seasonal’ referring to winter or summer).

Throughout the strategic phase, running up to 7 days before operation, this stand plan may be refined depending on newly available information: infrastructural constraints, operational practices, the airport’s declared capacity, flight expected schedules, airlines’ stand planning preferences, ground handler capabilities, etc. From 7 days until 2 hours before operation, the pre-tactical phase sets in. In this phase, it will be the airline which, in conjunction with the service providers, will develop a stand and services plan.

It is in this pre-tactical phase that the prototype will be used. The stand plan will be its input: a list of flights allocated to stands between in- and off-block times. Output of the planning process will be a stand and services plan established through the prototype’s decoupling algorithm. In other words, this output still contains the stand plan, but adds all services to it in a manner that gives each service provider control over its part of the planning.

In the tactical phase, finally, the stand and services plan produced by the prototype will be put to good use. In case of disruptions, the service provider(s) affected by (or causing) the disruption will re-plan its part of the overall turnaround plan using the prototype’s plan. Thus, only the affected services will have be updated (e.g., the boarding company being delayed because of a no-show passenger) and then re-planned, leaving all other services intact.

---

3 As explained in section 2.2, this ‘control’ is guaranteed because the services in the stand and services plan are not fixed yet to specific timepoints, but instead are assigned time intervals. Thus, the service provider is free to further plan the services as (s)he sees fit. If the services are planned within the assigned intervals, the solution is guaranteed to be conflict-free.
3 Scope and Requirements

In this chapter, the scope of the turnaround model is defined. Furthermore, the high-level requirements for its prototype implementation are specified.

3.1 Scope

In Phase I of the project, the scope of the turnaround model is necessarily limited. This scope can be defined with respect to the model input, the model itself, and the model output. The quantitative boundaries (e.g., at least 3 aircraft types) below are estimates. If applicable, a related model assumption laid down in [D2, section 3.2] is listed. The scope of the input can be defined as follows:

- The number of aircraft types taken into account should be at least 3 (related to assumption 1C).
- The number of stands taken into account should be at least 20.
- The number of aircraft to be planned should be at least 10.
- Constraints of the following three input sources are taken into account:
  - Aircraft type specific minimal service times (defined by the aircraft manufacturer).
  - A simplified stand allocation plan (laying down which aircraft have been planned when and at which stand).
  - Airline specific maximal service times (specified by the airline).
- Service Provider specific constraints (in Phase I, only a matrix with minimum travel times between stands for each service provider) are taken into account.

Moreover, the scope of the model itself can be formulated as follows:

- At least five services (the main services) are taken into account: baggage, cleaning, catering, fuelling, and boarding (assumption 1A).
- At least five service providers are taken into account: the baggage loading operator, the water, cleaning and lavatory servicing operator, the fuelling company, the caterer, and the boarding company (assumption 1B).
- The model focuses on the planning of ground handling activities of one airline only.
- Certain services, split up into multiple parts depending on the aircraft type, will be mapped onto one service with a single defined start time and duration for reasons of simplicity.

---

4 Note that, contrary to assumption 1D [D2, p. 23], also the airline specific service times are taken into account in the model.

5 For instance, for the B777-200, the baggage services are split up into the unloading and loading of the FWD and AFT compartments. For other aircraft types, other divisions may apply.
• Service providers are assumed to have the same number of resources available during the entire time period covered by the prototype (e.g., an entire day of operations).

• The model will yield a set of at least five local plans that together form the new stand and services plan (assumption 2A).

3.2 High-level Requirements

Having defined the scope of the turnaround model, it is now time to list the most important requirements to be met by its implementation. Starting point for these high-level requirements is the objective formulated in section 2.3. This objective stated that “The turnaround model should minimize the required co-ordination in planning between actors as much as possible – whilst preventing that their plans are in conflict with one another.” Given this objective, a number of requirements can be formulated. These requirements can be split up based on the planning phase they relate to.

3.2.1 Requirements for the Pre-tactical Phase

In the pre-tactical planning phase, the prototype will read the stand allocation plan and service time information, and produce a stand and services plan based on its decoupling algorithm.

REQ. 1: The prototype shall produce a conflict-free, decoupled stand and services plan based on the input it receives (specified in detail in section 4).

REQ. 2: If any of the input files cannot be processed or is in the wrong format, a proper error message shall be raised.

REQ. 3: The prototype shall produce the decoupled stand and services plan within a reasonable amount of time:

• For a problem instances of max. 50 flights, the prototype shall produce a solution within 1 minute, if a solution can be found.
• For larger problem instances (up to 1000 flights), the prototype shall produce a solution within 10 minutes.

REQ. 4: If after the maximum search time no solution has been found, the prototype shall stop its search and report that no solution has been found.

REQ. 5: If the output cannot be written, a proper error message shall be raised.

These numbers of resources are specified in the task-actor-resource assignment file – see chapter 4 and 6 below.
3.2.2 Requirements for the Tactical Phase

In the tactical phase, the prototype’s output will allow its users to respond much quicker to last-minute changes. This time gain will allow the users to cope with disruptions that could not be dealt with in case of the nominal stand allocation plan. In this phase, the prototype will assist users in re-planning their part of the stand and services plan in order to cope with disruptions.

REQ. 6: The prototype’s output shall demonstrate that, for a given scenario, a smaller number of service providers needs to be involved in re-planning when using the decoupled stand and services plan. This smaller number of parties involved will lead to a repaired plan faster, since less time is required to co-ordinate between the different interests and solution propositions of the involved parties.

REQ. 7: The prototype’s output shall demonstrate that, for a given scenario, the service providers that are required for accommodating the disruption need less time to re-plan their local plan. Thus, not only are fewer parties involved in re-planning (Req. 6), but also less time is required for each party to do its re-planning. As a consequence, disruptions can be accommodated much quicker when using the decoupled stand and services plan – as opposed to the nominal plan.
4 Prototype Design Description

In this section, a high-level design description is given of the prototype implementing the turnaround model. The prototype will be operated in the pre-tactical phase, producing a decoupled stand and services plan (satisfying requirements 1-5). In chapter 7, the usefulness of this decoupled plan for the tactical phase will be demonstrated (requirements 6-7).

Below, a general overview of the prototype’s design is given, as well as the input files it will require. Furthermore, the pre-processing and internal processing activities of the prototype are sketched. Finally, the prototype’s output is listed.

4.1 Introduction

In figure 1, a general overview of the prototype and its input and output streams is presented.

![Prototype general overview](image)

**Figure 1: Prototype general overview**

In Figure 1, the following steps described in [D2, 2007] have been implemented:

1. Read (1) the aircraft manufacturer’s minimum service times, (2) the list of aircraft to be serviced in the stand allocation plan, (3) the airline’s service norm times, (4) stand matrix defining travel times between stands, and (5) the actor-task-resource assignments specifying the tasks each actor is responsible for (e.g. boarding and de-boarding for the
boarding company), and the maximum number of resources available. After reading these input files, the data extracted is stored in adequate data structures (pre-processing).

2. Construct a large Simple Temporal Network (STN) based on the stored data structures of inputs (1), (2) and (3).

3. Add local constraints based on the (5) task-actor-resources assignment\(^7\) to the complete STN.

4. Use decoupling to split up the complete STN into local STNs for each service provider. Next, all local constraints are removed to allow a maximum of flexibility for each service provider\(^8\).

5. Output the stand and services plan, listing for each service of each service provider a plan matching the original stand plan, to the output directory specified by the user.

Note that the further planning of services within their guaranteed intervals is left to the service providers (SP), allowing them a maximum of flexibility. The prototype outputs for each SP a local STN that matches both the global constraints (stemming from the stand allocation plan) and local constraints (such as the number of resources of that SP, limiting the number of tasks that can be executed simultaneously). In this local STN, both a maximum and minimum interval is specified within which each task of the SP can be planned – as is further explained in section 6.

Below, the prototype’s design is described in a step-by-step fashion. These steps will guide the reader from its input through a sequence of pre-processes and internal processes to the output.

4.2 The input

Figure 2 yields the input files for the prototype:

---

\(^7\) Based on the tasks and resources per actor, the prototype adds extra local constraints to the STN. For instance, when the fuelling company has only 2 vehicles available for the fuelling task, precedence constraints may have to be added between flights two ensure that a maximum of two aircraft can be serviced at the same time. These local constraints will be removed after decoupling to allow SPs to have a maximum of flexibility to do its own refined planning given the prototype’s plan.

\(^8\) Note that input file (4), the stand matrix, is not yet taken into account in the prototype. This input will however be used in Phase II of the CAED project, when the prototype is foreseen to be extended by a complex resource allocation planning module.
In this figure, the first two input files have almost the same format. In both the manufacturer service times file and the airline service times file, a list of aircraft types is given specifying for each type the turnaround time and time intervals for each ground handling service. In the manufacturer service times file, for instance, the earliest start time and duration for the deboarding of a B777-200 may be [1; 7.5], whereas the airline’s service times file specifies [1; 10]. Similarly, the turnaround time for this aircraft type may be 45.0 (Boeing) and 127.0 (KLM) respectively. In this manner, all so-called timebound constraints (i.e., constraints tying variables to specific time values) for each service of each aircraft type is listed.
In the airline’s service times file, however, one additional source of constraints is added: the precedence constraints (see section 4.1, p. 33 of [D2, 2007] for a discussion on precedence constraints). These types of constraints specify which services should be performed before or after which other services. For the B777-200, for instance, the following precedence constraints are listed:

```
#=================================================
#---- Precedence Constraints
#=================================================
precedence;setbridge<unboard
precedence;unboard<cabin
precedence;unboard<catering_fwd
precedence;unboard<catering_aft
precedence;cabin<board
precedence;catering_fwd<board
precedence;catering_aft<board
precedence;board<removebridge
precedence;toilets<water
precedence;unboard<fuel
precedence;unload_baggage<load_baggage
#=================================================
```

Together, the timebound and precedence constraints lay down all temporal constraints that follow from the aircraft manufacturer and the airline.

The third input file contains the seasonal stand allocation plan as it has been created during the strategic planning phase. For each flight, a call sign, destination, in-block, off-block, stand/gate and aircraft type is given.

The fourth input file, the stand matrix, consists of a simple 2D matrix. This matrix defines the minimum distance between two stands/gates in minutes. It is assumed that these travel times are the same for each service provider. Note that, in the Phase I version of the prototype, this input file is read but not yet included – in the form of constraints – in the planning process.

The fifth input file consists of a list of tasks assigned to each actor. Moreover, the maximum number of resources (e.g., fuelling vehicles) and the priority that each actor has are included in this file. With this information, the prototype knows which tasks need to be planned per actor,

---

9 Note that in this matrix, the time to get from a gate X to a gate Y is typically not the same as the time to get from Y to X: the travel time depends on the routes and directions of traffic for ground handling vehicles at the airport.
how many tasks can be planned at the same time and which priority these tasks have relative to other actors’ tasks.

For more information about the input files, the reader is referred to section 6.1.

4.3 Pre-processing the input

After reading the input files, the prototype stores the information in suitable data structures. These data structures consist of vectors containing the minimal and norm service times (start times and duration) for each aircraft type that can be recognized by the prototype (see chapter 6). Another data structure is formed by a vector of flights to be planned, extracted from the high-level stand plan. The stand matrix is also stored in the form of a vector. These data structures will enable the prototype to quickly retrieve and process information when trying to decouple the planning problem and solve it locally. Chapter 5 and Appendix C of this document will give some more information about the preprocessor data structures implemented in the prototype.

4.4 Internal processes of the prototype

After creating appropriate data structures from the input, the prototype does the following:

1. construct a complete Simple Temporal Network (STN)
2. add local constraints to this STN
3. decouple the complete STN into a set of local STNs matching the stand plan
4. output the resulting stand and services plan

These four steps constitute the core of the prototype. For a description of the implementation of these steps, the reader is referred to the next chapter.

4.5 The output

Figure 3 below gives the output of the prototype: the decoupled stand and services plan (output (6) in figure 1). In this output, the seasonal stand plan is complemented with a plan for each service matching its constraints and the SP-specific local constraints. Note that these local plans do not yet specify the exact start and end time for each SP-specific task: instead, a bandwidth is defined within which each task should be planned (guaranteeing that no conflict arises with other tasks or other services).
Note that for each planned service, four time points are given. The first two specify the earliest and latest start time for the activity (e.g., unboard), the second two the earliest end time and latest end time. Both intervals together define the minimum and maximum bandwidth which can be used by the service provider to plan its own tasks.
5 Prototype Implementation and User Manual

This chapter describes the prototype’s implementation (section 5.1) at a high level. Code snippets are provided to illustrate how the prototype proceeds from input to output. In section 5.2, a short user manual is offered to assist the user in operating the prototype.

5.1 Implementation

For a detailed description of the main C++ classes constituting the prototype, the reader is referred to Appendix C. In this appendix, the following classes are documented:

- Decoupler (Decoupler class)
- Edgematrix (Datastructure for holding an STP)
- MainData (Providing high-level data processing routines)
- Output (Providing methods to write the results)
- ProcessInput (Input processing routines)
- STP (Class used to construct and process STPs)

Here, a high-level description is offered of the steps that the prototype takes to move from input to output - as distinguished in chapter 4. Note that the prototype has been implemented in C++ allowing it to be run either on a Linux/Unix or a Windows platform. At the start, the prototype should be called with the following arguments:

```c++
string service_manufacturer_filename;
string standplan_filename;
string service_airport_filename;
string standmatrix_fuelling_filename;
string task_actor_assignment_filename;
string delay_filename;
string output_dir;
double r_value;
```

The first five arguments correspond to the input files discussed in section 4.1. The 6th argument, the delay_filename, is optional. It defines possible delays with respect to the flights specified in the stand plan and corresponds to a possible tactical update of the scheduled information in that stand plan. Note that the delays specified in this file will be added as extra constraints to the problem domain (and the Simple Temporal Network constructed from it). The 7th argument indicates the directory to which the output should be written. The 8th argument, finally, is a value that can be used to specify how greedy the algorithm will be when trying to find a proper decoupling. A value between 0 and 1.0 should be specified; a value of 1.0 (the default) indicates that the algorithm should operate as greedy as possible.
After calling the prototype with these input parameters, class Main is invoked which in turn creates an object of class MainData (see Appendix C.3) and reads the input, sets the output and processes the input:

**Class Main**

```cpp
int main(int argc, char** argv)
{
    (..)
    MainData mydata = MainData(r_value);
    mydata.read_input(service_manufacturer_filename,
                      standplan_filename,
                      service_airport_filename,
                      standmatrix_fuelling_filename,
                      task_actor_assignment_filename,
                      delay_filename);
    mydata.setOutput(output_dir);
    mydata.processInput();
    (..)
}
```

In this code snippet, ‘(..)’ indicates that code has been left out. In class MainData offers, the three methods read_input(), setOutput() and processInput() are defined. Below, these three methods are described in brief. Together, these methods form the core of the prototype.

**Class MainData, method read_input()**

Method read_input of class MainData takes care of the reading of all required and optional input files. Starting from the arguments provided to the prototype (the locations of the input files), it calls all input-specific read-methods as specified in class ProcessInput:

```cpp
void MainData::read_input(  string servicetimes_manufacturer_filename,
                            string standplan_filename,
                            string servicetimes_KLM_filename,
                            string standmatrix_fuelling_filename,
                            string task_actor_assignment_filename,
                            string delay_filename) {
    ProcessInput::read_ACTypesManufacturer(  servicetimes_manufacturer_filename, servicetimes_manufacturer );
    ProcessInput::read_ACTypesAirport(  servicetimes_KLM_filename,
                                      servicetimes_KLM, precedence_constraints );
    ProcessInput::read_StandPlan(  standplan_filename, standplan );
    ProcessInput::read_StandMatrix(  standmatrix_fuelling_filename, standmatrix );
    ProcessInput::read_TaskActorAssignment(  task_actor_assignment_filename, task_actor_assignments );
    ProcessInput::read_Delays(delay_filename, delays);
}
```
// Consistency check on the datastructures.
inputDataValid();
return;
}

Class ProcessInput, detailed in Appendix C.5, subsequently reads all input files, storing all data in vectors of the corresponding type. If successful, program control returns to class MainData which proceeds by extensively checking the validity of all input.

Class MainData, method setOutput ()

This method is very short, and simply assigns the output directory (one of the parameters provided to the prototype) to a global variable called outputdir:

```cpp
void MainData::setOutput(string outdir){
    outputdir = outdir;
}
```

This outputdir will be used in ProcessInput where all input is processed and the results are written to the output directory.

Class MainData, method ProcessInput()

In this method, all the important work is done. It is defined as follows:

```cpp
void MainData::processInput(){
    (..)
    // Construct the STP
    constructSTP();

    (..)
    // Add the local constraints
    make_local_constraints();
    add_local_constraints( global_stp );

    (..)
    // Decouple
    Decoupler dec = Decoupler(*global_stp, partition(), r);
    dec.decouple();

    (..)
    // Get the decoupled STPs from the decoupler class and store them
    // in vector decoupled_stps
    vector<pair<string, shared_ptr<stp> > > decoupled_stps =
    dec.get_decoupled_stp();
```
// Loop through vector decoupled_stps and store each decoupled stp // in a separate file in the output directory
for (vector<pair<string, shared_ptr<stp> > >::iterator part_it = decoupled_stps.begin(); part_it != decoupled_stps.end(); part_it++) {
    Output::write_solution_space_EST_CSV(outputdir, part_it->first, part_it->second);
}

Thus, in this method, five important steps are taken here:

- A global STP is constructed based on the vectors containing all input data, using the STP-class (see Appendix C.6)
- Local constraints, specifying the sequence of tasks for each actor, are added to this STP
- The Decoupler class (see Appendix C.1) is invoked to create a decoupling of the global STP constructed
- The result is then stored in a vector called decoupled_stps that contains the locally independent STPs that have been created by the instantiation ‘dec’ of class Decoupler
- A loop is created to iterate through vector decoupled_stps and output for each element (each local STP, corresponding to each service provider in the domain), the solution to the ‘outputdir’ directory

This concludes a high-level description of the implementation of the prototype in C++.

5.2 User Manual

This section describes briefly how the prototype can be operated. As shown in section 5.1, a total of 7 parameters can be specified when calling the prototype. In general, the following commandline options are supported by the prototype:

```
--help                  Produce help message.
-m [ --m ] arg          Servicetimes manufacturer file
-a [ --a ] arg          Servicetimes Airline file
-s [ --s ] arg          Standplan file
-d [ --d ] arg          Standmatrix fuelling file
-t [ --t ] arg          Task actor assignment file
-p [ --p ] arg          Task delay file
-r [ --r ] arg          Value between 0 and 1.0. r < 1.0 is non-greedy.
-o [ --o ] arg          output directory name
```

*Figure 4: Commandline arguments supported by the prototype*
Thus, the parameters discussed in section 5.1 should be specified by using the commands above. For instance, to call the prototype with the five required input files stored in C:/Disk/scenario3, without a delay file and to specify that output should be written to C:/Disk/output3, the user should type

```
./prototype -m "C:/Disk/scenario3/servicetimes_manufacturer.txt"
-s "C:/Disk/scenario3/standplan.txt"
-a "C:/Disk/scenario3/servicetimes_KLM.txt"
-d "C:/Disk/scenario3/standmatrix_fuelling.txt"
-t "C:/Disk/scenario3/task-actor_assignments.txt"
-o "C:/Disk/output3"
```

in the command box in which the prototype is running.
Chapter 6: Prototype Testing and Graphical Extension

In this section, the input and test scenario is described that has been used to test the prototype (section 6.1). Section 6.2 outlines the test that has been conducted with this scenario and explains the resulting solution. Next, section 6.3 explores the performance of the prototype when constructing the STP and decoupling it into local networks. Section 6.4, finally, describes the Microsoft Excel macros that have been developed to graphically represent the prototype’s input and output.

6.1 Prototype Input

The prototype has been tested using a test scenario, the service times specified by both the airline and the aircraft manufacturer, the travel distances between gates and a list of tasks assigned per actor.

6.1.1 Test Scenario

Given the requirements laid down in section 3.1, a test scenario was required involving at least 10 aircraft to be planned, at least 20 stands and at least 3 aircraft types. To this end, on August 2, 2007 at 2:15 p.m. a total of 37 KLM flights have been recorded at Amsterdam Airport Schiphol. These flights constitute the baseline scenario that has been used for subsequent testing of the prototype. All flights have been recorded at the website: http://www.schiphol.nl. On this site, departure and arrival flights can be found, including aircraft type, stand/gate type, and on- and off-block times. Given this information, both legs of the flight (identified with different callsigns) can be coupled yielding a realistic stand plan as input for the prototype.

In order to comply with other available data (e.g., the service norm times and minimum times per aircraft type), some data in the scenario has been modified to match the specific input requests. Specifically, the following modifications have been performed:

- Some aircraft types have been changed to equivalent (in terms of service times/turnaround) aircraft types to match the service time information available (e.g., for Fokker 70/100 or A319/320, no airplane servicing arrangement was available; conversely, for B737-600 or Embraer170/195, no norm times were available). The following replacements have been made:
CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING

<table>
<thead>
<tr>
<th>Original a/c code</th>
<th>Corresponding a/c type</th>
<th>New a/c type code</th>
<th>Corresponding a/c type</th>
</tr>
</thead>
<tbody>
<tr>
<td>733, 735</td>
<td>Boeing B737-300</td>
<td>738</td>
<td>Boeing B737-800</td>
</tr>
<tr>
<td>100</td>
<td>F100</td>
<td>738</td>
<td>Boeing B737-800</td>
</tr>
<tr>
<td>321, 736, 73H</td>
<td>Airbus A321</td>
<td>738</td>
<td>Boeing B737-800</td>
</tr>
</tbody>
</table>

- Some departing flights that did not have a connecting arrival leg (e.g., aircraft staying overnight at Schiphol Airport), have been assigned a realistic on-block time: the flights to Kano, Houston, and Bonaire.
- The flight to London Heathrow was cancelled. Therefore, no stand has been assigned.

Below, the baseline scenario extracted from Amsterdam Airport Schiphol is given:

```plaintext
#=================================================
#---- Aircraft list
#-------------------------------------------------
KL0803;Manila;12:00;14:10;F02;B777_200
KL0577;Kano;12:05;14:05;F05;A330_200
KL0769;Bonaire;12:20;14:20;D53;MD_11
KL0893;Shanghai;12:30;14:35;F03;B747_400
KL0663;Houston;12:35;14:40;E02;B747_400
KL6037;Boston;12:59;14:55;E07;A330_200
KL1909;Hanover;13:05;14:10;B22;F50
KL1673;Barcelona;13:05;14:10;D71;B737_800
KL1237;ParisdeGaulle;13:10;14:10;C08;B737_400
KL1115;Stockholm;13:30;14:35;C04;B737_400
KL3157;Dublin;13:35;14:40;D24;B737_800
KL1811;Cologne;13:40;14:10;B27;F50
KL1307;Budapest;13:40;14:45;D18;B737_800
KL1019;LondonHeathrow;13:45;14:50;D52;B737_800
KL1407;Marseille;13:50;14:50;B28;B737_400
KL1591;Bologna;13:50;14:50;B29;B737_400
KL1189;Bergen;13:50;14:50;C09;B737_400
KL1477;Oslo;13:50;14:50;D63;B737_400
KL1655;Venice;13:50;14:50;D78;B737_400
KL2027;Lyon;13:55;14:55;B25;B737_400
KL1627;MilanMalp.;13:55;14:55;C04;B737_400
KL1285;Edinburgh;13:55;15:00;D05;B737_800
KL1365;Warsaw;13:55;14:55;D06;B737_400
KL1477;Glasgow;13:55;15:00;D49;B737_800
KL3399;Larnaca;13:55;15:00;E06;B737_800
KL1445;Aberdeen;14:00;15:00;D44;B737_400
KL3125;Prague;14:00;15:00;D65;B737_400
KL1311;Toulouse;14:10;14:50;B27;F70
KL1131;Copenhagen;14:10;15:15;C06;B737_800
KL1797;Munich;14:10;15:10;C10;B737_400
KL1931;Geneva;14:10;15:10;D27;B737_400
KL1315;Bordeaux;14:15;14:55;B23;F70
KL1157;Gothenburg;14:15;15:15;D76;B737_400
KL1729;Brussels;14:25;14:55;B26;F50
KL1859;Dusseldorf;14:30;15:00;D24;F50
KL1745;Luxembourg;14:40;15:10;B22;F50
KL1217;Sandefjord;14:50;15:15;B29;F70
#=================================================
```
6.1.2 Airline Service Times
The above scenario only represents one of the five input files for the prototype (file standplan.txt). Other input includes a file (servicetimes_KLM.txt) listing the service times as specified by the airlines involved in the scenario. In this case, only one airline is involved: KLM (see the requirement in section 3.1). KLM Ground Services norm times have been used for the following aircraft types:

- B737-400
- B737-800
- B777-200
- B747-400
- MD-11
- Fokker 50
- Fokker 70
- A330-200

Additionally, educated guesses have been made with respect to the norm times of yet two other aircraft types:

- Embraer 170
- Embraer 195

6.1.3 Aircraft Manufacturer Service Times
For the aircraft manufacturer service times (file servicetimes_manufacturer.txt), data has been retrieved for the following aircraft types:

- B737-400
- B737-800
- B777-200
- B747-400
- MD-11
- Embraer 170
- Embraer 195

Moreover, three other aircraft types have been added based on educated estimates:

- Fokker 50
- Fokker 70
- A330-200
6.1.4 Stand Matrix
The stand matrix (file stand_matrix.txt) specifies the travel time required between the gates in the scenario. In the scenario, a total of 33 gates have been used, rendering a matrix of 33x33.

A number of assumptions have been taken into account when specifying the travel times between gates (of which no real data is available). Note that these assumptions are made with respect to the topology of a specific airport: Amsterdam Airport Schiphol.

First, it is assumed that the travel times for all actors (fuelling, catering, cleaning, etc.) are the same. Second, it is assumed that 2 minutes is sufficient to travel between two gates on the same pier. Third, for two adjacent piers, 3 minutes is sufficient. Fourth, between piers that are two piers away from each other, 3.5 minutes is sufficient. Fifth, for piers that are three piers away from each other, 4 minutes is required. Finally, between pier B or C and pier G and H, 5 minutes is required.

6.1.5 Task-actor assignments
Given the requirement that at least five actors should be involved, we have chosen to include baggage, cleaning, catering, fuelling, and boarding as services in the prototype’s input. Given the fact that water and toilet services are also included in the aircraft manufacturer and airline norm times, these two extra actors have also been included. Note that the prototype is generic with respect to the actors and tasks to be planned. Both the (number of) actors and the tasks that fall under their responsibility can be varied depending on the definitions outlined in the task-actor assignment file.

The file task-actor_assignments.txt first specifies which tasks are to be performed by which actors. For example, for the boarding company:

```
#---- Boarding Company
actor;boarding_company
tasks;unboard;board
```

specifies that tasks “unboard” and “board” are part of this company’s responsibility. Second, the file specifies the number of resources that are available for each actor. For instance:

```
#---- Catering Company
actor;caterer
resources;2
```
denotes that the catering company can employ two teams at the same time. Finally, the file lists the priority that actors have with respect to other actors. For instance:

```plaintext
#---- Airport Authorities
actor;airport_authorities
priority;1
```

indicates that the airport authorities have priority 1 (the highest priority). Priorities can thus be given with respect to the size of the time intervals assigned to actors. In general, actors with a high priority will be assigned a larger time interval than actors with a lower priority in case both need to plan activities in the same time interval. Note that priorities are relative to one another: a priority of 1 for the boarding company and 2 for the cleaning company will lead to the same decoupling as a priority of 10 and 20, respectively.

### 6.2 Prototype Test

The prototype has been tested using the above scenario (as standplan.txt file). Other input files include files listing the service times of both the airline and the aircraft manufacturer (servicetimes_KLM.txt and servicetimes_manufacturer.txt), a file specifying the travel distances between gates (standmatrix.txt) and a file listing the tasks assigned per actor (task-actor_assignments.txt). See chapter 5 for more details. Below, a screenshot of the prototype shows how the input files are processed, how a STP is constructed from the data, how this STP is solved in 1 second and how this STP is in turn decoupled in less than 1 second.

*Figure 5: Screenshot of the prototype reading and processing the input files*

After decoupling, the prototype lists for each task of each actor (e.g., task KL8004_cabin or KL1131_fuel) on the screen at least the minimum start time and maximum end time between which this activity should be planned. This output is written to the output directory specified as
a prototype parameter (see section 5.1). After a successful run, the output directory may contain the following files:

- `airportAuthorities_solution_space.csv`
- `airportAuthorities_solution_space_with_est.csv`
- `boarding_company_solution_space.csv`
- `boarding_company_solution_space_with_est.csv`
- `cargo_company_solution_space.csv`
- `cargo_company_solution_space_with_est.csv`
- `caterer_solution_space.csv`
- `caterer_solution_space_with_est.csv`
- `cleaning_company_solution_space.csv`
- `cleaning_company_solution_space_with_est.csv`
- `fuelling_company_solution_space.csv`
- `fuelling_company_solution_space_with_est.csv`
- `toilet_servicing_company_solution_space.csv`
- `toilet_servicing_company_solution_space_with_est.csv`
- `water_servicing_company_solution_space.csv`
- `water_servicing_company_solution_space_with_est.csv`
- `with_global_solution_space_with_est.csv`
- `without_global_solution_space_with_est.csv`

**Figure 6: List of output files generated by the prototype**

Note that for each actor, two files are created. The first file (e.g. `fuelling_company_solution_space.csv`) contains the unformatted minimum output of the prototype. The minimum and maximum time for each activity is denoted using a uniquely identifiable format that includes references to the year and month of the time point. An example activity is denoted as:

```
KL1591_fuel,2007-Aug-24 14:00:01.125000,2007-Aug-24 14:41:00
```

to indicate that flight KL1591 should be fuelled between 14:00 and 14:41 on the day of operations.

The second file (e.g. `fuelling_company_solution_space_with_est.csv`) contains the formatted and maximum output of the prototype – this file should generally be used as output. For flight KL1591, this file specifies for instance:

```
KL1591 Fuel 14:00:01
```

---

10 The latter part of this filename, ‘with_est’, indicates that this file includes the earliest start time (est) and earliest end time of all tasks to be performed by the actor. This file gives the earliest start time solution of the network: see [D1, p. 16-17].
In the second column, the task is specified (fuel). In the third column, the earliest start time of the activity is given. The fourth column yields the Earliest Start Time (EST) solution - in this case, this solution equals the earliest start time in the third column. The fifth column defines the earliest end time. The last column, finally, lists the latest end time for the activity.

Some words of explanation are required here. First of all, note that the maximum range of [14:00, 14:41] for fuelling stems from the KLM norm times. This range, matching the aircraft type for this specific flight (a B737-400), defines the maximum time fuelling may take within the turnaround interval. Within this range, however, a minimum time may be defined based on the minimum service times as specified by the aircraft manufacturer. Since Boeing specifies that the fuelling of a B737-400 can be performed in 13 minutes, the fifth column states that fuelling can be completed as early as 14:13. In short, the range [14:00, 14:00, 14:13, 14:41] specifies the interval and latitude that can be used by the fuelling company to plan its service for flight KL1591.

### 6.3 Performance

The performance of the prototype can be divided into two steps that the prototype needs to perform:

- The construction of a minimum Simple Temporal Network\(^\text{11}\) based on all input data
- The decoupling of the STN into local networks

As can be observed from the screenshot in section 6.2, both tasks are completed within 1 second for the given scenario (37 flights). This is very quick, especially if one takes into account that the temporal network that is constructed includes over 800 nodes and thousands of temporal constraints. Thus, requirement 3 of section 3.2, stating that the prototype should produce a plan within max. 1 minute, is satisfied for our 37-flights-scenario.

To get an idea of the performance boundaries of the prototype, much more extensive testing is required involving very large and complex scenarios. This falls out of the scope of Phase I of the project. In Phase II, the prototype will be extended (including resource allocation and local STN solving, amongst other things). Furthermore, extensive testing against large-scale scenarios of the resulting prototype is foreseen.

\(^{11}\) A minimum STN is a STN in which all constraint redundancies have been removed. The process of minimizing involves the ‘tightening’ of all temporal constraints in the network, rendering a minimal network. See section 5.2, p. 40 ff in [D1, 2007].
6.4 Graphical Input and Output Add-on

In order to get a good insight into the input and output of the prototype, a total of four different macros have been written in Excel as an add-on to the prototype. These macros contain code in Visual Basic for Applications that automatically processes data and constructs a graphical representation from it. Below, these macros are described and example graphs are given of the test scenario and its results.

6.4.1 Macro ProduceStandPlanGraph.xls

This macro has been written to automatically read the input scenario standplan.txt into Excel and produce a graph based on this data. When executed, first an Excel worksheet called ‘standplan’ is created when reading standplan.txt (which should be place in directory C:\Temp\Caed_input):

<table>
<thead>
<tr>
<th>callsign</th>
<th>destination</th>
<th>in-block</th>
<th>off-block</th>
<th>duration</th>
<th>gate</th>
<th>aircraft</th>
</tr>
</thead>
<tbody>
<tr>
<td>KL0803</td>
<td>Manila</td>
<td>12:00</td>
<td>14:10</td>
<td>2:10</td>
<td>F02</td>
<td>B777_200</td>
</tr>
<tr>
<td>KL0577</td>
<td>Kano</td>
<td>12:05</td>
<td>14:05</td>
<td>2:00</td>
<td>F05</td>
<td>A330_200</td>
</tr>
<tr>
<td>KL0769</td>
<td>Bonaire</td>
<td>12:20</td>
<td>14:20</td>
<td>2:00</td>
<td>D53</td>
<td>MD_11</td>
</tr>
<tr>
<td>KL0893</td>
<td>Shanghai</td>
<td>12:30</td>
<td>14:35</td>
<td>2:05</td>
<td>F03</td>
<td>B747_400</td>
</tr>
<tr>
<td>KL0663</td>
<td>Houston</td>
<td>12:35</td>
<td>14:40</td>
<td>2:05</td>
<td>E02</td>
<td>B747_400</td>
</tr>
<tr>
<td>KL6037</td>
<td>Boston</td>
<td>12:59</td>
<td>14:55</td>
<td>1:56</td>
<td>E07</td>
<td>A330_200</td>
</tr>
<tr>
<td>KL1909</td>
<td>Hanover</td>
<td>13:05</td>
<td>14:10</td>
<td>1:05</td>
<td>B22</td>
<td>F50</td>
</tr>
<tr>
<td>KL1673</td>
<td>Barcelona</td>
<td>13:05</td>
<td>14:10</td>
<td>1:05</td>
<td>D71</td>
<td>B737_800</td>
</tr>
</tbody>
</table>

*Figure 6: Part of Excel Worksheet ‘standplan’ that macro ProduceStandPlanGraph.xls creates.*

Next, another worksheet is created that contains a chart that graphically represents the input scenario. Figure 7 below gives this chart:
CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING

Stand plan chart

Figure 7: Standplan chart created by macro ProduceStandPlanGraph.xls

In this chart, every flight’s turnaround time is represented by a bar starting at the scheduled on-block time and ending at the scheduled off-block time. The total turnaround time is indicated at the right side of each bar. On the Y-axis, each flight’s unique callsign is listed; on the X-axis, the time of the day is indicated (between 11:30 and 15:30 in this case).

6.4.2 Macro ProduceStandPlanAndDelayGraph.xls

This macro is similar to the previous macro, except that instead of reading only file standplan.txt, it reads a file called delayed_standplan.txt from directory C:\Temp\Caed_input as well. Whilst file standplan.txt specifies the scheduled off-block times for each flight, the latter file specifies any changes that may have occurred to these scheduled times at the day of operations. Thus, a comparison can be made between the strategic or pre-tactical planning of standplan.txt, and the tactical or operational situation in delayed_standplan.txt. The macro reads both files, comparing the off-block times of each flight; next, in case of delays, an extra yellow bar is added to each turnaround bar to indicate how large the delay is. An example chart may look like this:
CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING

Figure 8: Standplan chart created by macro ProduceStandPlanAndDelayGraph.xls.

6.4.3 Macro SelectAndFilterFlight.xls

Above, two macros have been discussed to visualize the input to the prototype. Below, we will discuss two macros that should be executed to visualize the prototype’s output.

Macro SelectAndFilterFlight.xls allows the user to do three things:

- Show services
- Filter flight
- Produce services graph

These three possibilities correspond to three sub-macros. To start with the first, ShowServices.xls: this macro reads all output as listed in the prototype’s output directory (see section 6.2 above) as constructs worksheets for each actor listing all its tasks and corresponding time intervals. For instance, for the boarding company the following partial worksheet is created:

KL1909 unboard 13:05:00 13:05:00 13:07:54 13:20:44
Figure 9: Part of Excel Worksheet 'boarding' that macro ShowServices.xls in SelectAndFilterFlight.xls creates.

The following sub-macro can then be invoked to filter out a specific flight in which the user is interested. When running this sub-macro, the following dialogue box appears:

![Select Flight](image)

When entering flight KL0577, the sub-macro automatically gathers all ground handling activities that need to be performed for this flight, including the earliest start time, the earliest end time and the latest end time for each activity. The result for flight KL0577 may look like this:

<table>
<thead>
<tr>
<th>callsign</th>
<th>activity</th>
<th>start</th>
<th>earliest end</th>
<th>latest end</th>
</tr>
</thead>
<tbody>
<tr>
<td>KL0577</td>
<td>unload</td>
<td>12:05:00</td>
<td>12:31:00</td>
<td>13:28:00</td>
</tr>
<tr>
<td>KL0577</td>
<td>load</td>
<td>12:31:00</td>
<td>13:15:00</td>
<td>14:10:00</td>
</tr>
<tr>
<td>KL0577</td>
<td>toilets</td>
<td>12:05:00</td>
<td>12:31:00</td>
<td>13:06:30</td>
</tr>
<tr>
<td>KL0577</td>
<td>water</td>
<td>13:06:30</td>
<td>13:36:30</td>
<td>14:06:00</td>
</tr>
<tr>
<td>KL0577</td>
<td>setbridge</td>
<td>12:05:00</td>
<td>12:06:00</td>
<td>12:07:06</td>
</tr>
<tr>
<td>KL0577</td>
<td>removebridge</td>
<td>14:06:12</td>
<td>14:07:12</td>
<td>14:12:12</td>
</tr>
</tbody>
</table>

Figure 10: Part of Excel Worksheet 'RESULT' that macro FilterFlight.xls in SelectAndFilterFlight.xls creates for flight KL0577.

Finally, another sub-macro can be executed to create a graph based on the above information. This sub-macro, ProduceServicesGraph.xls, will create the following graph for flight KL0577:
Figure 11: Graph produced by macro ProduceServicesGraph.xls in SelectAndFilterFlight.xls creates for flight KL0577.

In this graph, each ground handling activity is represented by a purple bar listing the minimum time required for that activity, and a yellow bar indicating the maximum interval in which this activity can be planned.

6.4.4 Macro SelectAndFilterFlight.xls

The above macro offered functionality to produce graphs for one specific flight, listing all associated ground handling activities and corresponding minimum and maximum time intervals. Sometimes, however, it may be useful to get an overview of exactly the opposite: a graph showing all flights as planned for one specific ground handler. For example, the user may want to know how the fuelling services are distributed over the flights. To this end, another macro has been implemented to create such a graph. Below, a graph for all fuelling activities is shown which results from running macro SelectAndFilterFuelling.xls:
Figure 12: Graph produced by macro SelectAndFilterFuelling.xls.

Note again that the minimum and maximum time interval as assigned by the prototype are represented by a purple bar (left) and yellow bar (right).

This concludes the description of add-ons to the prototype. In the next section, the use of the prototype’s output in the tactical phase is discussed.
7 Using the Prototype’s Output in the Tactical Phase

In this chapter, we assume that the prototype has already created a stand and services plan through decoupling (the pre-tactical phase). In the tactical phase, the prototype’s output is used to assist service providers in re-planning their pre-tactical plan in order to solve unforeseen disruptions. This chapter provides a description of how the prototype’s output can be used in the tactical phase. Note that in Phase I of the project it will suffice to show that the pre-tactical plan established through decoupling demonstrates tactical advantages.

7.1 Introduction

In this section, the advantages offered by the prototype in the pre-tactical and tactical planning phases are illustrated by means of a simple example. In the next two subsections, a more formal discussion is provided of how the prototype can be employed.

After running the prototype in the pre-tactical planning phase, a set of decoupled plan representations is created which can be utilized by the service providers to do their own planning. For instance, a fuelling company may receive two maximum intervals for aircraft X and Y to be fuelled; within this interval, as we have seen, the prototype also provides a minimum interval based on the minimum time required to fuel these specific aircraft. Having no knowledge about any fuelling company specific constraints, the prototype typically chooses the earliest start time for both these minimum intervals:

![Figure 13: Example local plan representation for fuelling aircraft X and Y.](image)

In Phase II of the project, extra functionality may be added to the prototype in order to actually support the tactical re-planning process.
Based on this prototype output, however, the fuelling company is free to pre-tactically plan its services as it sees fit. As long as fuelling stays within the yellow bar, the local solution is guaranteed to be conflict-free with respect to all other ground handling services around X and Y. For instance, when the fuelling company only wants to employ one fuelling vehicle for aircraft X and Y, it may choose to locally plan these services as follows:

![Figure 14: Example pre-tactical plan for fuelling aircraft X and Y.](image)

In Figure 14, the fuelling company has added a local constraint specifying that aircraft X should be fuelled before Aircraft Y. Within this order, there is still some margin (compare the orange bar with the encompassing yellow bar) for both aircraft to slightly shift the fuelling service if required. More importantly, both aircraft can now be fuelled by the same fuelling vehicle without conflicting with one another, with other aircraft to be fuelled, and with all other ground handling activities at the airport.

Now let us assume that on the day of operations aircraft X arrives 5 minutes late. This means that the start and end time for fuelling X needs to be delayed by 5 minutes. Since the yellow rectangle is still larger than the orange one, however, it is possible to simply adjust the scheduled time for fuelling Aircraft X by “right-shifting” the orange block 5 minutes:

![Figure 15: Tactical shifting based on a 5 minute delay of aircraft X.](image)
The result is shown in the bottom part of Figure 15. Aircraft X will be fuelled a little later than scheduled originally, without interfering with Aircraft Y. It was not necessary to adjust the planning for Aircraft Y.

Let us assume however that the delay of X, originally estimated to be 5 minutes, grows to a delay of 30 minutes. In that case, it may not be possible to simply shift aircraft X within its local planning margin – the delay is too large to fit within the yellow bar. Something else needs to be done here. Since we know that the local constraint of fuelling X before Y was added after decoupling and by the fuelling company itself, however, it is not difficult to find a solution.

In the original prototype output, both aircraft X and Y could be fuelled somewhere in the entire range of the maximum interval (see Figure 13; this range is indicated by light-yellow bars in Figures 14 and 15). Therefore, the fuelling company can simply change its locally added constraint, fuelling aircraft Y before aircraft X instead of the other way around. This solution is shown in Figure 16:

![Figure 16: Tactical re-planning based on a 30 minute delay of aircraft X.](image)

In this case, re-planning has been performed on a local level. Because of the decoupled plan representation, produced by the prototype, a solution could be found locally – again without involving any other aircraft or service providers.

This concludes the introductory discussion on how the prototype’s results can be beneficially employed in the pre-tactical and tactical phase. In the next two sections, a more detailed, formal description is given of how the local STNs produced by the prototype can be modified in the
light of tactical/operational changes. Two disruption scenarios will be considered: a delayed service provider and a delayed arrival.

7.2 Delayed service provider
A delayed service provider is a frequent occurrence at airports. In this scenario, a service provider suffers an unforeseen delay, threatening to delay the entire turnaround process of the aircraft in question (and, hence, the estimated time of departure). Moreover, the delay may have knock-on effects for the servicing of subsequent aircraft. The prototype’s decoupled plan can help resolve the delay in an elegant manner, however. In general, two steps may be taken - depending on the gravity of the delay.

In the first step, the service provider tries to absorb the delay itself. Since its local planning is decoupled from the other service providers, re-planning can take place without requiring any further co-ordination with other service providers. For example, consider the example planning of [D2, p. 38], reproduced in Appendix A of this document. Again, this example involves the fuelling of two aircraft, X and Y. Furthermore, assume that the fuelling company suffers a delay of 5 minutes before it can start fuelling aircraft X. In this case, the temporal constraint between $x_0$ and $x_2$ will be changed from $[8, \infty]$ (the time between 12:00 a.m. and start fuelling) to $[8+5=13, \infty]$. Given this change, two other updates are required to ensure the network’s consistency. First, the temporal constraint between $x_2$ and $x_3$ will be $[10,32]$ instead of $[10,37]$; second, the constraint between $x_0$ and $x_3$ is modified to $[10+13=23, 45]$ instead of $[18,45]$. Figure 17 below gives the resulting STN. Compare Appendix A for ease of reference.
The fuelling company can now simply repair its plan based on the updated problem representation – without bothering other service providers. In this re-planning, local constraints (availability of personnel, vehicles, routes and travel time between aircraft) will be taken into account. An example solution may be, for instance, to simply shift all fuelling activities 5 minutes in time. Compare this solution with the original planning of [D2, p. 41], as reproduced in Appendix B of this document.

In this case the fuelling company has been able to find a new solution within the constraints of the fuelling company’s local STN, which may again be merged into a solution for the overall problem. Thus this disruption has been solved locally by the fuelling company.
In case of a large delay, the fuelling company may try to deploy extra resources to still adhere to the planning. If this is not possible, and the delay is such that the fuelling company cannot absorb it in its own planning, a second step will be taken.

In the second step, the fuelling company will try to upscale re-planning to the next level. At this level, one other service provider will be involved. This will be a service provider that is temporally dependent on the fuelling company: for instance, the boarding company. This service provider is on the so-called critical path, meaning that the start or end time of fuelling company affects – on its path – the start of end time of the boarding company (see Figure 19 below for example critical paths). In this case, a local STP will be decoupled from the original overall STP for both service providers together. Based on this STP, encompassing both fuelling and boarding, the two service providers may try to solve the disruption together. If they can, we have still kept re-planning as locally as possible; if they cannot, this process may be repeated involving yet another service provider.

Details of this procedure of upscaling the re-planning level fall out of scope of Phase I of this project (see section 3.1). In Phase II, however, this functionality is foreseen to be added to the prototype in order to better handle a variety of disruptions. Depending on the gravity of the disruption, the Phase II prototype will extend the number of service providers in a manner that still guarantees that solutions will be sought as locally as possible.

![Figure 18: An example local planning for the fuelling service](image-url)
7.3 Delayed arrival

The delayed arrival scenario results from an inbound flight being delayed, threatening to delay the ground handling services at the planned stand.

Consider a delayed arrival as the disruption (with a 15 minute delay). Furthermore, assume that the stand plan is rather tight: something has to be done in order to absorb the delay. If not, the airline will have to pay a fine for the extra time at the gate, or worse, risks the aircraft to be towed away – when the extra time at the gate exceeds a certain limit. In this case, instead of figuring out which parties should contribute to resolving the delay (which is a time-inefficient process), we follow the following simple procedure.

First, we observe that some parties affect each other (i.e., having a temporal dependency between each other), whilst others don’t. For example, the fuelling company has to wait until deboarding is finished before it can execute its task, whilst the boarding process cannot start unless catering and cleaning is finished. On the contrary, fuelling can take place during the unloading and loading of baggage – there is no temporal dependency here. In summary, only the ground handling activities that – given the 15 minute delay – will delay the end time of the turnaround will be considered.

Between these critical activities, we will share the total time of delay evenly. Of course, many more advanced algorithms may – and probably should – be applied here. For now, however, we assume that all parties will agree to share the pain evenly between those parties that can actually contribute to a reduction of the delay.

In our example, the question is how to distribute the 15 minute delay amongst the ground handling parties in order to absorb it and still make the scheduled off-block time. Let us restrict the problem to the five main ground handlers distinguished in chapter 3: baggage, cleaning, catering, fuelling, and boarding. For these five, depending on the aircraft type, one might schematically represent the chain of temporally dependent parties as follows:

---

13 For example, one might consider the time a service provider has for its activity as an indication for the amount that service provider may be able to contribute to reducing the delay.
Figure 19: two chains of temporally dependent parties

Distributing the 15 minute delay evenly between these five parties will then entail the following. First, we can observe that there are two critical paths. In the first path, the boarding company should reduce its deboarding time by 15 / 3 = 5 minutes, whilst the fuelling, catering and cleaning companies will have to do the same. Furthermore, since the boarding company is again in the critical path, it should also reduce the boarding time by 5 minutes.

In the second path, since the cargo company does not depend on any other parties (i.e., it can deliver its services anywhere between T=0 and T=end), it can use the entire turnaround time to unload and load baggage. Given the delay, however, it will have to reduce this planning bandwidth by 15 minutes: between T=15 and T=end.

From here on, the process proceeds in a similar fashion as above (the delayed service provider). For example, in case of the fuelling company, a 5 minute speed-up of the fuelling process is required. Consider again the example planning of Appendix A (See: [D2, 2007], p. 38), involving two aircraft X and Y. Furthermore, let us assume that aircraft X has a delay of 15 minutes. The temporal constraint between x₀ and x₂ will be changed now from [8, ∞] (the time between 12:00 a.m. and the start of the fuelling) to [8+10=18, ∞], since de-boarding will have reduced the delay in the critical path already to 10 minutes. Further, the time at which fuelling should be finished is required to be at most 5 minutes later than originally planned, instead of 15 minutes – since the fuelling company should, following de-boarding, contribute 5 minutes to the delay reduction. Thus, the temporal constraint between x₂ and x₃ will be [10,32] instead of [10,37], whilst the constraint between x₀ and x₃ is modified to [18+10=28, 45+5=50] instead of...
[18,45]. Figure 20 below gives the resulting STN. Compare with Appendix A for ease of reference.

Figure 20: The decoupled fuelling service in case of a delayed arrival of 15 minutes

Given this updated planning problem representation, the fuelling company can now proceed to do its own planning. An example solution may be, for instance, to reduce the time between the end of fuelling X and the start of fuelling Y (compare with Appendix B):
Of course, many other solutions may be chosen. For instance, the fuelling company may choose not to reduce the time between X and Y but instead to simply start fuelling Y later (knowing that fuelling Y can last until $t=75$). The solution chosen depends on the fuelling company itself: its local constraints, business model, preferences, planning rules. Note that if the fuelling company cannot find a solution, it may decide to deploy extra resources (an extra fuelling vehicle + personnel) if available\textsuperscript{14}. In this manner, the fuelling company might still be able to solve the newly constrained planning problem.

\textsuperscript{14} As mentioned above, this solution falls outside the scope of the Phase I prototype.
8 Conclusions

In this document, the turnaround model and corresponding implementation developed in the CAED project have been described. Chapter 1 started out by describing the context of the project and the structure of the WP3 document. In chapter 2, the main objectives and user group identified for the turnaround model have been listed. In chapter 3, the scope of the model has been discussed. Based on these modelling decisions, a set of high-level requirements for a prototype was listed. Chapter 4, in turn, focused on the prototype implementing the model. Some design decisions were discussed, and the functionality of the prototype was outlined in a step by step fashion. Chapter 5 detailed the prototype’s implementation and offered a short user manual. In chapter 6, a scenario has been described that was used to test the prototype; elementary test results have been provided to demonstrate that the prototype meets its requirements. Chapter 7, finally, discussed how to benefit from the prototype’s result, a pre-tactical stand and services plan based on decoupling, in the tactical phase.

In summary, in this deliverable – together with both other deliverables in the project – the innovative idea of decoupling in planning has been elaborated and developed into a mature model and prototype. This model and its prototyped implementation have been applied to the domain of ground handling, demonstrating that in this domain the pre-tactical planning based on decoupling can offer important advantages. These advantages will show up in the tactical phase, when disruptions necessitate re-planning of the original plan. In these cases, the decoupled local plans offered by the proposed new planning approach ensure that re-planning can be kept as local as possible – whilst guaranteeing that a solution does not conflict with other plans. This enables ground handlers to solve many tactical disruptions locally, thus drastically reducing the co-ordination overhead involved in negotiating with other parties. Given the large number of plan disruptions occurring daily at airports, and the increase expected in air traffic, such a planning tool seems to be a valuable asset.

This document constitutes the third and final deliverable of Phase I of the CAED project. In Phase II, a number of new steps are foreseen. First of all, the prototype developed in Year I shall be extended towards a full-blown planning tool, assisting the user by means of a graphical user interface into the pre-tactical planning of a large variety of ground handling services. Second, the planning tool shall be tested against a much larger scenario; further, testing of its tactical advantages should be performed in a realistic setting involving real planners to quantify the tactical benefits of the approach. Third, software support can be offered to the user when re-planning its plan based on the decoupled output of the initial prototype.
Apart from this, some entirely new features are foreseen to be added to the existing model and prototype. First, an extension of the model is foreseen involving the allocation of resources. To this end, the travel times of resources between stands, the number and availability of vehicles or personnel and other constraints will be added to the model and planning tool. This will allow local planners to use their own task assignment system to determine at any point in time which task should be executed when, and by which resource.

Additionally, some of the ideas presented in brief in chapter 7 of this document can be further developed to enhance the flexibility of the decoupling approach. For instance, future research could focus on the level of decoupling. The domain may be partitioned not into separate ground handlers, as is done now, but to a next level: into individual equipment (fuelling vehicles, catering vehicles, etc.) or personnel. Alternatively, one may group certain service providers together (e.g., cleaning and catering) for reasons of efficiency. Yet another topic concerns the implementation of a mechanism to upscale the level of decoupling when local re-planning is not feasible. In such a mechanism, a new decoupling is produced to group two (or more) service providers together if either one of them cannot find a re-planning solution individually.
9 References

References are divided into document references and online references.

9.1 Document References

AAM ConOps  Eurocontrol, *Eurocontrol ATM Operation Concept - Concept of Operations for 2011*, volume 2, Ed. 1.0, May 2005


9.2 Online references

AAM Eurocontrol, Airport Airside Management project, Airport Operations Architecture Description Document. In the course of this project a number of Use Cases and Operational Scenarios have been developed to describe operations around stand allocation. Also, actors lists including e.g. the Stand Planner have been developed. Downloadable at:
http://www.eurocontrol.int/oca/gallery/content/public/docs/OATA-P2-D4.2.7-04-52%20Airport%20Operations%20ADD.pdf
See page 13 for the ‘Stand Planner’ actor; page 18 for the ‘Establish Stand Allocation Plan’, page 20 for the ‘Update Stand Allocation Plan’.

Boeing Boeing website, Airplane Servicing Arrangements, downloadable at:
http://www.boeing.com/commercial/airports/acaps

CDM Handbook Eurocontrol, Airport CDM Guide, February 2003, downloadable at:
http://www.eurocontrol.int/airports/gallery/content/public/pdf/cdm_guide.pdf

G2G Eurocontrol, Gate to Gate Programme (5th Framework, started in 2002). For more project information:
http://www.eurocontrol.int/eee/public/standard_page/SSP_gate_to_gate.html

PR Eurocontrol, Press Release, 3 November 2004. Paul Wilson is quoted in this press release, downloadable at:

TAM Eurocontrol, Total Airport Management preliminary study, final report, November 2001, downloadable at:
http://www.eurocontrol.int/care-innov/gallery/content/public/docs/studies2001/tam_finalreportr1.1.pdf
Appendix A  The Decoupled Fuelling Service

Below, the decoupled fuelling service as given in [D2, p. 38] is reproduced for ease of reference.

![Diagram of the decoupled “fuelling” service](image)
Appendix B  Original Example Planning for the Fuelling Service

Below, the original planning (without the 5 minute delay) of the fuelling service, as given in [D2, p. 41], is reproduced for ease of reference.

Figure 23: An example local planning for the “fuelling” service
Appendix C Prototype Implementation Documentation

This appendix lists the main classes, structs, unions and interfaces of the CAED Prototype including a brief description. The following main classes are documented:

- Decoupler (Decoupler class)
- Edgemat (Datastructure for holding an STP)
- MainData
- Output
- ProcessInput
- STP

C.1 Decoupler Class

Decoupler class.

#include <Decoupler.h>

Public Member Functions

- **Decoupler** (const stp &s, vector<boost::tuple<string, int, string_set> > parts, double r_val)
- void decouple()
- shared_ptr<stp> getSTP()
- vector<pair<string, shared_ptr<stp>>> get_decoupled_stp()
- bool checkDecoupling()

Detailed Description

Decoupler class. Used to decouple a single STP into several sub-STPs.

Constructor & Destructor Documentation

Decoupler::Decoupler (const stp & s, vector<boost::tuple<string, int, string_set> > > parts, double r_val)

Constructor.

Parameters:

- s The stp that we want to decouple.
- parts The names of the nodes that are in various partitions
- r_val The speed of convergence: 1.0 is a greedy approach.

Member Function Documentation
bool Decoupler::checkDecoupling ()

Returns:
True if all xy-edges are dominated.

void Decoupler::decouple ()
Decouple the stp.

vector< pair< string, shared_ptr< stp > > > Decoupler::get_decoupled_stp ()

Returns:
A vector with the sub-STPs.

shared_ptr< stp > Decoupler::getSTP ()

Returns:
The stp contained in the decoupler.

The documentation for this class was generated from the following files:
- Decoupler.h
- Decoupler.cc

C.2 Edgematrix Class

Datastructure for holding an STP.
#include <Edgematrix.h>

Public Member Functions
- void insert_variable (string name)
- void remove_variable (string name)
- bool contains (string name) const
- void create_edge (string from, string to)
- bool remove_edge (string from, string to)
- bool exist_edge (string from, string to) const
- time_duration get_constraint (string from, string to) const
- time_duration get_distance (string from, string to) const
- string get_dominator (string from, string to) const
- void set_constraint (string from, string to, time_duration value)
- void set_distance (string from, string to, time_duration value)
- void set_dominator (string from, string to, string name)
- shared_ptr< list< string > > get_variable_list () const
- shared_ptr< list< pair< string, string > > > get_edge_list () const
- shared_ptr< vector< string > > get_triangles (const string from, const string to) const
- void make_triangles ()

**Detailed Description**

Datastructure for holding an STP.

**Member Function Documentation**

**bool Edgematrix::contains (string name) const**

Returns true if variable exist, false otherwise.

**Parameters:**

name The name of the variable.

**void Edgematrix::create_edge (string from, string to)**

Creates and edge between from and to. Assert(false) if edge already exists.

**bool Edgematrix::exist_edge (string from, string to) const**

Returns:
True if the edge between from and to exists, false otherwise.

**time_duration Edgematrix::get_constraint (string from, string to) const**

Returns:
The constraint of the edge between from and to.

**time_duration Edgematrix::get_distance (string from, string to) const**

Returns:
The distance of the edge between from and to.

**string Edgematrix::get_dominator (string from, string to) const**

Returns:
The dominating node of the edge between from and to. Returns an empty string ("") if edge is not dominated.

**shared_ptr< list< pair< string, string > > > Edgematrix::get_edge_list () const**

Returns:
A list with all edges contained in the datastructure.

**shared_ptr< vector< string > > Edgematrix::get_triangles (const string from, const string to) const**
Runs in \( O(1) \) if a triangle structure exists (see \texttt{void make_triangles()}). Runs \( O(n) \) otherwise (where \( n \) is the degree of the node).

Returns:
A list with all variables with which the edge between \texttt{from} and \texttt{to} can make a triangle.

\texttt{shared_ptr< list< string > > Edgematrix::get_variable_list () const}

Returns:
A list with all variables contained in the datastructure.

\texttt{void Edgematrix::insert_variable (string name)}

Inserts variable with the specified name. \texttt{Assert(false)} if variable already exists.

Parameters:
\texttt{name} The name of the variable.

\texttt{void Edgematrix::make_triangles ()}

Makes (temporary) datastructure which contains all triangles in the graph. The datastructure is invalidated by calls to \texttt{insert_variable()}, \texttt{remove_variable()}, \texttt{create_edge()} and \texttt{remove_edge()}.

\texttt{bool Edgematrix::remove_edge (string from, string to)}

Removes the edge between \texttt{from} and \texttt{to}. Does nothing if edge doesn't exist.

\texttt{void Edgematrix::remove_variable (string name)}

Removes variable with the specified name. Does nothing if variable is inexistent.

Parameters:
\texttt{name} The name of the variable.

\texttt{void Edgematrix::set_constraint (string from, string to, time_duration value)}

Sets the constraint of the edge between \texttt{from} and \texttt{to} to \texttt{value}.

\texttt{void Edgematrix::set_distance (string from, string to, time_duration value)}

Sets the distance of the edge between \texttt{from} and \texttt{to} to \texttt{value}.

\texttt{void Edgematrix::set_dominator (string from, string to, string name)}

Sets the dominating node of the edge between \texttt{from} and \texttt{to} to \texttt{name}.

The documentation for this class was generated from the following files:

- \texttt{Edgematrix.h}
C.3 MainData Class

#include <MainData.h>

Public Member Functions

- **MainData** (double r_val)
- void **read_input** (std::string servicetimes_manufacturer_filename, std::string standplan_filename, std::string servicetimes_KLM_filename, std::string standmatrix_fuelling_filename, std::string task_actor_assignment_filename, std::string delay_filename)
- void **setOutput** (std::string outdir)
- void **print_data** ()
- void **processInput** ()

Static Public Member Functions

- static std::string **give_callsign** (std::string s) [static]
- static std::string **give_type** (std::string s) [static]

Detailed Description

brief The class MainData contains all data and coordinates everything.

Constructor & Destructor Documentation

**MainData::MainData** (double r_val)

Constructor

Parameters:

- r_val The speed with which the decoupler is to converge

Member Function Documentation

**static std::string MainData::give_callsign** (std::string s) [static]

Gives back the callsign of a node name

**static std::string MainData::give_type** (std::string s) [static]

Gives back the type of activity of a node ('fuel' for example)
void MainData::print_data ()
Outputs the contents of all persistent data to the output stream.

void MainData::processInput ()
Process the input files.

void MainData::read_input (std::string servicetimes_manufacturer_filename, std::string standplan_filename, std::string servicetimes_KLM_filename, std::string standmatrix_fuelling_filename, std::string task_actor_assignment_filename, std::string delay_filename)
Reads input from the specified files

Parameters:
• servicetimes_manufacturer_filename
• standplan_filename
• servicetimes_KLM_filename
• standmatrix_fuelling_filename
• delay_filename

void MainData::setOutput (std::string outdir)
Sets the output directory

Parameters:
outdir The output directory

The documentation for this class was generated from the following files:
• MainData.h
• MainData.cc

C.4 Output Class

#include <Output.h>

Static Public Member Functions
• static void write_solution_space_CSV (std::string pathname, std::string fname, boost::shared_ptr< stp > s)
• static void write_EST_CSV (std::string pathname, std::string fname, boost::shared_ptr< stp > s)
- 62 -

CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING

- static void write_solution_space_EST_CSV (std::string pathname, std::string fname, boost::shared_ptr<stp> s)

Static Protected Member Functions

static void write (std::string pathname, std::string file, const std::ostringstream &outp)

Detailed Description

Output class to write the results to file.

Member Function Documentation

static void Output::write (std::string pathname, std::string file, const std::ostringstream &outp) [static, protected]
Writes the actual file.
Parameters:
- path The path we output to.
- file The filename
- outp The ostringstream containing the data to write.

static void Output::write_EST_CSV (std::string pathname, std::string fname, boost::shared_ptr<stp> s) [static]
Writes the solution space for the given stp to file., in text CSV (as per RFC 4180).
Parameters:
- path The path we output to.
- fname The name of the output file.
- s Reference to the stp that is outputted.

static void Output::write_solution_space_EST_CSV (std::string pathname, std::string fname, boost::shared_ptr<stp> s) [static]
Writes the solution space for the given stp to file., in text CSV (as per RFC 4180).
Parameters:
- path The path we output to.
- fname The name of the output file.
- s Reference to the stp that is outputted.
CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING

 Writes a file with the callsign, the start of the interval, the EST start, the EST end and the end of the interval, separated by semicolons.

 Parameters:

- \textit{pathname}
- \textit{fname}s

 The documentation for this class was generated from the following file:

- Output.h

\section*{C.5 ProcessInput Class}

\begin{verbatim}
#include <ProcessInput.h>
\end{verbatim}

\textbf{Static Public Member Functions}

- static bool \textit{read_ACTypesManufacturer} (std::string &filename, std::vector< ACType > &ACTypes)
- static bool \textit{read_ACTypesAirport} (std::string &filename, std::vector< ACType > &ACTypes, std::map< std::string, std::vector< Precedence > > &precedence_constraints)
- static bool \textit{read_StandPlan} (std::string &filename, std::vector< Flight > &standplan)
- static bool \textit{read_StandMatrix} (std::string &filename, std::map< std::string, boost::posix_time::time_duration > &standmatrix)
- static bool \textit{read_TaskActorAssignment} (std::string &filename, std::vector< Actor > &task_actor_assignments)
- static bool \textit{read_Delays} (std::string &filename, std::vector< Delay > &delays)

\textbf{Classes}

- struct \textit{malformed_line}

\textbf{Detailed Description}

brief Class that contains all input routines. All routines are static so the class doesn’t need to be instantiated.

\textbf{Member Function Documentation}

\begin{verbatim}
static bool ProcessInput::read_ACTypesAirport (std::string &filename, std::vector< ACType > &ACTypes, std::map< std::string, std::vector< Precedence > > &precedence_constraints) [static]
\end{verbatim}

Reads in ACTypes from the specified file. Whenever something goes wrong, the program exits.
Parameters:

static bool ProcessInput::read_ACTypesManufacturer (std::string & filename, std::vector< ACType > & ACTypes) [static]
Reads in ACTypes from the specified file. Whenever something goes wrong, the program exits.

Parameters:
filename The file to read from

static bool ProcessInput::read_Delays (std::string & filename, std::vector< Delay > & delays) [static]
Reads the delays from the specified file. Whenever something goes wrong, the program exits.

Parameters:
filename The file to read from.

Returns:
true if succesful.

static bool ProcessInput::read_StandMatrix (std::string & filename, std::map< std::string, boost::posix_time::time_duration > & standmatrix) [static]
Reads the standmatrix from the specified file. Whenever something goes wrong, the program exits.

Parameters:
filename The file to read from

static bool ProcessInput::read_StandPlan (std::string & filename, std::vector< Flight > & standplan) [static]
Reads the standplan from the specified file. Whenever something goes wrong, the program exits.

Parameters:
filename The file to read from

static bool ProcessInput::read_TaskActorAssignment (std::string & filename, std::vector< Actor > & task_actor_assignments) [static]
Reads the task_actor_assignment from the specified file. Whenever something goes wrong, the program exits.

Parameters:
filename The file to read from
The documentation for this class was generated from the following files:
- ProcessInput.h
- ProcessInput.cc

C.6 STP Class

Class to solve STP's. Solver class for STPs.
#include <stp.h>

Public Member Functions
- stp ()
- stp (const stp &s, const string_set &znames)
- void addVariable (string var)
- void addVariable (string var, ptime time)
- bool assignTimeToVariable (string var, ptime time)
- bool contains (string var) const
- bool exist_edge (string from, string to) const
- void setConstraint (string from, string to, time_duration value)
- bool tryConstraint (string from, string to, time_duration value)
- void setConstraint (string from, string to, time_duration min, time_duration max)
- void setPrecedenceConstraint (string from, string to)
- time_duration getC (const string from, const string to) const
- time_duration getD (const string from, const string to) const
- string dominatedBy (const string from, const string to) const
- void removeConstraint (string from, string to)
- void removeVariable (string name)
- shared_ptr< list< string > > > getVariableList () const
- shared_ptr< list< pair< string, string > > > > getEdgeList () const
- ptime get_zero_time () const
- void set_zero_time (ptime time)
- void reset ()
- bool minimize ()
- shared_ptr< vector< pair< string, ptime > > > > give_EST ()
- shared_ptr< vector< pair< string, ptime > > > > give_LST ()
- ptime give_EST (string name)
- ptime give_LST (string name)
- shared_ptr< vector< boost::tuple< string, ptime, ptime > > > > give_solution_space ()
Static Public Member Functions

static string give_callsign (const string &s)
Gives back the callsign of a node name.

static string give_type (const string &s)
Gives back the type of activity of a node ('fuel' for example).

Detailed Description
Class to solve STP's. Solver class for STPs. Translated from the java original written by Pieter Buzing.

Constructor & Destructor Documentation
stp::stp ()
Default constructor. Automatically creates the zero time node (X0).

stp::stp (const stp & s, const string_set & znames)
Creates a sub-stp with only the specified nodes.
Parameters:
• s The stp that is being z-partitioned.
• znames The names of all nodes which should be in the z-partition.

Member Function Documentation

void stp::addVariable (string var, ptime time)
Adds a new TP variable to the STP. Does nothing if var already exists. Uses assignTimeToVariable to set the time.
Parameters:
var The name of the new TP variable.
time The time of the TP variable.

void stp::addVariable (string var)
Adds a new TP variable to the STP. Does nothing if var already exists.
Parameters:
var The name of the new TP variable.
bool stp::assignTimeToVariable (string var, ptime time)
Assigns time to variable var. If assigning value would make the STP inconsistent, nothing is done and false is returned. What is essentially done is that the constraint from X0->var is set to time and the constraint var->X0 is set to -time, thus fixing value of the TP.

Parameters:
var Name of the TP variable.
time The time of the TP variable.

Returns:
True if value can be assigned to var, false if value is not in the allowed range.

bool stp::contains (string var) const

Returns:
True if var is in the STP, false otherwise

string stp::dominatedBy (const string from, const string to) const

Returns:
The name of the node which dominates the edge between from and to. Or an empty string if the edge is not dominated.

bool stp::exist_edge (string from, string to) const

Returns:
True if an edge between from and to exists, false otherwise.

ptime stp::get_zero_time () const
Gives the zero time of the STP.

time_duration stp::getC (const string from, const string to) const

Returns:
The constraint of the edge between from and to. (C value)

time_duration stp::getD (const string from, const string to) const

Returns:
The (minimized) distance of the edge between from and to. (D value)

shared_ptr< list< pair< string, string > > > stp::getEdgeList () const
Gives back all edges from this STP.

shared_ptr< list< string > > stp::getVariableList () const
CAED D3: A TURNAROUND MODEL AND PROTOTYPE BASED ON DECOUPLING

Gives back all variables from this STP.

```cpp
ptime stp::give_EST (string name)
Gives back the EST of the specified variable.
```

```cpp
shared_ptr< vector< pair< string, ptime > > > stp::give_EST ()
Gives back the EST solution of the STP
```

```cpp
ptime stp::give_LST (string name)
Gives back the LST of the specified variable.
```

```cpp
shared_ptr< vector< pair< string, ptime > > > stp::give_LST ()
Gives back the LST solution of the STP
```

```cpp
boost::tuple< std::string, ptime, ptime > stp::give_solution_space (string name)
Gives back the allowed range if the specified variable.
```

```cpp
shared_ptr< vector< boost::tuple< string, ptime, ptime > > > stp::give_solution_space ()
Gives back the allowed range of all variables in this STP.
```

```cpp
void stp::make_triangles ()
Makes (temporary) datastructure which contains all triangles in the STP. The datastructure is invalidated by creating or deleting node or edges in the graph. Having this datastructure will significantly speed up minimizing.
```

```cpp
bool stp::minimize ()
Minimizes the STP.
```

```cpp
void stp::removeConstraint (string from, string to)
Removes the edge between from and to and the opposing edge between to and from. Does nothing if the edge(s) doesn't exist (as per the behaviour of Edgematrix).
```

```cpp
void stp::removeVariable (string name)
Removes the specified variable and all its incoming and outgoing edges. Does nothing if the node doesn't exist (as per the behaviour of Edgematrix).
```

```cpp
void stp::reset ()
Resets all edges.
```
void stp::set_zero_time (ptime time)
Sets the zero time of the STP.

void stp::setConstraint (string from, string to, time_duration min, time_duration max)
Sets the constraints between from and to. Constraints are set so that to must occur between [min, max] after from. This will replace any existing previous constraints. Note that this call is equivalent with setConstraint( from, to, max) followed by setConstraint( to, from, -min ).

void stp::setConstraint (string from, string to, time_duration value)
Sets a constraint value. Does not automatically propagate. This method can be used for both tightening and loosening of constraints.

void stp::setPrecedenceConstraint (string from, string to)
Set the constraints between from and to such that from must precede to. This will replace any existing previous constraints. Note that this call is equivalent with setConstraint( from, to, +infinite) followed by setConstraint( to, from, 0 ).

bool stp::tryConstraint (string from, string to, time_duration value)
Tries to set the constraint to the new value and calls minimize(). If assigning this value makes the STP inconsistent, false is returned and the STP is reverted to its original state.

Returns:
true iff the constraint was set, false otherwise.

The documentation for this class was generated from the following files:

- stp.h
- stp.cc