Modelling railway dispatching actions in switching max-plus linear systems

Student: Dirk van der Meer
E-mail: D.J.vanderMeer@Student.TUDelft.NL
Student number: 1158511

Date: October 22th 2008

TU Delft
Technische Universiteit Delft

Front page: No infra constraints are active as train 1959 awaits its scheduled departure time at Den Haag HS in the winter sun of December 15th 2007.

Modelling railway dispatching actions in switching max-plus linear systems

Master thesis

Student:
Dirk van der Meer
D.J.vanderMeer@student.TUDelft.NL
Student number: 1158511

University:
Delft University of Technology
Faculty of Civil Engineering, Transport & Planning section

Graduation Committee:
Prof.dr.ing. I.A. Hansen
Faculty of Civil Engineering, Transport & Planning section
I.A.Hansen@tudelft.nl

Dr. R.M.P. Goverde
Faculty of Civil Engineering, Transport & Planning section
R.M.P.Goverde@tudelft.nl

Dr.ir. T.J.J. van den Boom
Delft Center for Systems and Control
A.J.J.vandenBoom@tudelft.nl

Preface

This thesis is the result of a graduation project to obtain a Master's degree in Civil Engineering at the Delft University of Technology. This graduation project has been carried out at the department of Transport and Planning of the faculty of Civil Engineering and Geosciences.

Interested in railways and railway operations as long as I can remember, I used the flexibility of the Transport and Planning Master's program to lay a strong emphasis on public transport and rail guided transportation systems in my personal study program. During the course 'Railway Traffic Management' I found out about the use of max-plus algebra in railway operations research, which, as a combination of railway timetabling, civil engineering and mathematics, attracted my interest. After a detour via the structural aspects of a railway system in the form of an internship at HTM, I decided to return to the research on max-plus algebra in railway operations by choosing this as a subject for my graduation thesis.

I would like to thank Rob Goverde for his great contribution as a daily supervisor. From the beginning, he understood my way of working and explained just the right theory and backgrounds. Thanks to Ton van den Boom, the second member of my graduation committee, for his refreshing comments resulting from his theoretical view. I finally thank Prof. Hansen for his positive and constructive, yet critical and systematic way of supervising the project.

Although not of substantial significance for the contents of the project, my roommates of room 4.42 contributed in various ways to a perfect working environment. So thanks Koenis, for sharing your plans to take over the world, Luis Costa Costa 'Tacos y Burritos' Garcia Lopez Viale, for bringing about the Mexican spirit, Ivo, for your philosophical view on several topics, Frank, for your pleasant company and for showing us what it looks like to work hard, and last but certainly not least Ollemollie, for (almost literally) always being there for some good chitchat. Thanks to the people from room 4.39, in particular Martijn and Nicole, for letting the event 'go to lunch' occur at exactly 12.15 without delay every day.

Dirk van der Meer
Delft, October 2008

Abstract

In order to reduce delays in a railway system and to prevent them from propagating through the network, a dispatcher can apply dispatching actions, for example by changing the location of a planned overtaking, or cancelling a train run. A dispatching action is defined as an intervention in the rail traffic system with the purpose of solving a conflict between train runs, thereby aimed at reducing delays and their propagation. This research project is aimed at the development of a tool that helps the dispatcher when deciding which dispatching actions to apply by quickly evaluating their effectiveness on network level using algorithms based on max-plus algebra and timed event graphs.

Max-plus algebra is a mathematical framework capable of modelling railway systems with all their interdependencies in space and time, which is used to develop powerful tools to analyse railway operations. When the max-plus model is extended with the ability to adapt in accordance with dispatching actions, it can be used to predict the impact of those dispatching actions. For the approach followed in this thesis, max-plus models will be expressed graphically using timed event graphs, since their transparency enables modelling a railway network in a clear and unambiguous way.
In order to study the stability and performance of periodic (e.g. hourly) timetables, the existing max-plus models and the corresponding timed event graphs are cyclic. However, modelling dispatching actions in such a cyclic environment is difficult to implement, which is why an acyclic model has been used in this research project. Some methods for implementing this in practice have been proposed.
The level of detail of the model was chosen to correspond with its emphasis on analysis and control rather than accurate simulation of railway operations. Therefore, block signals on the open track are not modelled, and stations and junctions are modelled as timetable points acting as ‘black boxes’. All necessary time separations between trains are modelled as constraints between arrivals at and/or departures from these timetable points. Application to a test case showed that the results of this approach are accurate enough to enable a network wide evaluation of dispatching actions.

The timed event graph representation has been used to investigate the implications of dispatching actions on the model. These implications have been described as so called ‘construction rules’ describing the changes in the timed event graph necessary to represent the corresponding dispatching action, and following immediately from the implications of the dispatching action on the model. The following dispatching actions have been implemented as algorithms using these construction rules:

- Change the sequence order of trains,
- Short-turn a train (i.e. at a station before its terminal station),
- Postpone the departure or arrival of trains at stations with conflicting interlocking routes.

In practice, more different dispatching actions can be evaluated with the theory developed in this thesis. Changing the sequence order of trains can for instance be used to move a scheduled overtaking to another station, which is technically a different dispatching action. Furthermore, the described construction rules can be used to develop other algorithms, for instance for cancelling train runs, introducing new train runs, etc.

In order to use the algorithms for finding effective dispatching actions for a given network and a set of initial delays, they can be implemented in an optimization algorithm. In this project, a greedy approach has been used. To maximize their effectiveness, order changes of trains have to

be evaluated in combination with the appropriate postponements of arrivals and departures of the involved trains when conflicting interlocking routes are present in the network. The total passenger delay in the network served as the objective function, thereby including the negative effects for passengers when train trips are partly cancelled due to short turning of delayed trains. A test case with two delay scenarios showed that the generated dispatching actions are plausible and consistent.

Dutch abstract

Nederlandse samenvatting

Om vertragingen in een spoorwegnetwerk te verminderen en vertragingsvoortplanting te voorkomen kan een treindienstleider bijstuurmaatregelen, zoals het verplaatsen van een geplande inhaallocatie of het opheffen van een trein, toepassen. Een bijstuurmaatregel is gedefinieerd als een interventie in de treindienst met als doel het voorkomen van conflicterende treinbewegingen, waarbij de vertraging en de vertragingsvoortplanting gereduceerd worden. Dit project is gericht op de ontwikkeling van een hulpmiddel voor de treindienstleider voor het evalueren van de effectiviteit van bijstuurmaatregelen, gebruik makend van max-plus algebra en timed event graphs.

Max-plus algebra is een wiskundig raamwerk waarin spoorwegnetwerken met al hun afhankelijkheden in ruimte en tijd gemodelleerd kunnen worden. Krachtige systemen voor het analyseren van het railverkeer maken hiervan gebruik. Wanneer een max-plus model wordt uitgebreid met de mogelijkheid om te kunnen veranderen wanneer een bijstuurmaatregel wordt doorgevoerd, kan het worden gebruikt om de effectiviteit van zulke maatregelen te evalueren. In dit project zullen de max-plus modellen als timed event graphs worden gerepresenteerd, omdat de transparantie van deze grafische weergave het mogelijk maakt om een spoorwegsysteem helder en duidelijk te modelleren.
Voor het bestuderen van de stabiliteit en prestaties van periodieke (uur)dienstregelingen wordt gebruik gemaakt van periodieke max-plus modellen en timed event graphs. De implementatie van bijstuurmaatregelen in een dergelijk cyclisch model is echter problematisch. In dit onderzoek is daarom een acyclisch model gebruikt, waarbij tevens enkele methodes om dit in de praktijk te implementeren zijn voorgesteld.
Omdat het model in dit onderzoek voornamelijk wordt gebruikt voor het analyseren en regelen van spoorwegsystemen is een nauwkeurige simulatie hiervan minder relevant. Het detailniveau van het model is hierop aangepast, zodat blokseinen langs de vrije baan niet zijn gemodelleerd en stations en aansluitingen zijn gemodelleerd als dienstregelingpunten. De beveiliging die noodzakelijke afstanden tussen treinen in het netwerk garandeert wordt gemodelleerd in de vorm van randvoorwaarden in de tijd tussen de aankomsten en/of vertrekken bij deze dienstregelingpunten. Toepassing van het model in een testscenario heeft aangetoond dat dit nauwkeurig genoeg is om de effectiviteit van bijstuurmaatregelen op netwerkniveau te evalueren.

Voor het onderzoeken van de implicaties van bijstuurmaatregelen op het model is de timed event graph representatie gebruikt. De implicaties zijn omschreven als zogenoemde ‘constructieregels’ die de noodzakelijke wijzigingen in de timed event graph voor het representeren van de bijbehorende bijstuurmaatregel weergeven. De constructieregels volgen dus direct uit de implicaties van bijstuurmaatregelen op het model. De volgende bijstuurmaatregelen zijn, gebruik makend van deze constructieregels, als algoritmes geïmplementeerd:

- Het verwisselen van de volgorde van treinen,
- Het kort keren van treinen (waarmee wordt bedoeld: het laten keren van een trein voordat hij op zijn eindpunt is aangekomen),
- Het uitstellen van een aankomst of vertrek op een station met conflicterende rijwegen.

In de praktijk kunnen nog meer bijstuurmaatregelen met behulp van de in dit project ontwikkelde theorie worden geëvalueerd. Door het verwisselen van de volgorde tussen treinen kan bijvoorbeeld een geplande inhaallocatie worden verplaatst. Bovendien kunnen de omschreven constructieregels worden gebruikt voor het ontwikkelen van andere algoritmes, waarmee

bijstuurmaatregelen als het opheffen van treinen, het inleggen van treinen, enz. kunnen worden gemodelleerd.

Om de hierboven beschreven algoritmes te kunnen gebruiken voor het vinden van effectieve bijstuurmaatregelen voor een gegeven netwerk en een verzameling van initiële vertragingen, kunnen ze worden geïmplementeerd in een optimalisatiealgoritme. In dit project is hiervoor een greedy benadering gebruikt. Om hun effectiviteit te maximaliseren moeten volgordewisselingen hierbij altijd in combinatie met het uitstellen van aankomsten en vertrekken worden beoordeeld wanneer sprake is van conflicterende rijwegen in het netwerk. De totale reizigersvertraging is gebruikt als doelfunctie, zodat het negatieve effect voor reizigers wanneer treinen gedeeltelijk worden opgeheven vanwege een korte kering wordt meegenomen in de beoordeling. De ontwikkelde algoritmes hebben in een testscenario plausibele en consistente resultaten opgeleverd.

Table of contents

1 Introduction

Railway systems will always be subject to smaller and bigger disturbances causing train delays. Particularly in dense networks with a lot of train traffic, such as the Dutch railway network, delayed trains will cause conflicts by getting in the way of other trains, or they will affect connecting train services, thereby propagating the delay through the network and spreading the delay to other trains.
In order to reduce delays and prevent them from propagating, a dispatcher can apply dispatching actions, for example by changing the location of a planned overtaking, or cancelling a train run. This is not an easy task because the locations and time instances of occurring conflicts are not known in advance, and it is difficult to predict the impact of dispatching actions.
When delays occur in the network, train dispatchers need an answer to the question: how to get the train service back to the scheduled situation as quickly as possible? This research project is aimed at the development of a tool that helps the dispatcher answer this question quickly by evaluating the effectiveness of dispatching actions.

Max-plus algebra is a mathematical framework capable of modelling railway systems with all their interdependencies in space and time, which can be used to develop powerful tools to analyse railway operations. The computer application PETER, which can calculate the delay propagation in a railway network, as well as the robustness and stability of an hourly timetable, is an example of this [7]. When the max-plus model is extended with the ability to adapt in accordance with dispatching actions, it can be used to predict the impact of those dispatching actions. This can be done using switching max-plus systems [3]. A change of the railway system due to a dispatching action being carried out can have many implications on such a max-plus system. These implications are the main subject of this thesis.

For the approach followed in this thesis, max-plus models will be expressed graphically using timed event graphs. The transparency of such graphical models enables modelling a railway network in a clear and unambiguous way. The level of detail of the model is chosen to correspond with its emphasis on analysis and control rather than accurate simulation of railway operations. The timed event graph representation will be used to investigate the implications of dispatching actions on the model. These implications will be implemented in algorithms able to evaluate the effectiveness of dispatching actions.

The outline of this thesis is as follows: In the next chapter, the research subject and the main goal of this thesis will be described in detail. Chapter 3 contains a more detailed description of the dispatching actions and the railway system in which they can be carried out. Furthermore, a brief review of the literature on online dispatching systems will be given. The concept of timed event graphs is explained in chapter 4. In chapter 5 the used data structure for storing the timed event graph in the computer memory will be explained. Chapter 6 contains the main result of this thesis, an explanation of the algorithms developed to adapt the model according to dispatching actions. Furthermore, a greedy optimization approach for finding a set of effective dispatching actions based on a given set of delays is presented here. The developed algorithms are tested in a case study, which is described in chapter 7. Chapter 8 contains the conclusions of this thesis, along with an outlook on the applicability and recommendations for future research.

2 Problem analysis and research objective

2.1 Max-plus algebra in railways

The structure of a railway network contains many interdependencies between train movements. These interdependencies consist of passenger transfer connections between trains, constraints caused by the infrastructure and the rolling stock circulation, etc. Such a system can be effectively modelled using a *scheduled max-plus linear system* [7].
In order to create a max-plus model of a railway system, all train runs are broken down into series of processes and events. Furthermore, all constraints that have to hold for the events are expressed in max-plus algebra. This yields a max-plus linear system in which the event times and the timetable are expressed in vectors and the constraints caused by the structure of the network and the timetable are captured in the *system matrix*. The max-plus model can be used to calculate several characteristics of the system behaviour such as the stability of the timetable, the propagation of delays, etc. The max-plus model can be translated to a graphical representation in the form of a timed event graph, which is exactly equivalent with the max-plus model it represents.

2.2 Dispatching actions in max-plus algebra

In case of delays conflicts can occur between trains hindering each other. Traffic control and rescheduling actions can be carried out by dispatchers [12] in order to prevent delays from propagating to other trains on the network.
Some typical examples of rescheduling actions are:

- Switching the sequence order of two trains (i.e. letting another train depart before a delayed train).
- Change the location of a planned overtaking.

Control actions refer to traffic management actions in which the original train schedule is maintained, such as:

- Cancel a transfer connection between two trains.

Max-plus linear systems are based on a timetable with a basic hourly pattern. This means that the order of the trains on the tracks, the transfer connections, the train routes, etc., are exactly the same each hour. Railway traffic management actions as depicted above cannot be modelled within such a framework.
A railway system in which dispatching actions can take place can be considered as a system that can operate in different *modes*. In max-plus systems for railways each mode refers to a set of train sequences, transfer connections, etc. A system that can switch between different modes of operation can be modelled using *switching max-plus linear systems* [3]. In a switching max-plus linear system each mode of operation is represented by a different system matrix. With this extension, the model can be used to calculate the effectiveness of different dispatching actions with regard to the settlement of delays, or more general, to some objective function. In the end, an optimization algorithm can be implemented in order to find an effective (combination of) traffic management actions in case of delays.

2.3 Problem description

To model the switching structure of the system in case of a dispatching action, different system matrices for each given dispatching action are needed. The generation of all different system matrices in advance would lead to a combinatorial explosion of the amount of data if all hypothetical combinations of possible actions have to be taken into account. This means that new system matrices have to be generated 'on the fly' in the calculation process when the effectiveness of a dispatching action has to be calculated.
In most cases a discrete choice for a control action or rescheduling measure leads to implied changes in the structure of the model. For example: when at some station the order of two trains is changed, a sequence of changes along the railway line is implied until the location where the order is restored to the situation as scheduled. The implications of railway dispatching actions on the structure of the system matrix are the main subject of this project. This leads to the following problem question:

"How can a max-plus model be used to calculate the impact of dispatching actions in railway systems?"

To solve this problem, a research objective has to be achieved. In this project, the research objective is formulated as follows:

"To create a description of all relevant dispatching actions that can be modelled in switching max-plus linear systems, and to obtain an inventory of the implications of dispatching actions on the model which can be used to produce an algorithm to calculate the effectiveness of dispatching actions."

3 Dispatching actions in rail traffic management

3.1 Introduction

Before the implications on the max-plus model when dispatching actions are applied, the main subject of this thesis, are discussed, an overview and description of the most important dispatching actions and the railway system will be given in this chapter.
The outline is as follows. The next section describes the railway system as modelled in this thesis. All terms used in this subject referring to railways are defined here. Then, the enormous amount of possibly imaginable dispatching actions will be divided into two main groups: traffic control actions, which will be described in section 3.3, and rescheduling actions, to which section 3.4 is dedicated. In section 3.5, an overview of the literature on recent research on railway operations modelling and online rescheduling will be presented. Section 3.6 contains a brief conclusion of this chapter.

3.2 The railway system: definitions

In this section, the type of railway systems modelled in this thesis will be defined. Definitions are important for two reasons:

- A model cannot be built if the system to be modelled is not defined clearly.
- In the different fields of railway operation and research, some terms are defined slightly different, which can lead to confusion.

3.2.1 Train route and interlocking route

In Figure 3.1 a part of an example train trip via stations A and B is shown. In the station, the train can go to different tracks using switches and/or crossings. The exact route of the train through the station is defined as the *interlocking route*. Switches and crossings are assumed to occur only at stations (including their yards) and at junctions, so a track connecting the stations and/or junctions with each other is assumed to have no switches and crossings. Such is track is called the *open track*.
The term *train route* refers to the route of the train on network level (note the difference with *interlocking route*). A train *run* is the part of a train trip between to subsequent stations, and the *running time* is the time it takes to complete a train run. The term train *trip* refers to the journey of a train from its starting station to its terminal station (i.e. along its entire route through the network). The *dwell time* is the time between the arrival and departure of a train in the station (i.e. how long the train is standing still).

3.2.2 Timetable path and train lines

The trains modelled in this project are assumed to run according to a timetable created in advance. The path through time and space that each train is scheduled to follow is called the *timetable path*. The timetable is assumed to be *periodic*, which means that train trips with the same route, speed and stopping pattern occur repeatedly during the day, separated by fixed time intervals. An hourly timetable is obtained when these intervals amount exactly one hour. In this project, hourly timetables are used to develop and demonstrate the algorithms, but although other period lengths are rare in reality, the theory can be easily adapted to any other period length. A

group of periodically occurring train trips with the same route on network level, as well as the same interlocking route, speed and stopping pattern is called a *train line*. An example of a train line is the intercity connection from Amsterdam to Brussels, running every hour, each trip connecting the same stations. Note the difference with the term *railway line*, which refers to the rail infrastructure connecting two stations.
At the terminal station of a train trip, the rolling stock of that train will usually turn and perform a train trip in the opposite direction. This is called a *turn*.

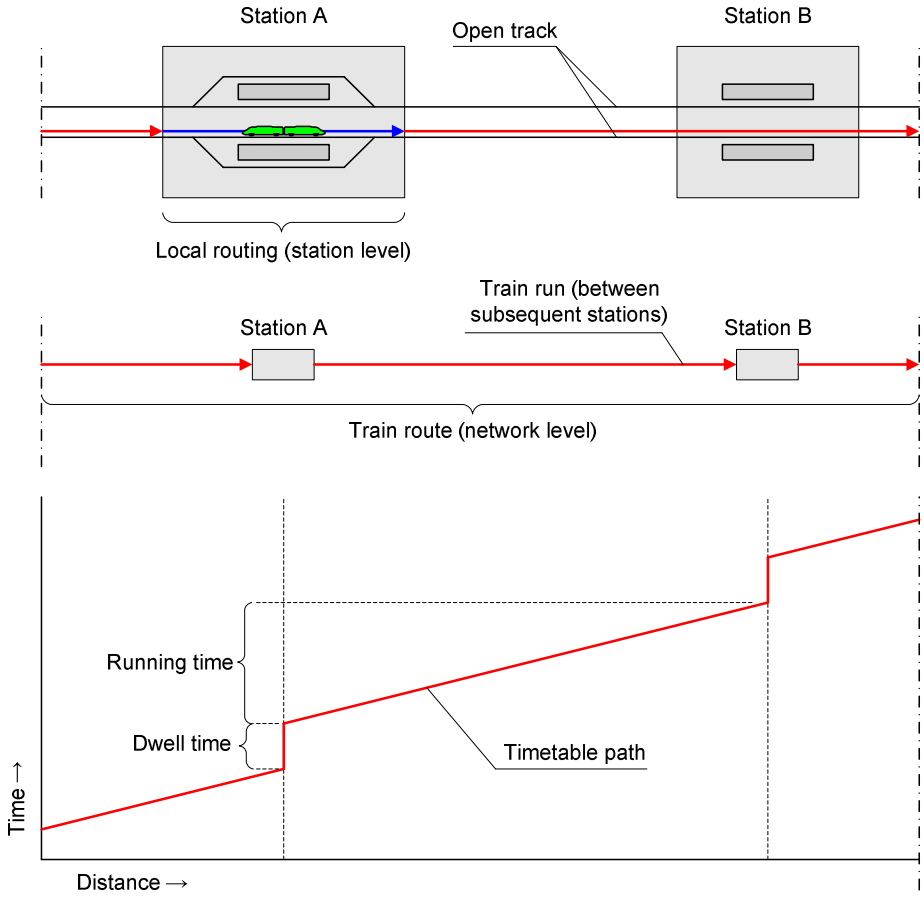


Figure 3.1 *Part of an example train trip with definitions.*

3.2.3 The dispatcher and dispatching actions

In this thesis, the train traffic is assumed to be centrally controlled by a dispatcher. The term *dispatcher* as used in this thesis refers to the employee who supervises and controls the train movements, as described in Pachl (2004). This is in accordance with the way most railway lines in Europe are operated.
Although a timetable is originally intended to be conflict-free, conflicts may arise between trains in case of delays. Using dispatching actions, the dispatcher has to solve the occurred conflicts, while minimizing the delay and the propagation of delays. A dispatching action is defined as follows:

A dispatching action is an intervention in the rail traffic system with the purpose of solving a conflict between train runs, thereby aimed at reducing delays and their propagation.

Dispatching actions can be divided in a group of traffic control actions and a group of rescheduling actions, which will be described subsequently in the next two sections.

3.3 Traffic control actions

Traffic control actions are small interventions in the rail traffic system. They are used to prevent delays from propagating through the network and to make the rail traffic more fluently. In this thesis, traffic control actions are defined as follows:

Dispatching actions in which the line routes and sequence orders of trains are not changed.

Traffic control actions have the following characteristics:

- A traffic control action can be carried out last-minute in most cases.
- The original schedule is not necessarily maintained.
- The relative positions of the trains in the network (i.e. their route and order) are maintained.

The most common examples of traffic control actions are:

1) Cancel a transfer connection between two trains.
2) Increase the running time.
3) Change the duration of a stop.
4) Modification of the stopping pattern (e.g. a stopping train becomes an express train and vice versa).
5) Assign another open track to the train (e.g. in a four-track section).
6) Assign another platform track to the train in a station.
7) Change the interlocking route of a train in a station.

Note that the route of the train on network level (the *line* route) is not changed by the 7th traffic control action. Only the interlocking route through the station is concerned here.

3.4 Rescheduling actions

Most conflicts cannot be solved effectively by using traffic control actions only. Rescheduling actions provide many more possibilities for solving those conflicts. In this thesis, rescheduling actions are defined as follows:

Dispatching actions in which the line routes and/or sequence orders of trains are changed.

The following characteristics can be assigned to rescheduling actions:

- In most cases, more time in advance is needed to enable a rescheduling action than in the case of a traffic control action (e.g. a transfer connection can be broken at all times, while an overtaking has to be planned before the trains actually have arrived at the station).
- By definition, the original schedule is not maintained.
- The relative positions of the trains in the network (i.e. their route and order) are changed.

The rescheduling actions will be divided into three groups:

- Rescheduling actions changing the order of trains.
- Rescheduling actions changing the line routes of trains.
- Change to an emergency timetable.

3.4.1 Rescheduling actions changing the order of trains

This type of dispatching actions is often carried out in case of bigger delays. By changing the order of trains, a delayed train can be prevented from propagating its delay to on-time trains hindered by it. Examples are:

1) Change the sequence order of trains by planning an overtaking.
2) Change the location of a planned overtaking.
3) Cancel an entire train run.
4) Insert an on-time train and cancel the delayed train.
5) Insert an extra train.
6) Short-turn a train (i.e. let a delayed train turn at a station before its terminal station).
7) Change the order of arrivals and/or departures at a station.
8) Cancel a planned coupling of two trains and let them continue as two separated trains.
9) Couple two trains that run on the same route.
10) Let a delayed train continue in the path of the next train of the same type (e.g. exactly half an hour later).

3.4.2 Rescheduling actions changing the line routes of trains

Changing the line route of a train on network level can be a convenient rescheduling action in case of big disruptions (e.g. closure of a track due to rolling stock or infrastructure failure). The line route of a train is changed by the following rescheduling actions:

11) Redirect a train to a different route through the network (while maintaining the starting and terminal stations).
12) Redirect a train to a starting and/or terminal station outside its regular route.

3.4.3 Change to an emergency timetable

In case of big disruptions in the rail network, more complex rescheduling actions are applied. These rescheduling actions are often part of a scenario that is available at the traffic control centre. Therefore, these rescheduling actions can be seen as a change to an emergency timetable:

13) Make an inventory of trains that can continue running or that can be re-routed, cancel all other trains.
14) Use measure 13, and improve the service level by inserting extra trains.
15) Go back and forth with one train through the bottleneck caused by the disruption.
16) Let trains with the same direction pass the bottleneck as a group in order to increase the capacity of the bottleneck.

Some (combinations of) rescheduling actions can cause the rolling stock circulation to become disrupted. In such cases, the rolling stock circulation has to be adjusted to the rescheduled timetable. This can be done using the aforementioned rescheduling actions (e.g. cancelling a train run for which no rolling stock is available). Additionally, the following complex rescheduling actions to create a feasible rolling stock circulation are:

17) Cancel an entire train line during a disruption, and re-insert the trains while assigning the correct rolling stock units to every train run after the disruption.
18) Exchange train units between trains at turn-around stations.
19) Couple or de-couple train units to a train in order to get the correct amount of seats.
20) Insert an empty run in order to create a feasible circulation of rolling stock.

In practice, a lot of different combinations of traffic control actions and rescheduling actions are applied in order to optimize the rail traffic and make the process more smoothly.

3.5 Review on online dispatching

After the discussion of different categories of dispatching actions in the previous sections, the question remains: which (combinations of) dispatching actions have to be applied in order to solve conflicts in an optimal way? During the last decades, research has been carried out to find methods for answering this question in practice. In this section, some results with relevance for this project will be reviewed.

3.5.1 Modelling rail traffic using blocking time diagrams

A straightforward way of modelling the rail traffic makes use of blocking time diagrams, which has been extensively described in the German literature. On lines where train separation in block distance is used, the track is divided in block sections which may be exclusively occupied by one train [12]. A blocking time diagram is a graphical representation of the blocking times of block sections due to train traffic, enabling easy detection of conflicts and their solutions. Note that this technique can also be used in combination with modern safety and signalling systems using moving blocks, such as ECTS level 3.

A method which makes use of a detailed calculation of blocking times for detecting conflicts is proposed by Jacobs [9]. Conflicts detected by a detailed calculation of running times in which specific characteristics of each train in the network are taken into account, are solved by locally rescheduling conflicting trains in such a way that the propagated delays are as small as possible. In case of different priorities of trains, the train with the lowest priority is postponed.

DisKon [8] is a system which is currently in development in Germany. DisKon is able to detect and solve conflicts on a railway track. The traffic situation and the calculated dispatching actions are graphically displayed using blocking time diagrams. The system is aimed at handling one railway line, so a focus on network level would mean that the network has to be split up in several railway lines. The dispatching actions used for solving conflicts are: changing the route of a train, changing a scheduled overtaking station, planning extra stops, cancelling a train or cancelling transfer connections.

3.5.2 Creating a model using graphs

Modelling the rail traffic system as a system of processes and events can yield powerful algorithms for real-time conflict detection and resolution. Such a system can be visualized using an alternative graph or a Petri net, or more specifically, a timed event graph.

D'Ariano et al. [5] use an alternative graph model in which sets of alternative arcs represent the possibilities for rescheduling the train traffic operations. From the graph, boundary conditions for train movements can be derived, which can then be used for conflict detection and resolution. A branch-and-bound algorithm is used to calculate an optimal solution to solve conflicts. Furthermore, speed profiles of the trains in the model are adjusted according to actual signal aspects to make the conflict detection and resolution more accurate. The system works on the level of a dispatching area (i.e. the area that is controlled by one dispatcher).

Another description of the alternative graph model can be found in Mazzarello et al. [11], where the alternative graph model is used to predict and solve conflicts in a case study at the Dutch Schiphol railway line and tested in a pilot for making the train traffic on the railway lines near Lage Zwaluwe more fluently. The latter is done by communicating speed advices to train engineers. However, the sequence order of trains on the railway lines remains the same, so no specific modelling system as discussed in this thesis is needed to represent this.

3.5.3 Methods using integer programming

Aside from rail traffic models using blocking time diagrams or graphs, a great variety of other ways for solving conflicts in real time has been proposed. Some different strategies will be shortly reviewed in this section.
Törnquist [13] proposes a heuristic method for solving conflicts by assigning new timetable paths (i.e. time-distance paths) to all trains, independent on which tracks were originally assigned in the timetable. In some cases, the order of trains is changed as well, as delayed trains can be allowed to stand back (i.e. wait on a siding track) for a number of other trains. A case study indicated that a planning horizon of 60 minutes is sufficient for achieving solutions which are good on the longer-term.
Adenso-Díaz et al. [1] present a method in which the traffic system is modelled using integer programming in order to calculate optimal rescheduling actions in case of disruptions affecting the train circulation. The rescheduling actions considered here are: 1) cancelling the affected service, in which case the effects on following services of the same unit have to be calculated, 2) sending another unit to carry out the service. A heuristic procedure for reducing the solutions space is presented.

3.5.4 Max-plus models with control possibilities

Max-plus algebra is a powerful tool to calculate delay propagation. However, only few publications are known in which max-plus algebra is used for calculating the impact of railway rescheduling measures on network level. Van den Boom & De Schutter [3] describe a way of modelling discrete event systems in which control actions are possible using a switching max-plus linear system. They allow the system to operate in different modes, whereby each mode can refer to a certain set of train orders and transfer connections. Goverde [6] uses max-plus algebra to calculate the consequences of breaking a transfer connection with regard to waiting times for passengers.

3.6 Conclusion

A dispatching action is an intervention in the rail traffic system with the purpose of solving a conflict between train runs, thereby aimed at reducing delays and their propagation. Dispatching actions can be divided into two groups: *traffic control* actions are dispatching actions in which the line routes and sequence orders of trains are maintained, while by *rescheduling actions* the line routes and sequence orders of trains are changed.
During the last decades, research has been carried out to find ways for improving dispatching procedures during disruptions in railway operations. Mainly, models using blocking time theory, graph theory or linear programming are used to find optimal dispatching actions.
Although most research on online rescheduling aims at systems that can operate in one dispatching area, one line of even one station, some delays and dispatching actions can have network-wide effects. However, no online tool able to calculate the network-wide effectiveness of dispatching actions in railway systems has been developed yet.

4 Timed event graphs for railway operations

4.1 Introduction

How can railway operations, with all their interdependencies and constraints in space and time, be modelled? The scheduled max-plus linear system, or its graphical equivalent in the form of a timed event graph, possesses properties making it a powerful tool to do this. In this chapter, the concept of timed event graphs will be explained using an example of two trains crossing each other at a station.
The outline is as follows: In section 4.2 will be explained why timed event graphs are used instead of the mathematical representation using max-plus equations. In section 4.3 will be explained how railway operations are broken down into processes and events, which is used in section 4.4 to explain the basic concept of timed event graphs. How a timed event graph can be used to model a railway system is shown in section 4.5. As stated in section 2.2, the periodic model of a railway system cannot be used to represent dispatching actions, which will be shown in section 4.6. This section will deal with a modification of the timed event graph such that the model is no longer periodic. This enables the implementation of dispatching actions. Section 4.7 is dedicated to the limitations of the presented model. Section 4.8 contains the conclusion of this chapter.

4.2 Why use timed event graphs?

A timed event graph and a max-plus model are equivalent in the sense that a timed event graph can be translated directly into a max-plus model, and vice versa. Consequently, two approaches for implementing dispatching actions in the model, the goal of this project, are possible. One approach is to represent the railway system as a timed event graph. When doing so, dispatching actions can be implemented by changing the timed event graph accordingly. The other approach involves representing the railway system as a system of max-plus equations. Dispatching actions are in that case implemented by changing the affected elements of the max-plus matrix accordingly.
For the development of the algorithms for this research project, the representation using timed event graphs is used, for the following reasons:

- Visualising the system using timed event graphs gives insight in the behaviour of the system.
- Implications of dispatching actions on the model are translated more easily to timed event graphs then to systems of max-plus equations.
- Algorithms handling timed event graphs are transparent in the sense that it is clear to see what they are doing and why they are doing that.

It should be remarked that the choice for modelling the system as a timed event graph does not necessarily have implications for later compatibility with algorithms based on pure max-plus theory. Since a max-plus matrix of a railway system is sparse, it is often represented as a list of the nonzero elements. Such a list principally matches the arclist representation used for storing timed event graphs in this project, which will be presented in the subsequent chapters. Therefore the timed event graph representation can merely be regarded as a temporary stage in the algorithm development process. The final result, a model with the ability to implement dispatching actions, remains the same and can be translated back to max-plus algebra when necessary. A way of representing dispatching actions as a switching max-plus linear system is shown in section 6.6.

4.3 Railway operations broken down into processes and events

In Figure 4.1, a small example network with two trains is shown. Train 1 is scheduled to go from station A to station C. After the departure of train 1, train 2 is scheduled to go to station B. In this example only the shown part of the railway network is considered and the possible processes and events before the departures of trains 1 and 2 are omitted. The interlocking routes of trains 1 and 2 are crossing each other when leaving station 1 and 2.

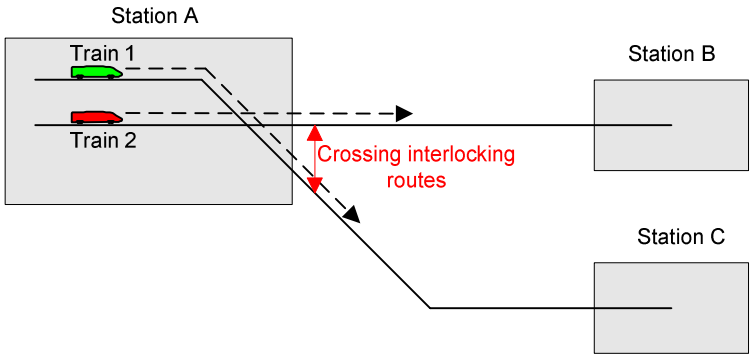


Figure 4.1 *Small example network with crossing train movements.*

In order to understand the concept of timed event graphs, one has to realize that railway operations can be broken down into a set of *processes p*, which take a certain amount of time to be completed, and a set of *events x*, which form the beginning or the end of a process.
To illustrate this, three processes and four events from the example of Figure 4.1 are identified in Table 4.1 and Table 4.2 respectively. Note that in reality this example can be part of a much bigger network of events and processes, but for the sake of simplicity only three processes and four events will be taken into account.

Table 4.1 *Three processes from the example.*

Process	Description
1	*Train 1 runs from A to C*
2	*Train 2 waits until train 1 has left the station.*
3	*Train 2 runs from A to B*

Table 4.2 *Four event from the example.*

Event number	Description
1	*Departure of train 1 from A*
2	*Arrival of train 1 at C*
3	*Departure of train 2 from A*
4	*Arrival of train 2 at B*

All identified processes and events are interrelated with each other. In the next section will be explained how this can be modelled using graph theory.

4.4 Basic concepts of timed event graphs

In this section, the example of section 4.3 will be translated to a timed event graph, while simultaneously the basic concepts of timed event graphs will be explained.

4.4.1 Places and transitions representing processes and events

Timed event graphs are bipartite directed graphs which form a subclass of Petri nets. This means that a timed event graph consists of a set of places and a set of transitions, which are connected by arcs. Graphically, the places are represented by circles and the transitions by rectangles, or bars. In a timed event graph, each place has exactly one incoming arc from a transition and one outgoing arc to a transition. As a consequence, each place together with its incoming arc and its outgoing arc can be interpreted is an arc itself, connecting two transitions directly, as can be seen in Figure 4.2. Therefore, the example described in section 4.3 consists of four transitions $x_1 \ldots x_4$, connected by three arcs $t_1 \ldots t_3$.
When railway operations are modelled, the transitions in the graph represent *events*, and the arcs represent *processes* (see section 4.3 for an explanation of *events* and *processes*). Since processes are obviously time-consuming (e.g. running time between two stations), each arc r in the timed event graph has a minimum time delay t_r attached to it. In the literature, this time delay is often referred to as the *arc weight* or *holding time.*

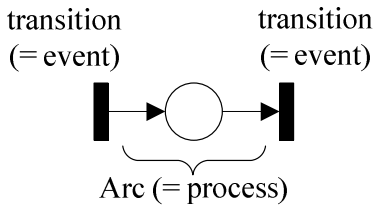


Figure 4.2 *Graphical representation of transitions and arcs.*

Recall the example from section 4.3. The processes and events identified in the example can be translated to sets of transitions and arcs. The resulting timed event graph is shown in Figure 4.3. Here the arc weight t_1 represents the running time from A to C, t_2 represents the waiting time for train 2 until train 1 has left the crossing and t_3 represents the running time from A to B.

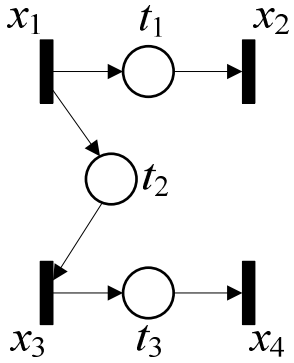


Figure 4.3 *Timed event graph of example network.*

How can one tell from a timed event graph which processes are active, and which are not? In other words: what is the actual state of the system? This is represented by the *marking* of the arcs, which will be explained in the next section.

4.4.2 Markings representing the actual state of the system

The marking of an arc is represented by drawing μ_r dots (tokens) in the circle which stands for place p_r. A token in a place represents an active process. When modelling of railway operations, a token in place p_r means that a train is actually performing process p_r at the represented time instant. In Figure 4.3 no tokens are present at all, which means that none of the three drawn processes is active. This corresponds to the situation in Figure 4.1, where no train has departed yet. In Figure 4.4 the same timed event graph is shown, but in this case the arcs 1 and 2 are marked. This means that process 1 and process 2 are active, which represents train 1 running from A to C, and train 2 waiting for the crossing to get clear.

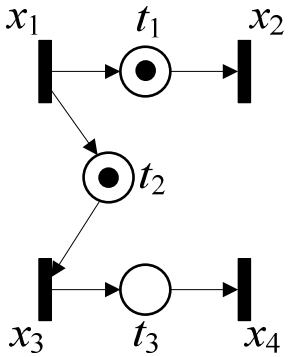


Figure 4.4 *Timed event graph with marked arcs.*

Firing rule

After the departure of train 1 from station A, train 2 has to wait long enough for the crossing to get clear. To model this, a correct timing and sequence of the events is guaranteed by the following two step *firing rule*, which each event has to obey:

i. *A transition x is* enabled *if each incoming place contains a token and the associated holding times have elapsed.*

ii. *A* firing *of an enabled transition x removes one token from each incoming place and adds one token to each outgoing place.*

The firing rule enables tokens to move over the places, thereby reflecting the dynamic behaviour of the system. An example of an event firing is shown in Figure 4.5.

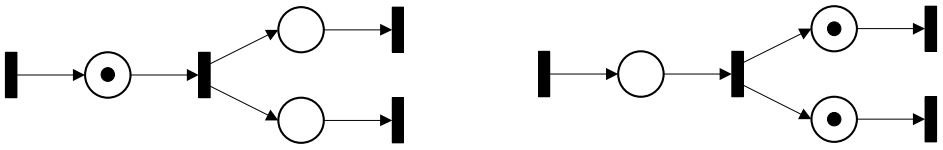


Figure 4.5 *Timed event graph after firing event x_2.*

The basic concepts described above can be used to model entire railway networks with all their interdependencies in time. In the next section will be discussed how all these interdependencies can be included in the model.

4.5 Modelling a railway network

A timed event graph is constructed in three phases, which reflect the different types of dependencies existing between the events in a railway network. In this section will be explained how all dependencies between train movements can be modelled using a timed event graph.

4.5.1 Modelling all train lines

In the first stage of constructing a timed event graph, the train lines themselves are constructed. Obviously, a train cannot arrive at a station before it has departed from the preceding station. This is modelled by a string of arcs, connecting all subsequent events a train line consists of. The arc weights between two stations reflect the minimum running time (i.e. the shortest time in which the train can cover the distance between the stations, not the *scheduled* running time). Arcs running from an arrival to a departure event reflect the dwell time. An example of this is shown in Figure 4.6, where the timed event graph of a train line from station A to station C, with an intermediate stop in station B is shown. When a train line is running f times per hour, the described procedure is repeated f times.

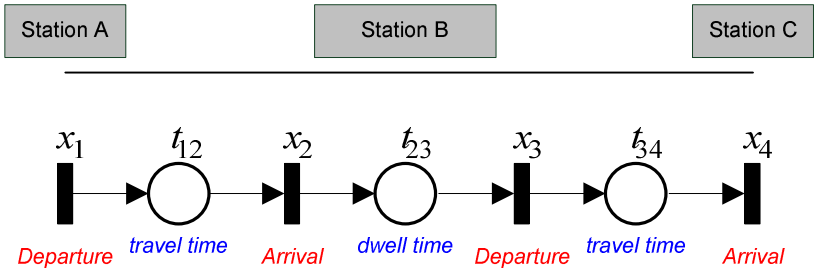


Figure 4.6 *Timed event graph of an example train line.*

4.5.2 Modelling infrastructure constraints

All trains sharing the same infrastructure are protected from collisions by signalling and safety systems [12]. These systems make sure that a minimum space separation between trains using the same infrastructure is assured. The space separations are translated into time separations so that they can be modelled using a timed event graph. In the second stage of the construction of a timed event graph, these time separations, called infrastructure constraints, are modelled. Two types of infrastructure constraints exist: headway constraints and hindrance constraints.
Before headway constraints and hindrance constraints are described, the modelling of stations as timetable points will be explained.

Modelling with timetable points

Each location where trains can interact with each other via headway or hindrance constraints is defined as a timetable point. This involves the following parts of a railway network:

- all stations,
- all junctions.

Block signals, only contributing to the space separation of trains on the open track, are not modelled. Since this model is constructed with the purpose of studying the arrival, departure and through times at stations, the train movements on the open track are insignificant. Feasibility at the open track and a fixed ordering of the trains on the open track and at timetable points are ensured by considering the time separations at the beginning and at the end of the open track [7].

All timetable points are modelled as 'black boxes', meaning that the topology of the tracks and the platforms in the station itself is not considered. An example is shown in Figure 4.7, where a schematized double track line with a platform between the tracks is shown (one railway track is represented by one line). The translation of this station to a timetable point is also depicted. Points where trains can enter or leave the station are called in/out-points (or short: IO-points).

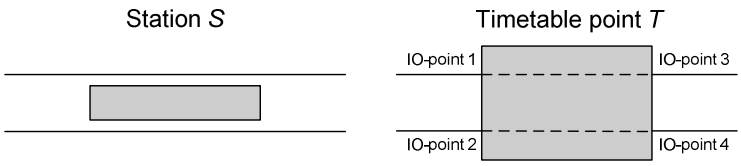


Figure 4.7 *Translation of a station S into timetable point T.*

Headway constraints

Headway constraints ensure the time spacing between trains running behind each other on the same railway track. They are modelled by arcs between subsequent departure or arrival events at the same IO-point of a station. Hence the following definition:

A headway constraint arc connects two events occurring at the same IO-point, thereby originating at the preceding event and ending at the successive event according to the schedule.

An example is shown in Figure 4.8, where three train lines are running from station A to station B (the black arcs). In order to model the required time spacing between the trains, the red arcs have been added to ensure the minimal headway times *h*.

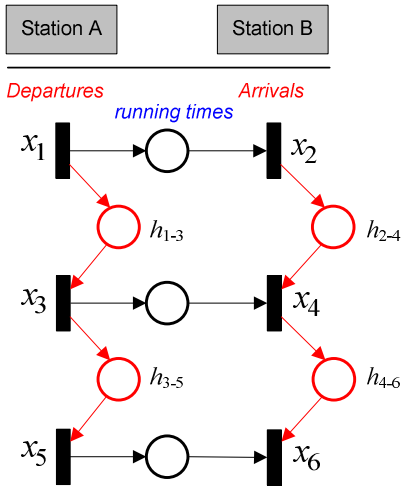


Figure 4.8 *Example of headway constraints at two stations. The headway constraints are coloured red and marked by their headway times h.*

In an hourly timetable the headway constraints have to form cycles in order to allow the same events to continue being fired in the next periods. An example of such a cycle, including its marking (see section 4.4.2 for an explanation of *marking*), is shown in Figure 4.9. Note that with the shown marking, x_2 would have to occur before x_5.

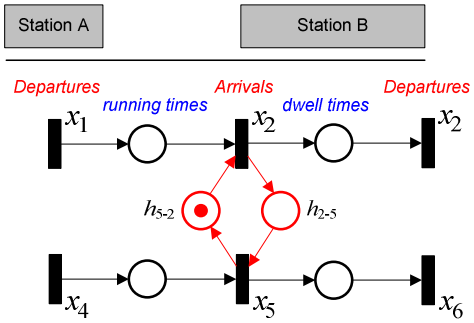


Figure 4.9 *Headway constraints at the entrance of station B for an hourly timetable with two lines.*

Hindrance constraints

Whereas headway constraints ensure time separation between events using the same IO-points of a station, hindrance constraints are used to model conflicts between events using *different* IO-points. Interlocking routes using different IO-points can conflict with each other when they use the same infrastructure on their route through the timetable point (e.g. when crossing each other). An example is shown in Figure 4.10. Although the trains use different IO-points, a time separation has to be included in the model because the trains cannot depart at the same time. Note that for modelling this, it has to be known whether the interlocking routs cross before of after the platform, since this yields different hindrance constraints.

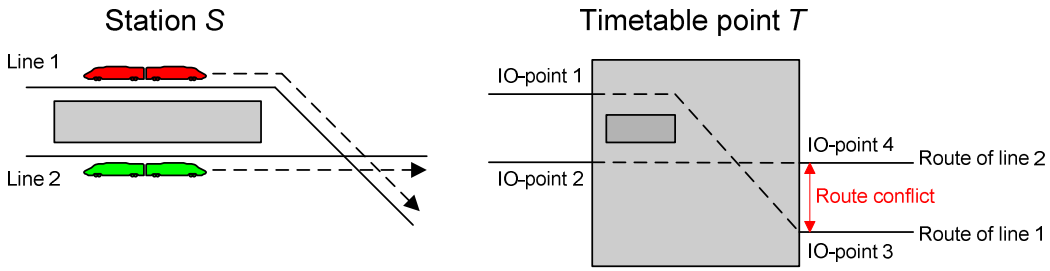


Figure 4.10 *Hindrance conflict between departures of train lines.*

As the interlocking route of trains through timetable points is not modelled in this project, it is assumed that every train line has its own fixed routing through the timetable point. Therefore it can be stated that hindrance conflicts occur between conflicting train *lines*. Hence, the following definition describes hindrance conflicts as implemented in this project:

A hindrance conflict arc connects two events of different, conflicting train lines, occurring at different IO-points but at the same timetable point, thereby originating at the preceding event and ending at the successive event according to the schedule.

Often, one train line conflicts with two or more other train lines. To assure a correct modelling of all time separations caused by hindrance conflicts, each event should be connected by hindrance arcs to the events corresponding to *all* train lines it is conflicting with. This means that if a movement of train line *L* causes hindrance to *n* other train lines, *n* outgoing hindrance arcs have to be added to the timed event graph for the corresponding event of this train line. An example is shown in Figure 4.11, where three train lines are scheduled to depart in the order 1, 2, 3. The departure of line 1 is hindering the two other lines by crossing them while departing from station *S*, which is why two hindrance constraint arcs originate from the departure event of line 1. Note that in a periodic timetable, this implies that the departure of line 1 has *n* incoming hindrance arcs as well.

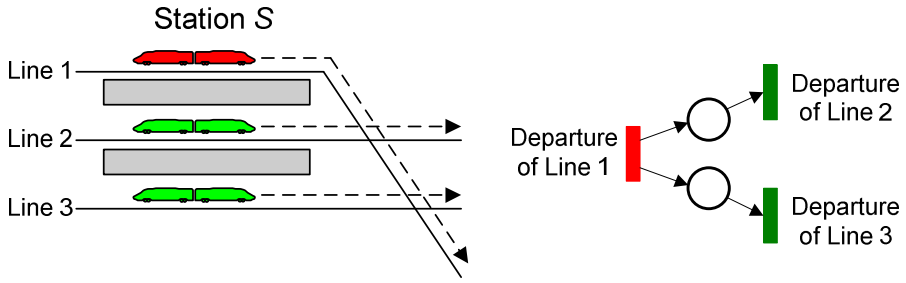


Figure 4.11 *Example of hindrance arcs. Line 1 is hindering two other train lines. The corresponding hindrance arcs are shown on the right.*

Since hindrance constraint arcs have to run in the direction of the scheduled order of events, this order has to be known for all events at a timetable point when generating hindrance arcs. The timetable vector *d* cannot be used for this purpose, since the scheduled event times only represent the correct order of all events in the *original* timetable. The scheduled order of events may be changed by dispatching actions, e.g. when trains overtake each other, making the scheduled order of events in the timed event graph differ from the order in the timetable. This means that the order of events occurring at the same timetable point has to be stored separately, which is implemented by using a linked list. For each event *x*, the preceding and the successive event are stored in such a linked list. The system with linked lists is chosen for its low complexity and its high flexibility, since only a few entries of the linked list have to be changed when the scheduled order of events is changed.
Note that the presence of hindrance arcs in the timed event graph does not necessarily imply that the corresponding hindrance conflict will actually occur in the railway system. Usually, the time separation between conflicting train movements is accounted for in the timetable and the trains are scheduled in such a way that no actual conflicts occur. However, in case of delays and/or dispatching actions the hindrance constraint arcs can become significant and conflicts can occur.

Redundant constraints are maintained
When including hindrance constraints in the model as described above, some infrastructure constraints in the model can become redundant. A constraint is called redundant when the time separation it represents is already assured by other constraints, an example of which is shown in Figure 4.12. In this example, events x_1 and x_3 necessarily have a time separation of 6 minutes via the constraints connecting event x_2. This makes the constraint between x_1 and x_3 redundant. In a real railway system, this situation would for example occur when two train movements in the same direction are separated by a third, crossing train movement. The headway constraint between the two trains in the same direction is likely to become redundant since sufficient time separation is already assured by the crossing train movement.

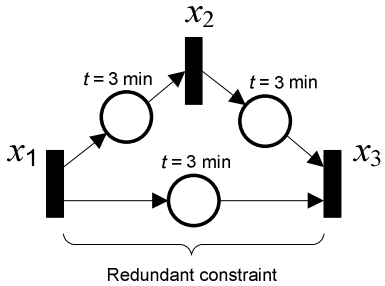


Figure 4.12 *Example of a redundant infrastructure constraint.*

Redundant headway arcs are maintained in the timed event graph, although they are irrelevant for the dynamic behaviour of the model. In this way, hindrance arcs and headway arcs are treated as two separate groups without affecting each other's topology in the timed event graph. This has been done for the following reasons:

- Hindrance and headway arcs can be generated without knowledge about the topology of the timed event graph. This yields quick and simple algorithms for implementing dispatching actions.
- The topology of headway arcs in the timed event graph contains information about the order in which the trains are running along the track, which is used by some algorithms.
- When events are postponed, redundant headway constraints can become active again. When redundant headway arcs are kept in the timed event graph, the correct time separation between trains is always assured, also when dispatching actions changing the timed event graph are carried out.

4.5.3 Modelling synchronization constraints

Two types of synchronization constraints are distinguished: passenger transfers and rolling stock connections. In the literature, this is sometimes referred to as soft synchronization constraints (as they can be broken without implications for the schedule) and hard synchronization constraints respectively (since a connecting train trip can simply not depart if its rolling stock has not arrived yet).

Passenger transfers

At many stations, trains are waiting for each other to enable passenger transfers. In the 3rd phase of the construction of a timed event graph, these connections are modelled. This is done by connecting the relevant events with an arc, where the arc weight is the transfer time. An example is shown in Figure 4.13, where the train at platform 2 is the feeding train, while at platform 1 the connecting train is scheduled. The red arc, representing the synchronization constraint, ensures that the departure of the connecting line at platform 1 can take place only after the arrival of the feeder line, after the transfer time $t_{\text{synchronization}}$ has elapsed.

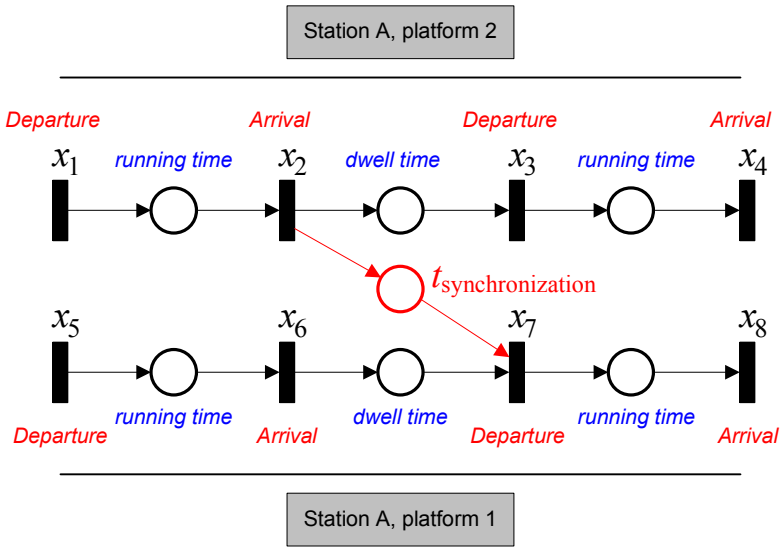


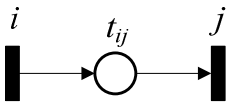
Figure 4.13 *Example of a transfer constraint between two trains.*

Rolling stock circulation
The rolling stock circulation is modelled in this stage in the same manner. In the case of rolling stock circulation, the departure of a train line has to wait for the arrival of the previous train line for which the same rolling stock is used. In most cases this will correspond to a turn at the end of a train line.

4.5.4 Determining the initial marking

In order to get a working timed event graph, the marking has to be set correctly. Recall from section 4.4.2 that the marking represents the present state of the system. When constructing a timed event graph, the state of the system at $t = 0$ (the beginning of a period) is calculated. The marking of the arcs at the reference time instant $t = 0$ is called the *initial marking*. The initial marking of an arc (i, j) can be calculated with the following formula, where $\lceil x \rceil$ denotes the least integer not smaller than $x \in \mathbb{R}$ (i.e. x is rounded up):

$$\mu_{ij} = \left\lceil \frac{d_i + t_{ij} - d_j}{T} \right\rceil$$



in which:
μ_{ij} = initial marking of arc (i, j),
t_{ij} = scheduled process time of arc (i, j),
d_j = scheduled time of event j,
d_i = scheduled time of event i,
T = cycle time (60 minutes in case of an hourly timetable).

4.6 Timed event graph without periods

4.6.1 Algorithms become complex when periodicity is maintained

Modelling a railway system using timed event graphs as described above is suitable for studying periodic timetables (e.g. an hourly timetable) but this becomes problematic when dispatching actions have to be modelled. In particular, changing the sequence order such that the k-th occurrence of an event is postponed after occurrence k+1 of another event is difficult, which will be illustrated in the following example:
Consider Figure 4.14, where postponing event i_1 to the next period would lead to the situation that event 1 is not present in period 1 at all, while period 2 contains the same event twice. This would be no problem if the scheduled order of events remains fixed, but events in period 2 may precede events in period 1 when the order is changed, or more general, occurrence k+1 of an event may precede the k-th occurrence of such an event. This is problematic since arcs have to run to previous periods, which is difficult to implement. The problem is solved in this project by creating an acyclic model out of a periodic model, which will be explained in the next section.

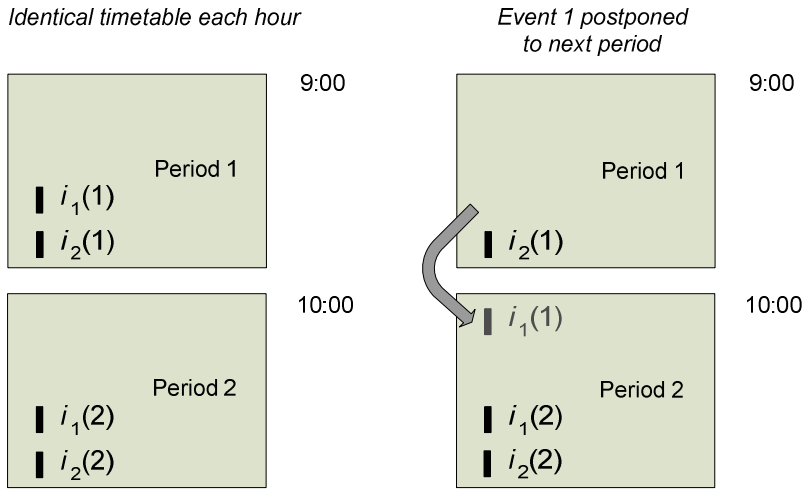


Figure 4.14 *Postponing an event to the next period, leading to the situation that an event precedes an event from the previous period.*

4.6.2 Unfolding periodic events during the day

The problems concerning the implementation of dispatching actions within a periodic environment have been solved by giving each periodic event a unique number for each period over the entire day. This can be described in two steps, shown in Figure 4.15. As input, the periodic model is used. In the first step, each event is stored separately for each period, implying that each event is identified with its event number *i* and its period number *k*. This step can be visualised as 'unfolding' the periodic events to unique events (*i*, *k*). The purpose of the second step is to prevent the algorithms from getting unnecessarily complicated. In this step, each event (*i*, *k*) gets a unique event number *i*.

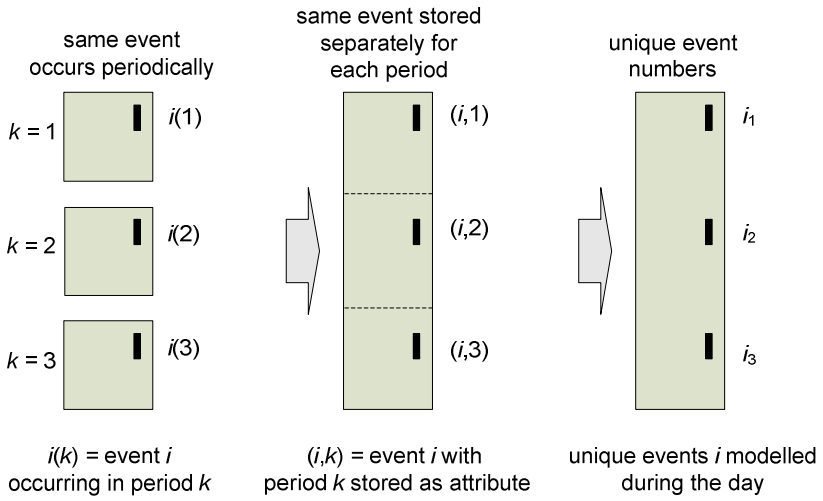


Figure 4.15 *Unfolding periodic events to obtain unique events during the day, shown as an example of three periods.*

The model used to perform the calculations can thus be regarded as one big period. However, the periodicity of the hourly timetable is still present in the model, as events of the same train line

occur periodically. This property is important for the generation of hindrance constraints, as will be described in section 5.3.4.
Using a timed event graph without periods has several effects on the properties of the model, and thus on the algorithms. The implementation of a system without periods is motivated by the following advantages of such a system:

- Postponing an event to the next hour can be implemented without ambiguous implications such as arcs running to a preceding period.
- Changes in the timetable during the day (e.g. slight differences in running times and stopping patterns of the first and last train trips of the day) can be modelled easily.
- The algorithms become less complex due to neglecting periods.
- The graph becomes acyclic for which fast critical path algorithms exist.

On the other hand, discarding periodicity comes with a negative aspect too. A periodic max-plus model has structural properties which can be exploited by advanced optimization algorithms. An example is the critical path, from which the minimal cycle time can be derived. This property can be used as a first indication of the stability of certain dispatching actions. However, advanced optimization algorithms are a subject beyond the scope of this research project, which is why periodicity can be discarded without implications for this research project.

4.6.3 Methods for implementing the system without periods

A non-periodic (acyclic) model as described in the previous section will be used as input for the algorithms presented in this thesis. For an implementation using a periodic model, several methods to create an acyclic system can be adopted. Three of them will be explained briefly in this section. The first method is used in the remainder of this thesis.

1. Create a non-periodic model in advance
This method involves defining each event in advance, after which the timed event graph is generated. After generation, this timed event graph can be used to carry out the required calculations. This method is suitable for creating small hypothetical networks and for offline use, and is therefore used in this project to develop and test the algorithms. Since the non-periodic model is created in advance, a non-periodic list of events is considered as input in chapter 5 when generating the timed event graph.

2. Create a non-periodic model from a cyclic timed event graph
When a cyclic timed event graph is available, the steps described in section 4.6.2 as the ‘unfolding’ process, can be adopted to create a non-periodic model for an entire day. This method is useful to make offline calculations with dispatching actions if a cyclic timed event graph is already available, for instance when dispatching actions are to be implemented in software like PETER [7].

3. Create a non-periodic model for real time use
A way to implement the non-periodic model in a real time environment is depicted in Figure 4.16. Within the planning horizon, dispatching actions have to be evaluated in a model without periods, for the reasons described in section 4.6.2. The model is constructed from one period which is stored separately. During the day, periods are added to the model in order to maintain the desired planning horizon. At the same time, the periods laying in the past can be deleted from the model for the obvious reason that dispatching actions cannot be carried out for events that occurred in the past, and therefore do not need evaluation. Of course, the deleted parts of the model can be

stored separately for later analysis, as they contain the railway operations as actually carried out including the dispatching actions.

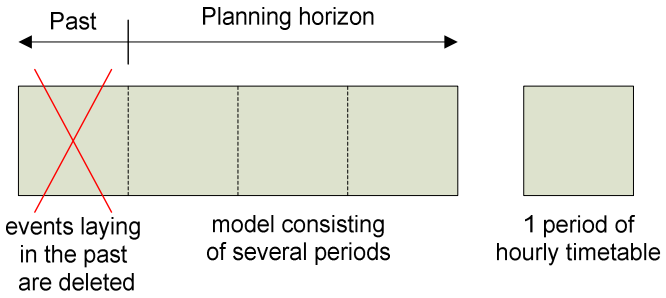


Figure 4.16 *Model consisting of several periods.*

4.7 Limitations of the model

Although the model described above allows a detailed representation of large scale railway networks with great computational power, it has some limitations too. The most important limitations of the model will be reviewed in this section.

Interlocking routes through stations are not modelled
By modelling all stations as timetable points, no possibility is left to include the routings through stations as a variable in the optimization process. The routings are regarded as a fixed property of a train line. The disadvantage of this is that dispatching actions regarding the interlocking routes cannot be modelled. However, optimizing the interlocking routes at bigger stations is very complex and adding this possibility to the optimization framework presented in this thesis is considered a subject beyond the main goal of this project.

Minimal running times are fixed
The minimal running times t for each train run are included as a fixed property of the corresponding arc (the arc weight) in the model. However, a fixed minimal running time is sometimes inconsistent with the real situation. Consider for example Figure 4.17, where the time-distance curve of a hindered train is shown. In the model, the waiting time for the signal is included via headway and hindrance constraints, but the additional time loss due to the acceleration of the train is not included.

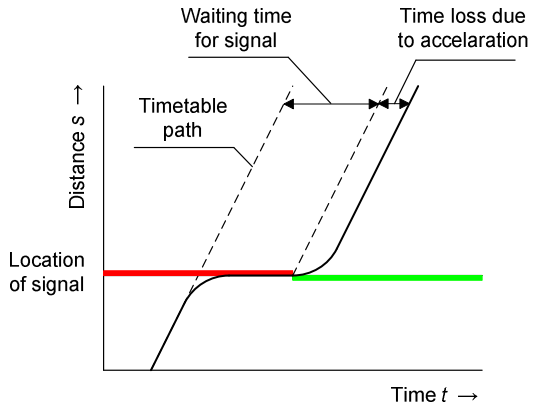


Figure 4.17 *Running time increase due to acceleration after waiting for a signal.*

This time loss depends on many factors, such as the rolling stock, weather, personal driving style of the train driver, whether the train came to a full stop or not, etc., and can only be included in the model with complex calculations of the train dynamics. A practical suggestion for a less complex implementation of this is to include an extra penalty for dispatching actions causing a through train to be hindered, such that this penalty accounts for the running time increase of this train while evaluating such a dispatching action.

Block or track occupation is not modelled explicitly

Modelling the headway constraints as described in this chapter leads to a clear and comprehensive model. However, a limitation of this way of modelling the time separation between trains is that the actual occupation of a bock by a train is not modelled explicitly. In case of delays, this can sometimes lead to the situation in the model that two trains seem to occupy the same block.

Although this looks like a big limitation of the model, the influence on the outcomes of the delay propagation algorithm (see section 5.4) is not necessarily big. Since the minimal headways are always assured via the appropriate constraints in the model, conflicts between trains in case of delays will become apparent correctly by using the delay propagation algorithm. Therefore, this limitation is no problem for the analysis and control purposes for which the model is used in this project. Moreover, the question actually relevant in this context is whether this limitation influences the outcome of the optimization algorithm when calculating the optimal combinations of dispatching actions. In the remainder of this thesis this is assumed highly unlikely. An investigation of the exact influence of this limitation on the algorithms ranges beyond the scope of this project. It is recommended for all future research on this subject to carefully consider the level of detail of the model in relation with the purpose it is used for, as has been done in this section.

A much more detailed model in which block and track occupation is modelled indeed, intended for calculating optimal dispatching actions on smaller parts of a railway network, such as a station or a dispatching area, is proposed in [5].

Deterministic running times are used

The last limitation discussed here concerns the fact that only deterministic (i.e. not stochastic) running times are used in this model. In reality, the minimal running time is never the same since it depends on many factors not accounted for in the model. Some of these factors can be controlled, for example by communicating advised speeds to train drivers, which can lead to the situation that the running times in reality are highly deterministic indeed. However, the introduction of a stochastic component of the running times in the model can lead to more realistic results when the running times realized in reality are partly stochastic.

4.8 Conclusion

Railway systems modelled as discrete event systems can be represented by systems of max-plus equations, which can be visualized as timed event graphs. For the development of algorithms in this thesis, the timed event graph representation will be used. This chapter explained the concept of timed event graphs. Railway operations are broken down into processes and events, which are represented in a timed event graph by arcs (places) and events (transitions) respectively. Subsequently, train lines are modelled as strings of processes and events.
In order to model the correct infrastructure constraints, timetable points are used to model stations as a black box where trains can enter and exit via IO-points. The infrastructure constraints are modelled as arcs between events of train lines using the same IO-points (in case of headway constraints) or using conflicting interlocking routes (in case of hindrance constraints).
Synchronization constraints ensure that trains in the model wait for transfer connections and that the rolling stock circulation is modelled correctly.
To evaluate the impact of dispatching actions, the periodicity of the model is abandoned, so one day is regarded as one big period.

5 Data structure for timed event graphs

5.1 Introduction

A suitable data structure is essential for the development of efficient algorithms. In this project, the data structure means: the way in which the timed event graph is stored in the computer. Since the system presented in this thesis is aimed at assisting the dispatcher while deciding which dispatching actions to carry out, calculations have to be performed quickly (i.e. the system has to operate in real time).
Recall the explanation of timed event graphs from chapter 4. The data structure has to be suitable for performing the following operations efficiently:

- Finding all arcs starting at event *i* (i.e. all arcs with tail *i*).
- Finding all arcs ending at event *i* (i.e. all arcs with head *i*).
- Deleting an arc from the timed event graph.
- Inserting an arc in the timed event graph.

An algorithm is stated to be efficient if the time it takes to perform one of the above operations is independent of the size of the model (i.e. the size of the modelled railway network). This can be achieved using a data structure with adjacency lists, which is why this data structure has been implemented in the algorithms presented in this thesis.

The goal of this chapter is to define and explain the used data structure and to introduce algorithms using this data structure to calculate the delay propagation in the railway network and the capacity consumption of a railway line. The outline is as follows: The variables making up the model in the computer memory are introduced in section 5.2, where the adjacency lists will be explained as well. When the data structure is defined, section 5.3 describes how the model is generated. Calculating the delay propagation in the network is important for the evaluation of dispatching actions. In section 5.4, a time efficient delay propagation algorithm using the described data structure is presented. Another useful parameter when evaluating dispatching actions is the capacity consumption. An algorithm for calculating this is presented in section 5.5. The conclusion can be found in section 5.6.

5.2 Variables for storing and editing the timed event graph

5.2.1 The matrix *Event*

The matrix *Event* contains characteristic information of each event present in the model. Since each event has to be stored separately, as explained in section 4.6, one row of the matrix *Event* represents one unique event during the day. Hence, an event occurring *n* times a day is present *n* times in the *Event* matrix. For each event *x*, a row of the matrix *Event* contains the following objects:

$Event(x) = (TN, LN, TTP, IO, type, P, N)$

where:
TN = unique train number of a train trip,
LN = line number,
TTP = timetable point,
IO = IO-point,

type = type of event,
P = preceding event scheduled at *TTP* (linked list),
N = next event scheduled at *TTP* (linked list).

The variable *type* in de matrix *Event* can have the following values:
1 = arrival,
2 = departure,
3 = arrival at the end of train run,
4 = departure at the start of a train run.

The purpose of the *Event* matrix is twofold. Most importantly, it is used by the algorithms presented in this thesis to identify characteristics of events relevant for applying dispatching actions correctly. The second purpose is to create output and give a meaning to it, for instance by relating delays to the corresponding train numbers.
Why storing a line number when each train has its unique train number anyway? Although an acyclic timed event graph is used, periodicity of the timetable is assumed (see section 4.6). Each train in the network can be assigned to a group of equivalent trains having the same properties. Each of those groups has a unique line number *LN*. This property is used by the algorithms to identify trains which have the same:

- interlocking route,
- stopping pattern,
- line route through the railway network.

The first property is used in the optimization process to derive hindrance constraints when dispatching actions are applied (see section 5.3.4 for the algorithm generating hindrance constraints). The second and third properties are used to calculate the delay of travellers who have to wait for the next train of the same type when their train is cancelled.

5.2.2 The arclist

The interdependencies between events are represented by arcs forming a timed event graph, as explained in chapter 4. This is implemented by storing all arcs in a list, called *A*. The arclist is stored as a matrix of which each row *r* represents one arc and contains the following objects:

$A(r) = (j, i, t, type)$

in which:
j = head event (i.e. the event to which the arc is running),
i = tail event (i.e. the event from which the arc originates),
t = holding time of the arc,
type = activity type.

The variable *type* can have the following values:
1 = running time arc,
2 = dwell time arc,
3 = headway arc,
4 = turning arc,
5 = hindrance arc,
6 = through arc.

Note that the marking of the arcs is not present here. Since the periodic events of the timetable are modelled and stored separately during one day ('unfolded', see section 4.6), the model has become acyclic. Consequently, markings loose their meaning, as they represent the possibility that arcs can run from one period to another.

5.2.3 The timetable vector *d*

The timetable vector *d* contains the times at which each event *x* is scheduled to occur. So:

d(*x*) = the scheduled event time of event *x*.

The event times are expressed in minutes starting from midnight, so for example 6:40 hours becomes $6 \cdot 60 + 40 = 400$ minutes. The timetable vector is regarded as input for the algorithms presented in this thesis.
The timetable vector is not changed when dispatching actions are carried out. The dispatching actions for which algorithms will be developed in the next chapter are aimed at changing the scheduled *order* of events. The underlying target is to minimize the delays in the network resulting from a given set of initial delays. By regarding the timetable vector as a constant, it can be used to calculate the delays of all events in the network as it corresponds to the original timetable.

5.2.4 Adjacency lists

The algorithms developed in this research project are capable of exploring the timed event graph without scanning the entire arclist for each operation. This capability is obtained by storing adjacency lists together with the arclist. For each event *i* two node-arc adjacency lists are generated. The first list contains all arcs *r* starting at event *i*, whereas the second list contains all arcs *k* ending at event *i*.
For an illustration of this, consider the example of a timed event graph shown in Figure 5.1. This graph consists of 6 events and 5 processes (the arcs). The arclist representing this is shown in Figure 5.2. For the sake of a clear example, the holding times and activity types of the arcs are omitted in this example. The adjacency lists belonging to this arclist are shown in the figure as well. Now suppose that an algorithm needs to 'know' which arcs have their ending (i.e. their head) at event 3. This can be found by looking at the 3rd row of the adjacency list for the heads. As can be seen in the figure, this row contains the numbers 2, 5 and 0, which means that the 2nd and the 5th arc in the arclist point to event 3. A zero in the adjacency list means that no (more) arcs start at the according event. The same can be done for the tails in the other adjacency list.

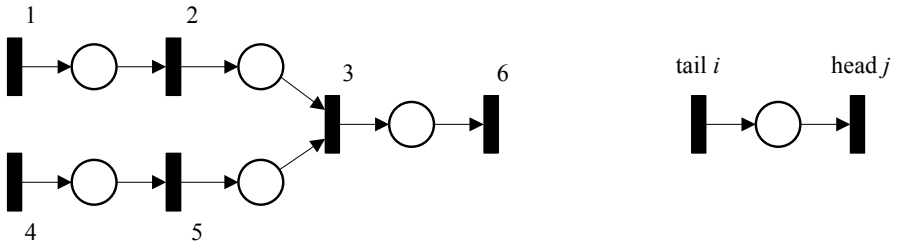


Figure 5.1 *Example of a timed event graph.*

heads j	**Adjacency lists** arcs $r \rightarrow$		
↓ 1.	0		
2.	1	0	
3.	2	5	0
4.	0		
5.	4	0	
6.	3	0	

tails i	**Adjacency lists** arcs $r \rightarrow$	
↓ 1.	1	0
2.	2	0
3.	3	0
4.	4	0
5.	5	0
6.	0	

	Arclist	
	heads j	tails i
1.	2	1
2.	3	2
3.	6	3
4.	5	4
5.	3	5

Figure 5.2 *Adjacency lists and arclist of example.*

As can be seen in the example, a lot of information about the structure of the graph can be retrieved without scanning the entire arclist, which would obviously be very time consuming when big graphs are concerned.
The adjacency lists are generated using Algorithm 5.1, which scans the entire arclist once, while storing each row number in the corresponding adjacency list. The adjacency lists are unsorted, hence the running time is linear and depends on the number of rows n of A, which is denoted by $O(n)$ in the algorithms literature. For very big arclists, this can become time consuming. This is why the adjacency lists are created before the optimization process starts.

Algorithm 5.1 (ARCLIST2ADJ)

Input:
A = list of all arcs
Output:
Adjtail = adjacency lists containing arcs starting at event *x*
Adjhead = adjacency lists containing arcs ending at event *x*

1. $m \leftarrow$ highest event number occurring in A ;
2. *Adjtail* ← zeros (*m*) ; *Adjhead* ← zeros (*m*) ;
3. **for** each row *r* of *A* **do**
4. *Adjtail* (i_r) ← *Adjtail* (i_r) ∪ *r* ; % store row number for tail event
5. *Adjhead* (j_r) ← *Adjhead* (j_r) ∪ *r* ; % store row number for head event
6. **return** *Adjtail*, *Adjhead*

Adjacency lists become useless if the arclist is modified in order to represent a dispatching action. This problem is solved by keeping the adjacency lists up to date when modifying the arclist, for which the algorithms presented in the next sections are used.

5.2.5 Deleting an arc

In order to keep the data structure working while deleting an arc from the timed event graph, the routine DELARC has been developed. A key idea behind the DELARC algorithm is that the remaining arcs are kept at their original locations in the arclist. This minimizes the number of changes required to keep the adjacency lists up to date, as the adjacency lists contain pointers to the *locations* of arcs in the arclist. Obviously, an empty row remains in the arclist after deleting an arc. The row numbers *r* of those empty rows are stored in the vector *Empty* so that these locations can be used to construct new arcs when necessary. The pseudocode of the DELARC routine is shown below.
Note that keeping the adjacency lists up to date when deleting an arc is just a matter of removing the pointers from the adjacency lists. This is implemented by moving the values from positions beyond the deleted value one position ahead in the vector, so that the last value remains zero. The

worst case running time of this algorithm thus depends on the size of the biggest adjacency list. However, when modelling railway networks, an event having more than around 10 successor events or preceding events will be extremely rare.

Algorithm 5.2 (DELARC)

Input:
A = list of all arcs
Adjhead = adjacency lists containing arcs starting at event *x*
Adjtail = adjacency lists containing arcs ending at event *x*
Empty = vector containing empty row numbers
r = row number of arc that will be deleted
Output:
A = updated list of all arcs
Adjtail = updated adjacency lists containing arcs starting at event *x*
Adjhead = updated adjacency lists containing arcs ending at event *x*
Empty = updated vector containing empty row numbers

1. $A(r) \leftarrow [0\ 0\ 0\ 0]$; % replace arc by a row of zeros in *Arclist*
2. $Empty \leftarrow Empty \cup r$; % add *r* to vector of empty rows
3. $adjhead(j_r) \leftarrow adjhead(j_r) \setminus r$; % remove pointer *k* from adjacency lists
4. $adjtail(i_r) \leftarrow adjtail(i_r) \setminus r$;

5.2.6 Inserting an arc

The routine ADDARC has been developed for inserting an arc into the arclist while keeping the adjacency lists up to date. The new arc is input for the algorithm, and is a vector of the same format of the rows of *A*, so:

arc = (*j*, *i*, *t*, *type*).

Algorithm 5.3 (ADDARC)

Input:
A = list of all arcs
Adjhead = adjacency lists containing arcs starting at event *x*
Adjtail = adjacency lists containing arcs ending at event *x*
Empty = vector containing empty row numbers
arc = arc that will be inserted
Output:
A = updated list of all arcs
Adjtail = updated adjacency lists containing arcs starting at event *x*
Adjhead = updated adjacency lists containing arcs ending at event *x*
Empty = updated vector containing empty row numbers

1. **if** $Empty \neq \varnothing$ **then**
2. $r \leftarrow Empty(1)$; % use empty row of arclist
3. $Empty \leftarrow Empty\ (2, \ldots, \text{size}(Empty))$; % remove used entry of *Empty*
4. **else**
5. $r \leftarrow \text{size}(A) + 1$; % use new row of arclist
6. $A\ (r) \leftarrow arc$;
8. $Adjhead(j_{arc}) \leftarrow Adjhead(j_{arc}) \cup r$; % add pointers to *arc*
9. $Adjtail(i_{arc}) \leftarrow Adjtail(i_{arc}) \cup r$;

Whenever possible, the new arc is inserted in an empty row in the arclist. When no empty rows are present, the arclist is extended with a new row to create space for the new arc. The routine is shown below. Since no loops are present, the running time is approximately constant.

5.2.7 Conflict matrices

Lists of conflicting train movements are needed for two reasons:

- When generating the timed event graph, hindrance constraint arcs have to be derived and incorporated in the model to ensure time separation between conflicting train movements, as explained in section 4.5.2.
- When implementing dispatching actions, new hindrance constraint arcs have to be derived and kept up to date with the new situation.

Conflict data

The conflicting train movements are stored in the variable *Conflicts*, which is a set of matrices containing all data needed for constructing hindrance conflict arcs. This is implemented using the cell-array notation of Matlab.

Recall from section 4.5.2 that hindrance conflicts are defined to occur between conflicting train lines *LN*, using different IO-points of the same timetable point. Consequently, for each combination of line number, timetable point and IO-point a matrix *Conflicts*{*LN*, *TTP*, *IO*} containing all line numbers of trains that are conflicting with the combination of *LN*, *TTP* and *IO* exists. Each row *r* of such a matrix contains the following objects:

$$Conflicts\{LN, TTP, IO\}(r) = (LN^{conflicting}{}_r, IO^{conflicting}{}_r, t_r)$$

where for each row *r*:

$LN^{conflicting}$ = line number of the conflicting line,
$IO^{conflicting}$ = IO-point used by the conflicting line,
t = minimal amount of time the conflicting train of $LN^{conflicting}$ has to wait for *LN*.

An empty matrix, denoted by $Conflicts\{LN, TTP, IO\} = \varnothing$, means that no train movement is conflicting with the combination {*LN*, *TTP*, *IO*}.

Note that the number of matrices in *Conflicts* can become very large. When for example a small network consists of 4 train lines, running between 5 timetable points, of which the biggest timetable point has 6 IO-points, the number of matrices in *Conflicts* equals $4 \cdot 5 \cdot 6 = 120$. However, most of these matrices are empty since many timetable points contain no conflicting routes, or very small since it is highly unlikely that a train line is conflicting with 10 or more other train lines at the same timetable point. A benefit of this data structure is that it yields quick access to a list of conflicting train lines when the line number, timetable point and IO-point are known, which is advantageous for the running time of the algorithm generating the hindrance constraint arcs. Examples of conflict matrices can be found in the case study, presented in chapter 7.

Generating *Conflicts* matrices

The study of conflicting train movements and (optimization of) interlocking routes through stations is an entire subject within the field of train operations research. Since this reaches beyond the goal and the limitations of this thesis, no detailed attention is given to the generation of the described conflict data. When testing the developed algorithms for this project in the case study, the *Conflicts* matrices were generated by hand.

When applying the developed algorithms in practice, the *Conflicts* matrices have to be generated using detailed knowledge about the system, such as:

- topology of tracks in the stations,
- routing of train lines through stations,
- running characteristics of the trains (e.g. speed, acceleration and decelaration rates, etc.).

5.3 Generating the timed event graph

In this section, an algorithm for generating the timed event graph will be presented. Three extra input variables needed for the generation of the timed event graph are introduced first: Two variables defining the passenger transfers and the rolling stock turns are introduced in section 5.3.1, and a vector containing the running and dwell times as defined in section 5.3.2. After this, the 'Generate TEG' algorithm is presented in section 5.3.3. Extra attention is given to the generation of hindrance constraint arcs, to which section 5.3.4 is dedicated.

5.3.1 Input data for synchronization constraints

Synchronization constraints are not created during the optimization process, so the information about them is only needed in the generation process. Therefore they are defined in this section.

Rolling stock turns

The matrix *Turn* contains all data needed to generate synchronization constraints for turning rolling stock. Each row *r* of the matrix *Turn* contains:

$Turn(r) = (TN^{\text{feeding}}_r, TN^{\text{connecting}}_r, t_r)$

where for each row *r*:

TN^{feeding} = train number of the feeding train,
$TN^{\text{connecting}}$ = train number of the connecting train,
t = minimal amount of time needed for turning.

Note that a turn always connects the end of the feeding train trip with the start of the connecting train trip. Hence, the timetable points where the turn occurs do not need to be stored.

Passenger transfers

The passenger transfers are stored in the matrix *Transfers*. Passenger transfers can occur at several timetable points during a train trip. So, as opposed to rolling stock turns, the timetable point where the transfer occurs has to be stored as well. Each row *r* of the matrix *Transfers* contains:

$Transfers(r) = (TN^{\text{feeding}}_r, TN^{\text{connecting}}_r, TTP_r, t_r)$

where for each row *r*:

TN^{feeding} = train number of the feeding train,
$TN^{\text{connecting}}$ = train number of the connecting train,
TTP = timetable point where the transfer is scheduled to occur,
t = minimal transfer time.

5.3.2 Input data for running and dwell times

The running and dwell times are stored in the vector *Arcweight*. The value of an element *Arcweight*(*i*) is defined as the process time of the process following event *i*. So if event *i* is a departure, then *Arcweight*(*i*) is a running time, and if *i* is an arrival, then *Arcweight*(*i*) is a dwell time.

5.3.3 The Generate algorithm

The timed event graph is used as input for the optimization process described in section 6.5, and is therefore generated in advance using Algorithm 5.4. The input of the algorithm consists of the sets *Event*, *Turns* and *Conflicts*, and of the timetable vector *d*, as described in section 5.2. The matrix *Event* used as input for the 'Generate_TEG' algorithm has two additional properties when compared with its definition in section 5.2.1:

1. The linked lists formed by the last two elements of each row *r* of the matrix *Event* are not present yet.
2. All events belonging to the same train trip occur as a closed group in the list (i.e. with no events of other train trips in between), and in chronological order according to the schedule.

The output of the algorithm is an arclist *A*, containing the timed event graph of the railway system, and the event matrix *Event*, completed with the linked lists. The algorithm works as follows:
In lines 1 – 10, the running and dwell time arcs are generated by scanning all events. When an event of the type 'arrival' or 'end' (of train trip) is found, a running time arc originating from the previous event is generated (line 5). In case of an event of the type 'departure', the arc from the previous event is a dwell time arc, which is generated in line 9. The type 'start' (of train trip) is not regarded by the algorithm here, as no running or dwell time arcs end at such an event.
When all running and dwell time arcs are generated, the linked lists are created in lines 11 – 14. Then, the synchronization constraints are added to the model in lines 16 – 22.
In lines 23 – 29, the headway arcs are generated. This is implemented by creating a list for each IO-point, containing all events occurring there at their scheduled order. Then, all events are connected by headway arcs ensuring the time separation needed for events occurring at the same IO-points (line 27).
Finally, in line 31, Algorithm 5.5 is called for each event $x \in Event$, and the produced hindrance constraint arcs are added to *A*. This algorithm produces the hindrance constraint arcs representing the hindrance conflicts caused by event *x*, and will be described in the next section.

Limitation of this algorithm: standard headways are used
A limitation of this algorithm is that the value used for the headway times is always the same. Obviously in reality the headways can differ depending on train types, the track layout of the station, etc. When using the algorithm for modelling a real railway system, each headway time should be calculated separately. However, the main subject of this thesis is how to implement dispatching actions in the model, and therefore calculating specific headway times is considered to be a subject beyond the scope of this thesis.

Algorithm 5.4 (GENERATE_TEG)

Input:
Event = list of all events occurring in the network, without linked lists
Arcweight = list of process times
d = timetable vector
Turns = list of turning trains
Conflicts = lists of train lines hindering each other

Output:
Event = list of all events occurring in the network, with linked lists
A = arclist of generated timed event graph

1. **for** each event $x \in Event$ **do**
2. **if** $type_x$ = 'arrival' or $type_x$ = 'end' % arrival or end or train line
3. $i \leftarrow x - 1$;
4. $j \leftarrow x$;
5. $A \leftarrow A \cup \{j, i, Arcweight(i), \text{'running'}\}$; % add running arc from previous event
6. **elseif** $type_x$ = 'departure'
7. $i \leftarrow x - 1$;
8. $j \leftarrow x$;
9. $A \leftarrow A \cup \{j, i, Arcweight(i), \text{'dwell'}\}$; % add dwell time arc from previous event
10. **end**
11. **for** event $x \in Event$ **do**
12. find preceding event at the same timetable point *Event_previous* ;
13. find next event at the same timetable point *Event_next* ;
14. $Event(x) \leftarrow (Event(x), Event\_previous, Event\_next)$; % create linked list
15. **for** each row *r* of *Turns* **do**
16. find arrival event x^{feeding} of TN^{feeding}_r ;
17. find departure event $x^{\text{connecting}}$ of $TN^{\text{connecting}}_r$;
18. $A \leftarrow A \cup \{ x^{\text{connecting}}, x^{\text{feeding}}, t^{\text{turn}}_{\text{r}}, \text{'turn'}\}$; % add turning arc
19. **for** each row *r* of *Transfers* **do**
20. find arrival event *E* such that $TTP_E = TTP_r$ and $LN_E = LN^{feeding}_r$;
21. find departure event *D* such that $TTP_D = TTP_r$ and $LN_D = LN^{connecting}_r$;
22. $A \leftarrow A \cup \{ D, E, t_r, \text{'transfer'}\}$; % add transfer arc
23. **for** each timetablepoint *TTP* **do**
24. **for** each IO-point *IO* **do**
25. find first event scheduled *Event_current* ;
26. **while** next scheduled event *Event_next* exists
27. $A \leftarrow A \cup \{Event\_next, Event\_current, t_{\text{headway}}, \text{'headway'}\}$; % add headway arc
28. $Event\_current \leftarrow Event\_next$;
29. find next event scheduled *Event_next* ;
30. **for** each event $x \in Event$ **do**
31. *Arcs_hindrance* ← generate hindrance forwards(*x*) ; % generate hindrance conflict arcs
32. $A \leftarrow A \cup Arcs\_hindrance$; % add hindrance arcs

5.3.4 Generating hindrance constraint arcs

Hindrance constraint arcs between conflicting events (i.e. conflicting train movements) have to be generated when building a timed event graph, as well as when implementing dispatching actions. Two types of hindrance constraint arcs are distinguished: Forward hindrance constraints are arcs

originating from event *x*, running to events conflicting with *x* (i.e. events that have to wait *t* minutes after event *x* has occurred). Backward hindrance constraints are arcs running *to* event *x*, originating from events conflicting with *x* (i.e. when event *x* itself has to wait *t* minutes after the conflicting event has occurred).
When generating a timed event graph, the algorithm 'GenHindrance_Forwards', creating the forward hindrance constraints, is used. Since this algorithm is called for each event present in the model, the incoming hindrance constraints do not need to be generated separately. The algorithm works as follows:
In line 1, the number of rows of *Hindrance*{LN_x, TTP_x, IO_x}, which is the number of train lines for which hindrance is caused by event *x*, is assigned to the variable *Nconflicts*. Recall from 4.5.2 that one outgoing hindrance arc for each hindrance conflict caused by event *x* has to be produced. The counter *Nbuilt* is used to check if all hindrance conflicts are generated. In line 3 the vector *Notbuilt* is created, containing the value 'one' for each row *k* of *Hindrance*{LN_x, TTP_x, IO_x}. This vector is used to flag row *k* if its corresponding hindrance arc has been added. In line 4, the next event scheduled after *x* is retrieved from the linked list.
In line 5, the actual loop starts. Via the linked list, all events *E* scheduled subsequent to event *x* at the same timetable point are visited in the scheduled order. At line 6, the conflict matrix corresponding to event *x* is scanned. Note that only the conflict matrix corresponding to the combination {LN_x, TTP_x, IO_x} has to be scanned. At line 7 each event *E* is checked for having a hindrance conflict with *x*, using the aforementioned conflict matrix. If this is the case, the corresponding row *r* of *Hindrance*{LN_x, TTP_x, IO_x} is flagged (line 8) and the hindrance arc is added to the output matrix *Arcs*. The loop ends if no more events are scheduled at TTP_x or if all events hindered by event *x* are found, which is when *Nbuilt* equals *Nconflicts* (recall from section 4.5.2 that the number of outgoing hindrance arcs from an event equals the number of train lines conflicting with it).

Algorithm 5.5 (GENHINDRANCE_FORWARDS)

Input:
Event = list of events
Hindrance = lists of hindrance constraints between train lines
x = event for which forward hindrance arcs have to be generated
Output:
Arcs = list of hindrance constraint arcs

```
1.  Nconflicts ← size(Hindrance{LN_x, TTP_x, IO_x}) ;
2.  Nbuilt ← 0 ;
3.  Notbuilt = ones(Nconflicts) ;
4.  E ← N_x ;                                          % next event scheduled after x
5.  while E exists & Nbuilt < Nconflicts
6.     for each row r of Hindrance{LN_x, TTP_x, IO_x}   % search conflict list
7.        if LN_E = LN_r & IO_E = IO_r & Notbuilt (r) then
8.           Notbuilt (r) ← 0 ;                        % flag corresponding row of Hindrance
9.           Nbuilt ← Nbuilt + 1 ;
10.          Arcs ← Arcs ∪ {E, x, t^Hindrance_r , 5} ;  % add hindrance arc
11.    E ← next event scheduled after E ;
12. return Arcs
```

A periodic timetable is important for this algorithm
Note that the algorithm works properly only when the timetable is periodic. As explained in section 4.6, an *acyclic* timed event graph of a *periodic* timetable is used in this thesis to implement dispatching actions. An example of what can happen when the timetable is not periodic is shown in Figure 5.3. Suppose that train line 2 is a line that runs only once per day and

that the routing of its departure is conflicting with train line 1. Algorithm 5.5 would in this case build hindrance constraint arcs from each departure event of train line 1 to the conflicting departure event of train line 2, which is continuously the same event. This yields many redundant arcs and an increased complexity of the graph, and is therefore undesirable. When the timetable is periodic, conflicting events are always near each other (i.e. within one hour), and the described problem can therefore never occur.

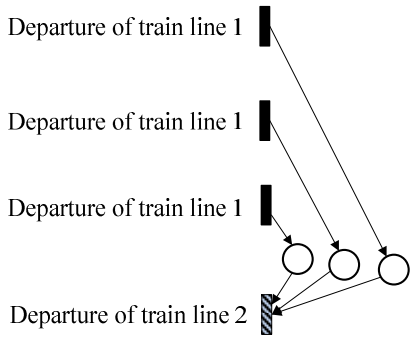


Figure 5.3 *Bad hindrance constraints generated when a timetable is not periodic..*

5.4 Calculating the delay propagation in topological order

In order to determine the effectiveness of dispatching actions, the propagation of delays in the railway system has to be calculated. Goverde [7] developed an efficient algorithm to compute the delay propagation in large scale networks. However, this algorithm only works under the requirement that the timetable is periodic. Since an acyclic graph is used in this thesis, a new algorithm has been implemented.

The algorithm makes use of the fact that the propagation of each delay scenario can be calculated by observing each event only once, which is time efficient. The running time of the algorithm is $O(n)$, where n denotes the number of arcs in A. However, this is only possible if all events are observed in the correct order, which is known in graph theory as *topological order*. For instance, consider the timed event graph of Figure 5.4, where the delay of event 3 can only be calculated if the delays of both event 2 and event 5 are known. To make sure that all delays are calculated in the correct order, a recursive loop has been implemented. The algorithm consists of two parts and is based on recursive depth-first search algorithms, examples of which can be found in [4].

The input of the algorithm is the arclist A, with the adjacency lists for the heads *Adjhead*, timetable vector d and the initial delay vector Z. The output is an updated delay vector Z, with all propagated delays and a vector *Proparc*, containing the row numbers r of arcs from the arclist which are involved in propagating delays. This is used by the optimization algorithm (see section 6.5) to determine which trains are delayed, so that the appropriate dispatching actions can be selected.

Algorithm 5.6 and Algorithm 5.7 show the pseudo code, and their behaviour is illustrated in Figure 5.4. The corresponding arclist can be found in Figure 5.1. How would the algorithm calculate the delay propagation in this graph? First, all nodes are marked ‘unvisited’, after which Algorithm 5.6 will visit each event subsequently by calling Algorithm 5.7.

Algorithm 5.6 (PROPAGATE)

Input:
A = arclist
Adjhead = adjacency lists for heads
d = vector with all scheduled event times
Z = initial delay vector
Output:
Z = updated delay matrix
Proparc = list of arcs propagating delay

```
1.  unvisited ← ones(size(d)) ;                          % mark all nodes unvisited
2.  for each event x ∈ A do
3.     if unvisited(x) = 1                               % visit all unvisited nodes
4.        Z, Proparc ← visit node (x, unvisited, Adjhead, A, Z, d) ;
5.  return Z, Proparc
```

Algorithm 5.7 (VISIT)

Input:
x = event that will be visited
A = list of all arcs
Adjhead = adjacency lists for heads
d = vector with all scheduled event times
Z = initial delay matrix
unvisited = list of unvisited events
Output:
Z = updated delay matrix
unvisited = updated list of unvisited events
Proparc = list of arcs propagating delay

```
1.   unvisited(x) ← 0 ;                                  % mark node x visited
2.   if Adjhead(x) ≠ ∅ then
3.      for each preceding arc r ∈ Adjhead(x) do         % check all preceding arcs
4.         if unvisited(i_r) = 1 then
5.            visit node (i_r , unvisited,Adjhead, A, Z, d) ;
6.         delay ← d( i_r ) + Z( i_r ) + t_r – d( x ) ;
7.         if delay > 0 then
8.            Z(x) ← max(Z(x), delay) ;                 % update delay
9.            Proparc ← Proparc ∪ r ;
10.  return Z, unvisited
```

The key steps in Algorithm 5.7 are lines 2 and 3, where the algorithm checks whether any preceding arcs are present. For event 1 this is obviously not the case (see situation I of Figure 5.4), so the delay is calculated and updated when necessary (lines 7-8). When a delay is propagated via the arc under consideration, the arc is added to the vector *Proparc* in line 9. While visiting event 2 (situation II), event 1 is found as a predecessor. Since this event is already marked as visited, the delay can be calculated without problems, and the updated delay is returned to Algorithm 5.6.
Now, the algorithm arrives at event 3, where two predecessors are found (events 2 and 5). Here, the recursive loop comes into action. Since event 5 is still unvisited, the 'Visit' algorithm will call itself (line 6 of Algorithm 5.7) to visit event 5 while still in the process of visiting event 3 (situation III). During the visit of event 5, the same will happen for event 4, because this is an

unvisited predecessor of event 5 (situation IV). During the visit of event 4, no more predecessors are found, so Algorithm 5.7 proceeds with lines 7-10 (calculating the delay of event 4), after which the delays of events 5 and 3 will be calculated as well (situation V). Finally, event 6 will be visited. As can be seen, each arc is visited only once.

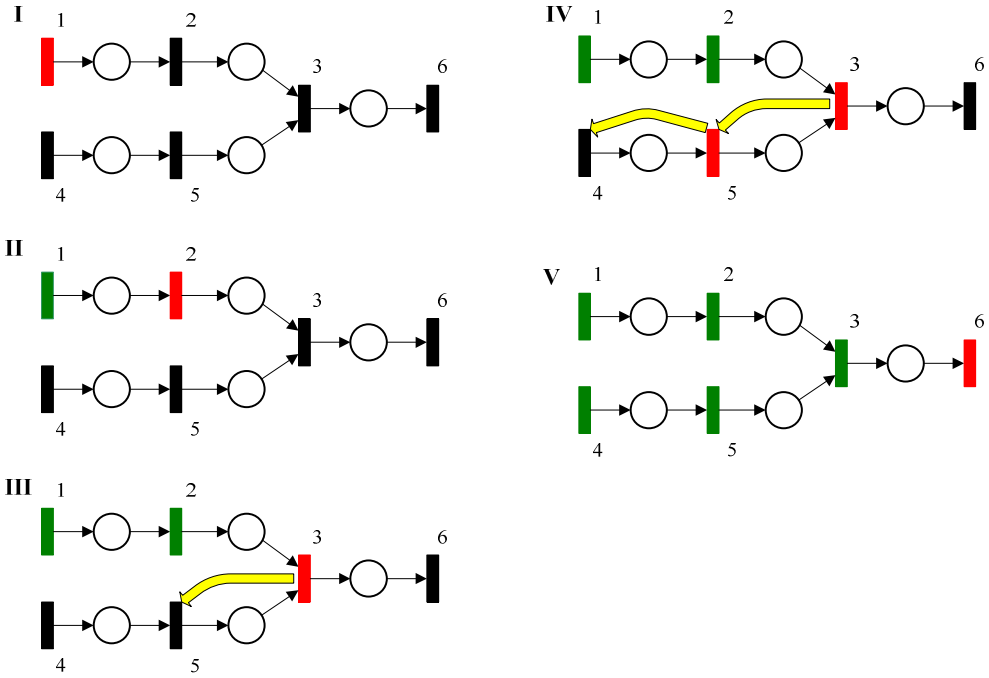


Figure 5.4 *Calculating the delay propagation in topological order. Events are shown in green when visited and in red while being visited at the moment. Recursive calls are shown as yellow arrows.*

The recursive calls of events 5 and 4 are graphically represented in Figure 5.5. In this figure can be seen that the recursive calls of Algorithm 5.7, resulting from its check for preceding events, automatically lead to the correct, topological, order for calculating the delays. Note that the example could have been calculated in many other ways. When for instance event 6 was visited first, all other events would have been visited recursively, as they are all predecessors of event 6.

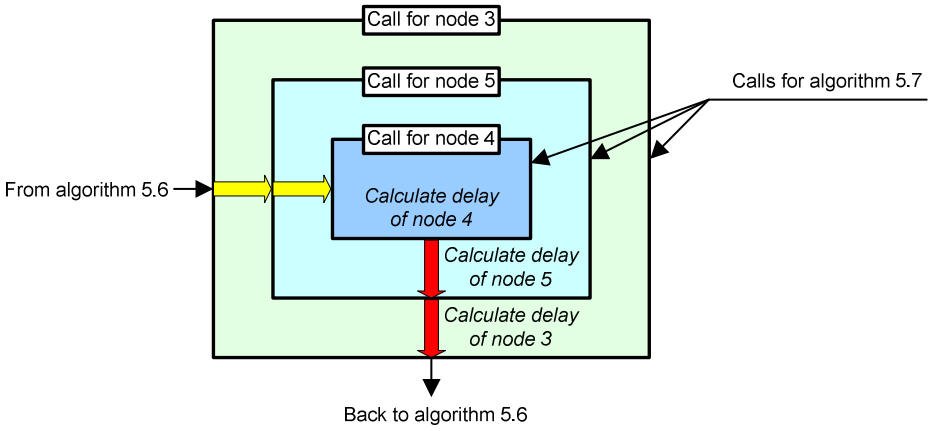


Figure 5.5 *Graphical representation of recursive calls (yellow arrows) and their returns (red arrows) in delay propagation algorithm 5.7.*

5.5 Calculating the capacity consumption of a railway track

As a measure of the traffic density on a railway line, the capacity consumption may be used as an indication of the amount of slack time in the schedule. This can be useful when evaluating dispatching actions in order to reduce the delays on a certain railway track. Therefore, an algorithm calculating the capacity consumption of a railway track has been developed.
Capacity consumption is a property of a *railway line* instead of one point along the railway track. Before the algorithm is explained, the definition of capacity consumption is considered in more detail. In Figure 5.6 a blocking time diagram with two trains is shown. The capacity consumption can be visualized by moving the timetable paths as closely together as possible with regard to the blocking times [14], which can be done by moving the trains together by t_b^{min} minutes. The time t_b^{min} is defined as the *minimum line headway*, which is the smallest buffer time between trains on a railway line. A minimum line headway exists between each subsequent pair of trains on a railway line. The *capacity consumption* of a railway track during a period is defined as the sum of the minimum line headways divided by the total duration of that period [12].
As explained in chapter 4, instead of modelling all blocks only the stations are modelled in the timed event graphs used for this project. Furthermore, the time separation between trains on a railway line is modelled in the timed event graph by including headway constraints between the trains at timetable points. This simplified situation has been visualized in Figure 5.7, where the red arrows denote the headways that have to be respected to get a conflict-free schedule. In order to determine the minimum line headway, the scheduled buffer times have to be calculated first. The algorithm calculates the scheduled buffer times with the following formula:

$$t_b = d_2 - d_1 - h_{12} \tag{5.1}$$

where:
t_b = buffer time,
d_2 = scheduled arrival, through or departure time of train 2,
d_1 = scheduled arrival, through or departure time of train 1,
h_{12} = minimum headway between trains 1 and 2 (following from infrastructure constraints).

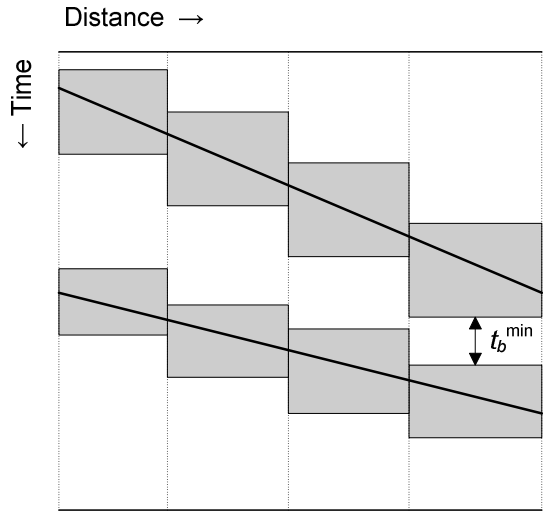


Figure 5.6 *The time separation between trains as modelled by a blocking time diagram.*

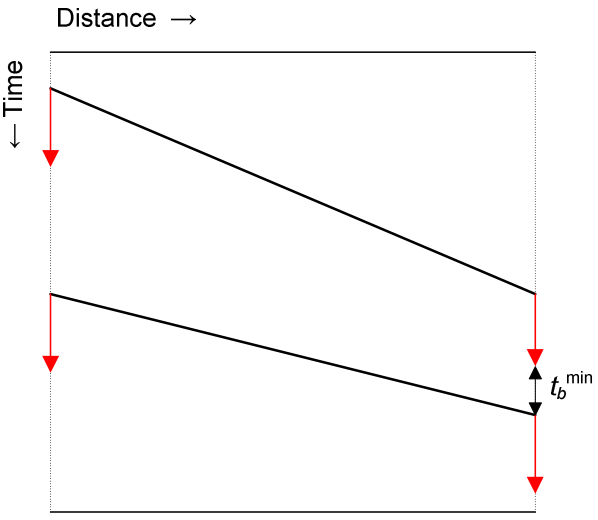


Figure 5.7 *The time separation between trains as modelled in a timed event graph.*

Now, the *minimum line headway* $t_b^{\min}$ between two trains is the smallest value found for t_b along the considered railway line, as shown in Figure 5.7. Note that this calculation yields the same results as the calculation using a blocking time diagram when the correct values for h (the headways) are used. Furthermore note that this way of determining $t_b^{\min}$ incorporates influences on the minimum line headway (e.g. the presence of longer blocks along the line, etc.) as long as those influences have been taken into account when determining the arc weights of the timed event graph of the considered railway network.
When the minimum line headways for all trains at the considered railway line in the considered time period are found, the capacity consumption η is calculated with:

$$\eta = 1 - \frac{\sum t_b^{\min}}{t_p}$$

where:

η = capacity consumption,
$t_b^{\min}$ = minimum line headway,
t_p = total time of considered period.

The required calculations are implemented in Algorithm 5.8 ‘Capacity_Consumption’. The input of the algorithm contains the timed event graph represented by the variables *A*, *Adjtail*, *d* and *Event*, and the variables determining at which railway track and time period the capacity consumption has to be calculated. The variable *x* is the event number with which a train trip starts its trip on the considered railway line *before* the considered period t_p starts. An example of such a train trip is train 1, shown in Figure 5.8, where the location in time of event *x_begin* is shown as well. TTP_{end} is the timetable point where the algorithm stops calculating, i.e. the end of the considered train line, as shown in Figure 5.8. The start and the end of the considered period are given by *t_begin* and *t_end* respectively.

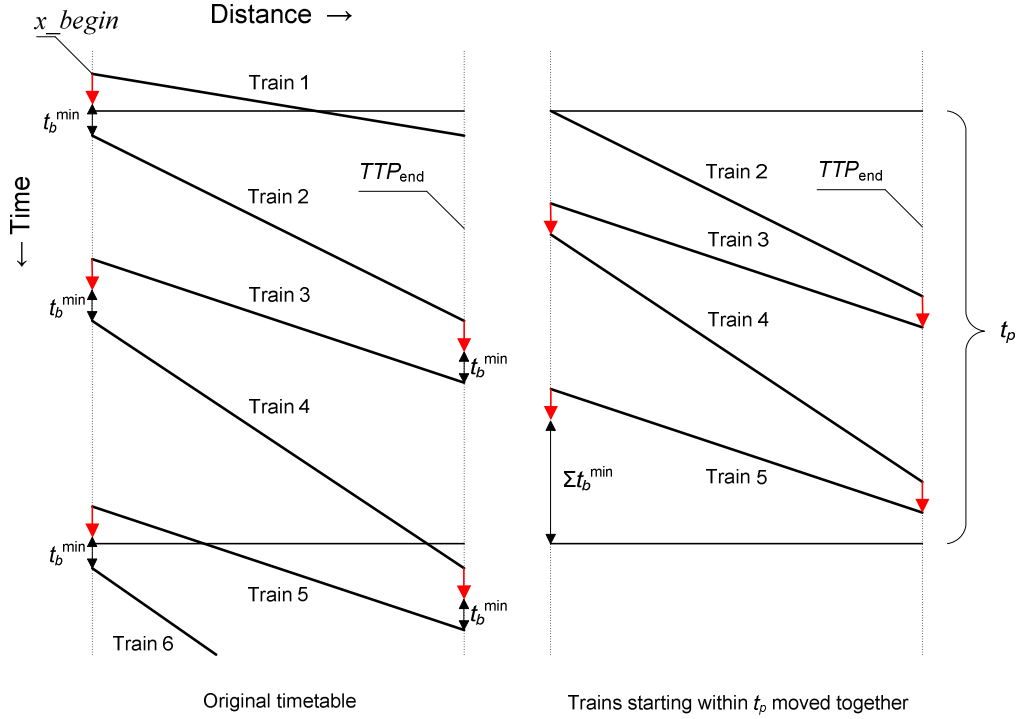


Figure 5.8 *Illustration of trains moved together by the algorithm 'Capacity consumption'.*

The three following assumptions have to be true for the algorithm to work correctly:

1. The event *x_begin* is scheduled earlier than *t_begin*.
2. Each train trip starting between *t_begin* and *t_end* runs to TTP_{end}.
3. The sequence order of the trains on the considered railway line stays the same, i.e. trains do not overtake each other.

Figure 5.8 visualizes how the algorithm works. Starting at event *x*, the algorithm finds all events starting at the considered railway line which are scheduled *before* a train starting within period t_p. In lines 1 – 5, the vector *startevents* is filled with these events. In the example of Figure 5.8, this vector would contain the first events of trains 1, 2, 3 and 4. In lines 7 – 20, the algorithm 'walks along' the train trips until the end of the considered railway line in order to determine t_b^{min} for each pair of subsequent trains. These minimum line headways are stored in the vector *buffertimes*, which is used in line 25 to calculate the capacity consumption.

Algorithm 5.8 (CAPACITY_CONSUMPTION)

Input:

A	= list of all arcs
Adjtail	= adjacency lists for heads
d	= vector with all scheduled event times
Event	= list of all events
x	= first event of a train trip starting before *t_begin*
TTP_{end}	= last timetable point of considered railway line
t_begin	= start time of considered period
t_end	= end time of considered period

Output:

p	= capacity consumption

```
1.  while x exists & d(x) < t_end do                          % search starting events
2.    find next event x_successor scheduled after x at the same IO-point ;
3.    if d(x_successor)>=t_begin  &  d(x_successor)<t_end
4.      startevents ← startevents ∪ x ;
5.      x ← x_successor ;
6.  end
7.  for each event x_start ∈ startevents do
8.    tb_min ← inf ;
9.    x ← x_start ;
10.   while x exists do                                       % follow train trip to calculate t_b^min
11.     find next event x_successor scheduled after x at the same IO-point, and headway h ;
12.     if d(x) + t  =< t_begin
13.       tb_min ← min(tb_min, (d(x_successor) – t_begin)) ;
14.     else
15.       tb_min ← min(tb_min, (d(x_successor) – d(x) – h )) ;
16.     if TTP_x = TTP_end do
17.       break                                               % stop if TTP_end is reached
18.     find next event x_next of the train trip ;
19.     x ← x_next ;
20.   buffertimes ← buffertimes ∪ tb_min ;
21. end
22. find next event x_successor scheduled after the last event x_start and the headway h ;
23. supplement ← t_end – (d(x_successor) + t) ;
24. buffertimes ← buffertimes ∪ supplement ;
25. p ←  1 – sum(buffertimes) / (t_end – t_begin) ;
```

Note that train 2 is moved up only until the start of the considered period t_p (see again Figure 5.8). This is compensated at the end of t_p, where instead of the entire t_b^{min} between trains 5 and 6, only the time until the end of t_p is considered. This is called the 'supplement' (see Figure 5.9), which is calculated in line 23.

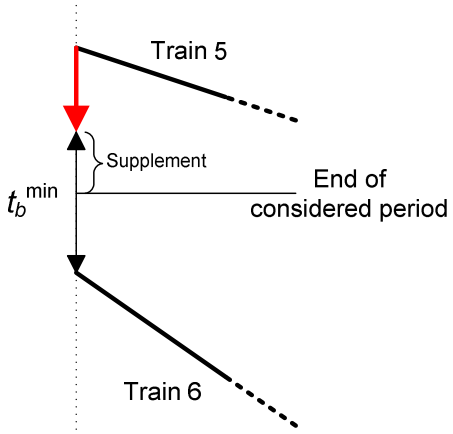


Figure 5.9 *Detail of figure 5.7 illustrating the supplement.*

The algorithm is able to handle delayed and/or rescheduled trains as well. When delays occur and/or dispatching actions have been applied, the delay propagation through the network has to be calculated with the algorithm 'Propagate', presented in the previous section. The event times calculated with this algorithm can then be used as input for the 'Capacity_Consumption' algorithm, instead of the scheduled event times. Formula 5.1 then becomes:

$$t_b = x_2 - x_1 - h_{12} \tag{5.2}$$

where:
t_b = buffer time,
x_2 = calculated arrival, through or departure time of train 2,
x_1 = calculated arrival, through or departure time of train 1,
h_{12} = time headway between trains 1 and 2.

Limitations of this algorithm
A limitation of this algorithm is that in some situations the timetable paths are moved closer together then would have been actually possible in reality. This is caused by the fact that the actual occupation of a block or track is not modelled, which is a limitation of the model itself already discussed in section 4.7. In some situations this can lead to a slightly lower calculated capacity consumption.

5.6 Conclusion

The ability to store and modify a timed event graph with minimal memory usage and optimal time efficiency is crucial for the development of a system that can calculate the effectiveness of dispatching actions quickly enough for real time operation. A data structure using adjacency lists possesses these qualities, and has been presented in this chapter.
After defining the variables used for storing the timed event graph in the computer memory, an algorithm for the generation of the graph was presented. Finally, algorithms for calculating the delay propagation in the network and the capacity consumption of a railway line have been discussed.

6 Implementing dispatching actions

6.1 Introduction

Chapter 5 described how a timed event graph is generated and stored in the computer memory as an arclist. A time efficient delay propagation algorithm has been introduced as well. The most important question in case of delays is: which is the most effective dispatching action? To determine the effectiveness of a dispatching action, it has to be implemented in the timed event graph, after which the delay propagation can be calculated.
This chapter deals with algorithms that can change the timed event graph in order to represent dispatching actions, and therefore contains the most important result of this research project. In section 6.2, the algorithm 'ChangeOrder' is presented, with which the sequence order of two trains running along a railway line can be changed. Aside of changing the sequence order of trains, this can be used to:

- Move the location of a scheduled overtaking,
- Remove a scheduled overtaking,
- Introduce a new overtaking.

Section 6.3 is dedicated to an algorithm with which train movements at stations can be postponed, which is used to reflect a change in the order of crossing train movements. A dispatching action with more consequences for the travellers is to short-turn a train, which can be implemented in the model by the algorithm presented in section 6.4. In order to find the effective dispatching actions, a simple optimization algorithm will be described in section 6.5. As explained in chapter 2, the railway system as modelled by a discrete event system can be represented by a system of max-plus equations. In section 6.6, a way of formally describing the possibility to apply dispatching actions in max-plus algebra will be introduced. Section 6.7 contains the conclusion of this chapter.

6.2 Change the order between trains

Consider the trains running from station 1 to station 3 depicted in the example of Figure 6.1, where train 1 is scheduled to run before train 2. At station 1, the trains arrive at two different IO-points, but they leave the station using the same IO-point (IO-point 3). This implies that the two trains have merged to the same track. They continue using the same infrastructure until their arrival at the terminal station, station 3. Now suppose that train 1 is delayed, and has to be postponed on the route between station 1 and station 3. This dispatching action implies that at four IO-points, namely IO-point 3 of station 1, IO-points 1 and 2 of station 2 and IO-point 3 of station 3 their running order has to be changed.

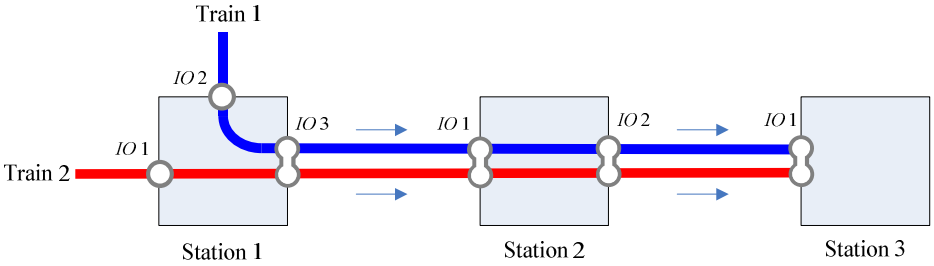


Figure 6.1 *Two trains sharing the same infrastructure between stations 1 and 3.*

The algorithm presented in this section starts working at the IO-point where the affected trains leave the timetable point from which the dispatching action starts. Then, the algorithm works its way along the train lines towards the end of the dispatching action. At each IO-point the same modifications of the timed event graph are performed, namely changing the headway constraints and changing the hindrance constraints. The modifications will be explained in the next two sections, after which the actual algorithm will be explained.

6.2.1 Construction rule for changing headway constraints

As described in section 4.5.2, all events occurring at the same IO-point of the same timetable point are connected with headway constraints to ensure the correct time separation between subsequent trains on the same track. When the order of two trains is changed, the headway constraints change accordingly. Figure 6.2 shows the headway constraints between a series of subsequent events in case of the originally scheduled order and in case of a changed order, in which event 1 has been postponed. The event preceding event 1 is called x_p and the event subsequent to event 2 is called x_n. For each IO-point where the order is changed, three headway arcs have to be changed. The necessary changes are summarized in construction rule 1, shown below.

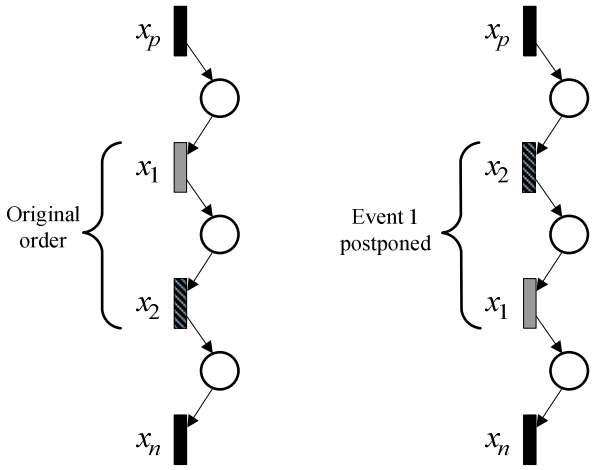


Figure 6.2 *Changing headway constraints when the order between trains is changed.*

Construction rule 1

Given:
- event x_2 scheduled after event x_1.
- $TTP_1 = TTP_2$ and $IO_1 = IO_2$ (events x_1 and x_2 use the same IO-point at the same timetable point).

When changing the running order of x_1 and x_2 three headway constraint arcs have to be changed as follows:

	Arc in original graph	**Becomes arc in changed graph**
1.	preceding event x_p to x_1	preceding event x_p to x_2
2.	x_1 to x_2	x_2 to x_1
3.	x_2 to next event x_n	x_1 to next event x_n

6.2.2 Changing the hindrance constraints

At stations where hindrance conflicts can occur due to conflicting interlocking routes, the hindrance arcs have to be updated in order to be consistent with the new situation. This is done in two steps:

1. The first scheduled event x_1 is removed from the linked list, and the hindrance constraints are updated accordingly.
2. Event x_1 is inserted in the linked list directly behind event x_2, and the hindrance constraints are updated accordingly.

For step one, an algorithm able to update the hindrance constraints when an event is removed from the scheduled order of events has been developed. Step two is carried out by an algorithm updating the hindrance constraints when adding a new event to the scheduled order of events. These algorithms will be discussed in the next section before the actual ChangeOrder algorithm using them will be discussed.

6.2.3 Construction rule for removing hindrance constraints

Three different situations can occur when removing an event *x* from the sequence of scheduled events at a timetable point:

- A pair of incoming and outgoing hindrance arcs can be deleted.
- A single incoming hindrance arc has to be connected with an event scheduled later than event *x*.
- A single outgoing hindrance arc has to be connected with an event scheduled sooner than event *x*.

All situations will be described with the example of two conflicting routings shown in Figure 6.3. Each time, a departure event of line 1 will be deleted to show the implications in the hindrance constraint arcs.

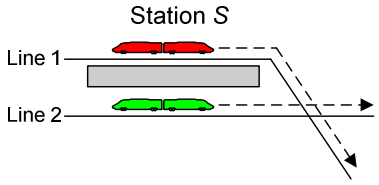


Figure 6.3 *Example of conflicting routings.*

A pair of incoming and outgoing hindrance arcs can be deleted

This situation occurs when the deleted event *x* has an incoming hindrance arc from an event *i* and an outgoing hindrance arc to an event *j* where $LN_i = LN_j$ and $IO_i = IO_j$ (i.e. events *i* and *j* represent events of the same train line using the same IO-point. When removing the conflicting event *x* scheduled between *i* and *j* the hindrance constraints can be removed as well, since the time separation between events *i* and *j* is still assured by headway constraints, as *i* and *j* occur at the same IO-point. This is shown in Figure 6.4.

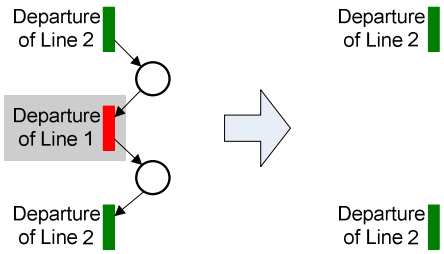


Figure 6.4 *A pair of incoming and outgoing hindrance arcs is deleted (only hindrance arcs are shown).*

A single incoming hindrance arc

When event *x* has an incoming hindrance arc from an event *i* but no outgoing hindrance arc to an event *j* such that $LN_i = LN_j$ and $IO_i = IO_j$, the incoming hindrance arc has to be connected to the next event *y* scheduled after *x* such that $LN_y = LN_x$ and $IO_y = IO_x$. This is shown in Figure 6.5. Note that the new hindrance arc is necessary to ensure the time separation between the two remaining events, as these events are not separated by a headway constraint since they do not use the same IO-points.

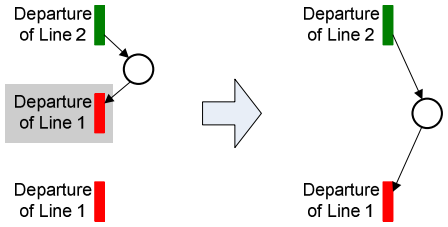


Figure 6.5 *A single incoming hindrance arc is connected with another event (only hindrance arcs are shown).*

A single outgoing hindrance arc

This situation is similar to the previous situation, but in this case, only an outgoing arc is present. The outgoing arc has to be connected to an event *y* scheduled preceding to *x*, such that $LN_y = LN_x$ and $IO_y = IO_x$, as shown in Figure 6.6. As with the previous example, this is necessary to ensure the time separation between two conflicting events which are not separated by headway constraints since they do not use the same IO-point.

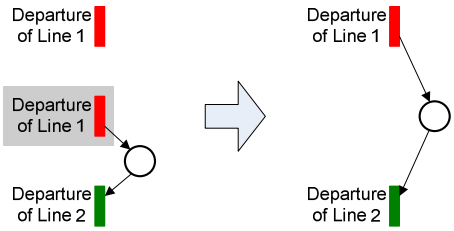


Figure 6.6 *A single outgoing hindrance arc is connected with another event (only hindrance arcs are shown).*

Construction rule

The necessary changes in the timed event graph when removing an event from the scheduled order of events at a timetable point follow from the three examples described above, and are summarized in the construction rule below.

Construction rule 2
Given:
- event x
- $Conflicts\{LN_x , TTP_x , IO_x\} \neq \emptyset$

When removing event x from the scheduled order of events at TTP_x, the following situations with their respective implications for the timed event graph can occur:

	Situation	**Implications**
1.	An incoming hindrance arc (i, x) and an outgoing hindrance arc (x, j) exist such that $IO_i = IO_j$ & $LN_i = LN_j$.	Both hindrance arcs can be deleted.
2.	An incoming hindrance arc (i, x) exists, but no outgoing hindrance arc (x, j) such that $IO_i = IO_j$ & $LN_i = LN_j$ exists.	The incoming arc (i, x) has to be connected to an event y scheduled subsequent to x such that $IO_y = IO_x$ & $LN_y = LN_x$.
3.	An outgoing hindrance arc (x, j) exists, but no incoming hindrance arc (i, x) such that $IO_i = IO_j$ & $LN_i = LN_j$ exists.	The outgoing arc (x, j) has to be connected to an event y scheduled preceding to x such that $IO_y = IO_x$ & $LN_y = LN_x$.

Algorithm for removing an event from the scheduled order of events
Construction rule 2 has been implemented in the algorithm 'Delevent_Hindrance' shown below. The input of the algorithm consists of the timed event graph represented by *A*, *Adjhead*, *Adjtail* and *Event*, and the event x that has to be removed from the scheduled order of events. The output consists of the matrix *Arcs2add*, of which each row is an arc that has to be added to the timed event graph according to the construction rule, and the vector *Arcs2del*, containing the row numbers of arcs that have to be deleted from *A*.
The algorithm works as follows: In lines 1 – 4, the linked lists containing the scheduled order of events are updated. In lines 5 and 6 the headway constraints are explored to find the next and the previously scheduled events n and p of the same train line. This is used in lines 7 – 17 to implement the construction rule.
No big vectors or matrices have to be processed in this algorithm, so its running time depends only on the complexity of the graph (i.e. the numbers of incoming and outgoing hindrance arcs to and from x, and whether the next and previously scheduled events n and p are found quickly by searching headway constraints in lines 5 and 6).

Algorithm 6.1 (DELEVENT_HINDRANCE)

Input:
A = arclist
Adjhead = adjacency lists for heads
Adjtail = adjacency lists for tails
Event = list of events
x = event that will be removed from the scheduled order of events
Output:
Arcs2add = arcs that have to be inserted in the timed event graph
Arcs2del = list of row numbers of arcs that have to be deleted
Event = updated list of events
1. **if** $P_x \neq 0$ % remove event from linked lists
2. $N(P_x) \leftarrow N_x$;

3. **if** $N_x \neq 0$
4. $P(N_x) \leftarrow P_x$;
5. find next scheduled event n such that $IO_n = IO_x$ & $LN_n = LN_x$;
6. find previous scheduled event p such that $IO_p = IO_x$ & $LN_p = LN_x$;
7. **for** each incoming hindrance arc r (i , x) to x **do**
8. $Arcs2del \leftarrow Arcs2del \cup r$; % construction rule 2.1
9. find outgoing hindrance arc u (x , y) to an event y such that $IO_y = IO_i$ & $LN_y = LN_i$;
10. **if** u exists
11. $visited(u) \leftarrow 1$; % mark outgoing arc visited
12. **elseif** n exists
13. $Arcs2add \leftarrow Arcs2add \cup \{n , i_r , t_r ,$ 'hindrance'$\}$; % construction rule 2.2
14. **for** each outgoing hindrance arc u (x , j) from x **do**
15. $Arcs2del \leftarrow Arcs2del \cup u$; % construction rule 2.1
16. **if** $visited(u) = 0$ & p exists
17. $Arcs2add \leftarrow Arcs2add \cup \{j_u , p , t_u ,$ 'hindrance'$\}$; % construction rule 2.3

6.2.4 Construction rule for inserting hindrance constraints

When inserting an event x into the scheduled order of events, the timed event graph has to be updated with respect to two aspects:

- Hindrance constraints to and from x have to be inserted into the timed event graph.
- Some existing hindrance constraints can be deleted.

To illustrate this, the example shown in Figure 6.3 will be used once more.

Hindrance arcs have to be inserted

Hindrance arcs have to be added to the timed event graph when an event x is inserted between two conflicting events, as shown in Figure 6.7. More precisely, for each line of the matrix $Conflicts\{LN_x , TTP_x , IO_x\}$, an outgoing hindrance arc to the next scheduled conflicting event and an incoming hindrance arc from the previously scheduled conflicting event has to be constructed.

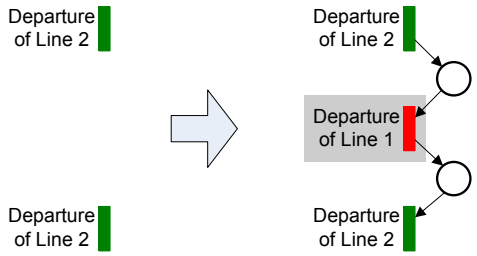


Figure 6.7 *Example of hindrance arcs that have to be inserted when the departure of Line 1 is inserted.*

Existing hindrance arcs have to be deleted

An example of a situation where existing hindrance arcs have to be deleted is shown in Figure 6.8. When inserting a departure of line 1, no incoming hindrance arc is needed, since the time separation between the two departures of the same train line is already assured by headway constraints. An outgoing headway arc to the departure event of line 2 is constructed, and the existing hindrance constraint can be deleted as the required time separation is ensured by the new constraints.

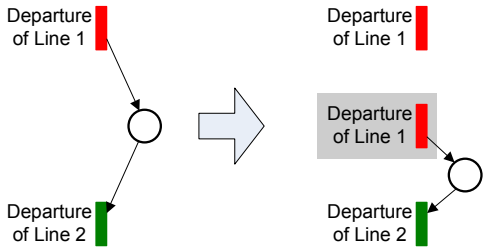


Figure 6.8 *Example of a situation where hindrance arcs have to be deleted when inserting an event.*

The necessary changes to the timed event graph described above are summarized in construction rule 3. Construction rule 3.1 describes the construction of new outgoing (forward) hindrance constraints, while rule 3.2 describes the incoming (backward) hindrance constraints. The situation that a new hindrance constraint is not needed, as shown in the example of Figure 6.8, is reflected by the event *q* in the construction rule. In the example, event *q* would be the previous departure of line 1 in the upper right corner of Figure 6.8.

Construction rule 3

Given:
- event x
- $Conflicts\{LN_x , TTP_x , IO_x\} \neq \varnothing$

When inserting event x in the scheduled order of events at TTP_x , for each row of $Conflicts\{LN_x , TTP_x , IO_x\}$ the following hindrance arcs have to be inserted or deleted:

	Hindrance arcs to insert	**Hindrance arcs to delete (when existing)**
1.	An outgoing hindrance arc to the conflicting event i scheduled subsequent to event x, unless an event q such that $IO_q = IO_x$ & $LN_q = LN_x$ is scheduled in between.	In incoming hindrance arc (j , i) running to i from an event j such that $IO_j = IO_x$ & $LN_j = LN_x$.
2.	An incoming hindrance arc from the conflicting event i scheduled preceding to x, unless an event q such that $IO_q = IO_x$ & $LN_q = LN_x$ is scheduled in between.	An outgoing hindrance arc (i , j) running from i to an event j such that $IO_j = IO_x$ & $LN_j = LN_x$.

Algorithm for inserting hindrance constraints

Construction rule 3 is implemented in the following algorithm. In lines 1 – 4 the linked lists containing the scheduled order of events are updated. The variables created in lines 5 and 6 are used to check whether all conflicts have been implemented, so that the algorithm can stop when this is the case. In lines 8 – 19, the algorithm walks along the linked list of events (lines 7 and 18), to build all outgoing hindrance arcs from *x*. In line 16 it is checked whether a hindrance arc exists that has to be deleted according to construction rule 3.1. When an event *q* is found according to the construction rule, the while loop started in line 8 is broken in line 9. The algorithm will then continue with line 20.

In lines 22 – 32, the incoming hindrance arcs according to construction rule 3.2 are constructed. Again it is checked whether existing hindrance arcs have to be deleted, this time in line 30. The algorithm terminates via line 23 when an event *q* of the same train line LN_x is found, or when no previous event *P* is found anymore.

Algorithm 6.2 (INSERTEVENT_HINDRANCE)

Input:

A	= arclist
Adjhead	= adjacency lists for heads
Adjtail	= adjacency lists for tails
Event	= list of events
x	= event that will inserted in the scheduled order of events
a	= event after which *x* will be inserted
Conflicts	= conflict matrices for hindrance conflicts

Output:

Arcs2add	= arcs that have to be inserted in the timed event graph
Arcs2del	= list of row numbers of arcs that have to be deleted
Event	= updated list of events

1. $N_a \leftarrow x$; % update linked lists
2. $P_x \leftarrow a$;
3. $N_x \leftarrow N_a$;
4. **if** $N_a \neq 0$ **then** $P(N_a) \leftarrow x$; **end**
5. *Nconflicts* ← size(*Hindrance*{LN_x, TTP_x, IO_x}) ;
6. *Nbuilt* ← 0 ; *Notbuilt* = ones(*Nconflicts*) ;
7. $E \leftarrow N_x$;
8. **while** *E* exists & *Nbuilt* < *Nconflicts* % create forward hindrance arcs
9. **if** $LN_E = LN_x$ & $IO_E = IO_x$ **then break** ;
10. **for** each row *r* of *Hindrance*{LN_x, TTP_x, IO_x} % search conflict list
11. **if** $LN_E = LN_r$ & $IO_E = IO_r$ & *Notbuilt* (*r*) **then**
12. *Notbuilt* (*r*) ← 0 ; % flag corresponding row of *Hindrance*
13. *Nbuilt* ← *Nbuilt* + 1 ;
14. *Arcs* ← *Arcs* ∪ {*E*, *x*, $t^{Hindrance}{}_r$, 5} ; % add hindrance arc
15. **for** each incoming hindrance arc *m* from an event *b* to *E* **do**
16. **if** $LN_b = LN_x$ & $IO_b = IO_x$ **then**
17. *Arcs2del* ← *Arcs2del* ∪ *m* ;
18. $E \leftarrow N_E$;
19. **end**
20. *Nbuilt* ← 0 ; *Notbuilt* = ones(*Nconflicts*) ;
21. $E \leftarrow P_x$;
22. **while** *P* exists & *Nbuilt* < *Nconflicts* % create backward hindrance arcs
23. **if** $LN_E = LN_x$ & $IO_E = IO_x$ **then break** ;
24. **for** each row *r* of *Hindrance*{LN_x, TTP_x, IO_x} % search conflict list
25. **if** $LN_E = LN_r$ & $IO_E = IO_r$ & *Notbuilt* (*r*) **then**
26. *Notbuilt* (*r*) ← 0 ; % flag corresponding row of *Hindrance*
27. *Nbuilt* ← *Nbuilt* + 1 ;
28. *Arcs* ← *Arcs* ∪ {*x* , *E* , $t^{Hindrance}{}_r$, 5} ; % add hindrance arc
29. **for** each outgoing hindrance arc *m* from *E* to an event *b* **do**
30. **if** $LN_b = LN_x$ & $IO_b = IO_x$ **then**
31. *Arcs2del* ← *Arcs2del* ∪ *m* ;
32. $E \leftarrow P_E$;
33. **end**

6.2.5 The algorithm 'ChangeOrder'

The algorithm consists of two parts. In the 'explore part' (lines 1 – 33), the timed event graph is explored and an inventory of all changes necessary to implement the dispatching action is made. All necessary changes are stored temporarily in the vector *arcs2del* (row numbers in *A* of arcs that have to be deleted) and in the matrix *arcs2add* (arcs that have to be inserted). After the explore part, the actual changes are performed in the 'graph update part' (lines 34 – 44). The separation in two sections is necessary since exploring and changing the timed event graph at the same time causes the algorithm to work improperly (exploring a partly changed timed event graph can lead to inconsistencies).
In line 1 the event x_2 corresponding to the next train is found by looking for headway constraints. Lines 2 – 33 contain the main loop of the algorithm. The new headway arcs, as explained in section 6.2.2, are constructed in lines 9, 16 and 23. There are two occasions in which the algorithm terminates before applying any changes to the timed event graph:

1. In line 18, when a headway arc from x_2 to x_1 is found (i.e. the running order is changed already).
2. In line 32, when an outgoing running or dwell arc is missing for x_1 or x_2, which means that line 1 or line 2 ends unexpectedly.

In such an occasion, *Warning* = 1 is returned, to indicate that the algorithm was unable to perform the requested dispatching action. In line 26, events x_1 and x_2 are stored in the two column matrix *H*, which will be used in the graph update part to generate new hindrance constraint arcs. In line 27, event x_1 is removed from the scheduled order of events, while the implied changes for the hindrance constraints are stored in line 28.
The 'graph update part' starts with carrying out all changes calculated in the 'explore part'. The necessary changes are performed in lines 35 and 37. After this, the structure of the timed event graph is as follows:

- The topology of headway constraint arcs corresponds to the new situation (i.e. the events of line 2 occur *after* the events of line 1).
- The topology of the hindrance constraint arcs corresponds to the situation in which the events of line 2 are deleted from the scheduled order of events.

To complete the new timed event graph, the events of line 2 have to be inserted in the scheduled order of events again. This is done in line 39, where the events of line 2, which were stored in matrix *H* during the exploration phase, are inserted in the linked lists directly subsequent to the events of line 1. The implied changes of the hindrance constraint arcs are carried out in lines 41 and 43. Note that the routines *arcs2add* and *arcs2del* are presented in sections 5.2.5 and 5.2.6.

Algorithm 6.3 (CHANGEORDER)

Input:

A	= arclist
Adjhead	= adjacency lists for heads
Adjtail	= adjacency lists for tails
Event	= list of events
Hindrance	= lists of hindrance constraints between train lines
x_1	= starting event
n	= number of IO-points where the order has to be changed
Empty	= list of empty rows in arclist *A*

Output:

A	= updated arclist
Adjhead	= updated adjacency list for heads
Adjtail	= updated adjacency list for tails
Event	= updated list of events

Empty = updated list of empty rows in A
Warning = 0 if dispatching action is possible, 1 = dispatching action is impossible

find event x_2 of the next train scheduled behind x_1 ; % explore part
$m \leftarrow 0$;
while $m < n$ **do** % main loop
  $m \leftarrow m + 1$;
  for each outgoing arc r of x_1 **do**
    if $type_r$ = headway **then**
      if successor event $j_r = x_2$ **then**
        $arcs2del \leftarrow arcs2del \cup r$;
        $arcs2add \leftarrow arcs2add \cup (x_1 , x_2 , t_r , type_r)$; % swap x_1 , x_2 (constr. rule 1.2)
    elseif $type_r$ = running or $type_r$ = dwell **then**
      $x_1^{next} \leftarrow$ successor event j_r ; % store next event of train line 1
  for each incoming arc r of x_1 **do**
    if $type_r$ = headway
      if preceding event $i_r \neq x_1$ **then**
        $arcs2del \leftarrow arcs2del \cup r$;
        $arcs2add \leftarrow arcs2add \cup (x_2 , i_r , t_r , type_r)$; % construction rule 1.1
      else
        $warning \leftarrow 1$; **return** % running order is changed already
  for each outgoing arc r of x_2 **do**
    if $type_r$ = headway **then**
      if successor event $j_r \neq x_1$ **then**
        $arcs2del \leftarrow arcs2del \cup r$;
        $arcs2add \leftarrow arcs2add \cup (j_r , x_1 , t_r , type_r)$; % see constr. rule 1.3
    elseif $type_r$ = running **or** $type_r$ = dwell **then**
      $x_2^{next} \leftarrow$ successor event j_r ; % store next event of train line 2
  $H \leftarrow H \cup \{ x_1 , x_2 \}$;
  [*add _temp, del_temp , Event*] $\leftarrow$ Delevent_Hindrance(x_1) ;
  $arcs2add \leftarrow arcs2add \cup add\_temp$; $arcs2del \leftarrow arcs2del \cup del\_temp$;
  if x_1^{next} exists **and** x_2^{next} exists
    $x_1 \leftarrow x_1^{next}$; $x_2 \leftarrow x_2^{next}$;
  else
    $Warning \leftarrow 1$; **return**
end
for each arc $r \in arcs2del$ **do** % graph update part
  (A , *Adjhead* , *Aadjtail* , *empty*) $\leftarrow$ delarc (A , *Adjhead* , *Adjtail* , *empty* , r) ;
for each row $r \in arcs2add$ **do**
  (A , *Adjhead* , *Adjtail* , *empty*) $\leftarrow$ addarc (A , *Adjhead* , *Adjtail* , *empty* , r) ;
for each row $\{x_1 , x_2\} \in H$
  [*arcs2add* , *arcs2del* , *Event*] $\leftarrow$ Insertevent_Hindrance(x_1 , x_2) ;
  for each arc $r \in arcs2del$ **do** % graph update part
    (A , *Adjhead* , *Aadjtail* , *empty*) $\leftarrow$ delarc (A , *Adjhead* , *Adjtail* , *empty* , r) ;
  for each row $r \in arcs2add$ **do**
    (A , *Adjhead* , *Adjtail* , *empty*) $\leftarrow$ addarc (A , *Adjhead* , *Adjtail* , *empty* , r) ;
return

6.3 Postponing arrivals or departures at a station

At stations where hindrance conflicts can occur, hindrance constraints ensure that conflicting train movements are separated in time. When the scheduled order of events would be strictly maintained in case of delays, those delays can easily be passed on to crossing train movements waiting for the delayed train. In such cases postponing the delayed event is an effective dispatching action to reduce the amount of knock-on delays. To find the optimal postponing actions, the algorithm 'Postpone', presented in this section, has been developed.

6.3.1 The definition of postponing in this project

An important difference with the algorithm 'Change Order' is that the sequence orders of trains at the *open track*, and thus the headway constraints, stay the same. Only the order in which trains enter and/or leave the *timetable points* is changed. This implies that only the hindrance constraints are changed when postponing an event. In the remainder of this thesis, 'postponing' will be defined as follows:

Event x *is called 'postponed' when it is moved ahead in the scheduled order of events of a timetable point without implications for the headway constraints.*

6.3.2 Check if postponing is possible

The developed algorithm postpones an event x_1 by moving it one line ahead in the linked list of scheduled events, which means that event x_1 is moved immediately behind the next event x_2. The first question that has to be answered before the algorithm can carry out the actual postponing action is: can event x_1 be postponed after event x_2 at all? Postponing event x_1 is not possible in the two following situations:

1. Event x_1 is the last event scheduled at this timetable point. Obviously, it cannot be postponed when this is the case.
2. Events x_1 and x_2 are using the same IO-point. This implies that the order of trains at the open track has to be changed when postponing event x_1.

When postponing is possible, two situations can occur: a situation in which hindrance constraints are changing, and a situation in which they are not. This will be explained in the next sections.

6.3.3 Situation without changing hindrance constraints

Postponing an event does not necessarily mean that hindrance constraints have to be changed. Hindrance constraints do not change when events x_1 and x_2 are not conflicting. Consider for example the situation shown in Figure 6.9. Suppose that line 2 has a delay, and that the departure of line 2 has to be postponed after the departure of line 3. As can be seen, lines 2 and 3 only have hindrance conflicts with line 1, but not with *each other*. As a result, the scheduled order of line 2 and 3 can be changed without any implications for the timed event graph. When this is the case, only the linked list containing the scheduled order of events has to be changed.

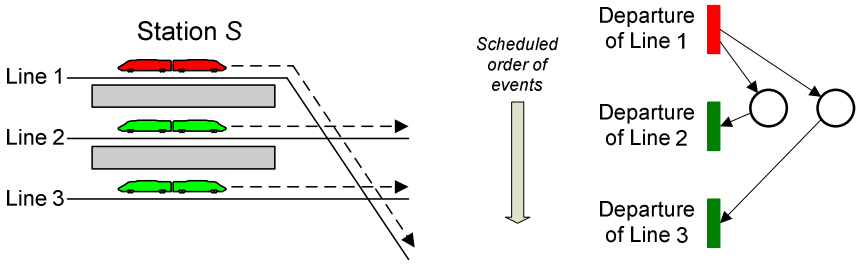


Figure 6.9 *Hindrance constraints at station S.*

6.3.4 Construction rule for changing hindrance constraints

When event x_1 has a hindrance conflict with x_2, postponing event x_1 implies changes in the timed event graph. How the new hindrance constraint arcs are generated is explained using Figure 6.10, where a station with two conflicting hourly train lines is shown. In the original order, the successive departures of lines 1 and 2 can be seen, with their hindrance constraints arcs connecting them. Suppose that line 1 is delayed (the red event in the figure) and will be postponed.

First, the old hindrance constraint arcs have to be deleted, after which three new hindrance constraint arcs have to be generated:

1. Hindrance constraint from event x_2 to event x_1.
2. Hindrance constraint from event x_1 to the next scheduled event with the same line number and the same IO-point as x_2.
3. Hindrance constraint from the preceding scheduled event with the same line number and the same IO-point as x_1 to event x_2.

However, numbers 2 and 3 are not necessary to construct when time separation is already assured by headway constraints. This is illustrated in the case of Figure 6.10. In the new order of events (after postponing the departure of line 1) two departures of line 2 are scheduled subsequent to each other. Since two departures of the same train line use the same IO-point, these events are separated by headway constraints anyway, and hindrance constraints are not necessary. The same is applicable to the two subsequent departures of line 1. The only new hindrance arc that has to be generated is shown on the right in Figure 6.10.

The necessary changes are summarized in construction rule 4.

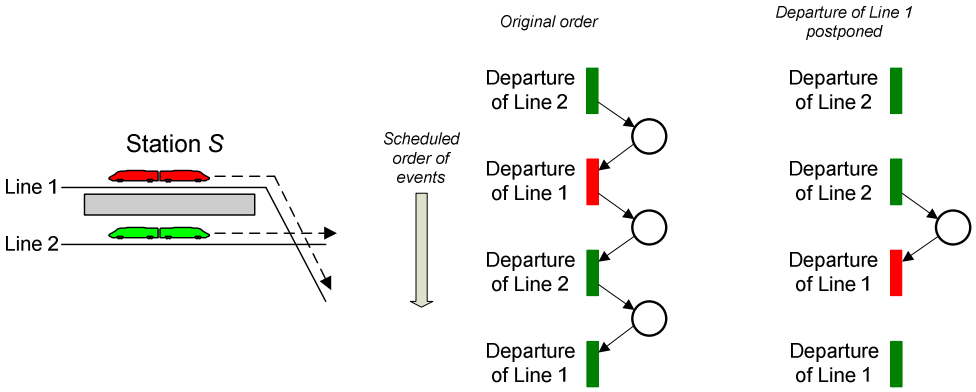


Figure 6.10 *Two lines with hindrance constraints in original order and when line 1 is postponed.*

Construction rule 4
Given: - event x_2 scheduled subsequent to event x_1.
- events x_1 and x_2 have a hindrance conflict

When postponing x_1 after x_2 three new hindrance constraint arcs have to be constructed:
1. from x_2 to x_1.
2. from x_1 to the next scheduled event N such that $IO_N = IO_2$ and $LN_N = LN_2$ unless an event Q scheduled between x_1 and N exists such that $IO_Q = IO_1$ and $LN_Q = LN_1$.
3. to x_2 from the previously scheduled event P such that $IO_P = IO_1$ and $LN_P = LN_1$ unless an event Q scheduled between x_2 and P exists such that $IO_Q = IO_2$ and $LN_Q = LN_2$.

6.3.5 The algorithm 'Postpone'

Algorithm 6.4 implements the postponement of one event according to construction rule 4 in the timed event graph, and works as follows:
In lines 1-5 is checked whether the postponement is actually possible (see section 6.3.2). Event *x* is the event that will be postponed, while event *E* is the first event scheduled subsequent to *x*. In line 6, the linked list is updated using the function 'ChangeList', which can be found in appendix 1. Whether events *x* and *E* are conflicting is checked in lines 7 and 8. In case of a conflict the hindrance arcs relating to this conflict have to be updated and lines 9 – 39 are executed. In lines 9 – 30 the relevant hindrance arcs are deleted. In line 31, the hindrance arc from *E* to *x* is constructed. In lines 32 – 34 a new hindrance arc from a previous train to event *E* is constructed when necessary according to construction rule 2, line 3, and in lines 35 – 37, a new hindrance arc from event *x* to a future train is constructed when necessary according to construction rule 2, line 2 (see section 6.3.4). Finally, the generated hindrance arcs are inserted in the timed event graph in lines 38 and 39.

Algorithm 6.4 (POSTPONE)

Input:
A = arclist
adjhead = adjacency lists for heads
adjtail = adjacency lists for tails
Event = list of events
Hindrance = lists of hindrance constraints between train lines
x = event that will be postponed
Empty = list of empty rows in arclist *A*

Output:
A = updated delay matrix
adjhead = updated adjacency list for heads
adjtail = updated adjacency list for tails
Event = updated list of events
Empty = updated list of empty rows in *A*
Warning = 0 if dispatching action is possible, 1 = dispatching action is impossible

```
1.  E ← next event scheduled after x
2.  if E exists = 0
3.      Warning = 1 ; return                    % no next event, so postpone is impossible
4.  if IO_E = IO_x
5.      Warning = 1 ; return                    % same IO-point, postpone implies order change
6.  Event = changeList (Event , x , E) ;        % change order in linked list
```

7. find k such that $Hindrance\{LN_x , TTP_x , IO_x\}(k) = (LN_E , IO_E , t)$;
8. **if** k exists
9. **for** each outgoing arc m of x **do**
10. **if** $type_m$ = hindrance **then**
11. $j \leftarrow$ successor event j_m ;
12. **if** $LN_j = LN_E$ **then** % j conflicts with x
13. $arcs2del \leftarrow arcs2del \cup k$;
14. **for** each incoming arc m of x **do**
15. **if** $type_m$ = hindrance **then**
16. $i \leftarrow$ preceding event i_m ;
17. **if** $LN_i = LN_E$ **then** % i conflicts with x
18. $arcs2del \leftarrow arcs2del \cup k$;
19. **for** each outgoing arc m of E **do**
20. **if** $type_m$ = hindrance **then**
21. $j \leftarrow$ successor event j_m ;
22. **if** $LN_j = LN_x$ **then** % j conflicts with E
23. $arcs2del \leftarrow arcs2del \cup k$;
24. **for** each incoming arc m of E **do**
25. **if** $type_m$ = hindrance **then**
26. $i \leftarrow$ preceding event i_m ;
27. **if** $LN_i = LN_x$ **then** % i conflicts with E
28. $arcs2del \leftarrow arcs2del \cup k$;
29. **for** each arc $k \in arcs2del$ **do**
30. (A , $adjhead$, $adjtail$, $empty$) $\leftarrow$ delarc (A , $adjhead$, $adjtail$, $empty$, k) ;
31. $arcs2add \leftarrow arcs2add \cup (x , E , t , 5)$; % hindrance arc from E to x
32. find event P preceding E such that $IO_p = IO_E$ & $LN_P = LN_E$;
33. **if** no event Q such that $IO_Q = IO_x$ & $LN_Q = LN_x$ is found sooner **do**
34. $arcs2add \leftarrow arcs2add \cup (E , P , t ,$ 'hindrance') ; % hindrance arc from P to E
35. find event N scheduled later than x such that $IO_N = IO_x$ & $LN_N = LN_x$;
36. **if** no event Q such that $IO_Q = IO_E$ & $LN_Q = LN_E$ is found sooner **do**
37. $arcs2add \leftarrow arcs2add \cup (N , x , t ,$ 'hindrance') ; % hindrance arc from x to N
38. **for** each row $k \in arcs2add$ **do**
39. (A , $adjhead$, $adjtail$, $empty$) $\leftarrow$ addarc (A , $adjhead$, $adjtail$, $empty$, k) ;

6.4 Short turning

Short turning can be a useful dispatching action in case of big delays or obstruction of the railway line. To illustrate this dispatching action, a turning train is visualized in Figure 6.11. Suppose train 1 has a big delay, or the railway line between stations 1 and 2 is (temporarily) obstructed. Letting train 1 turn already in station 1 yields the situation shown in Figure 6.12. Carrying out this dispatching action implies cancelling (i.e. deleting from the timed event graph) four events, namely the departure of train 1 from station 1, the arrival at station 2, the departure of train 2 from station 2 and its arrival at station 1. In the next section, the implications on the timed event graph when cancelling an event are explained. This is used in the subsequent section to develop an algorithm for short turning.

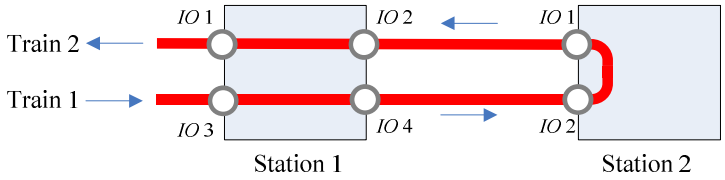


Figure 6.11 *Rolling stock of train 1 running back as train 2 after turning at station 2.*

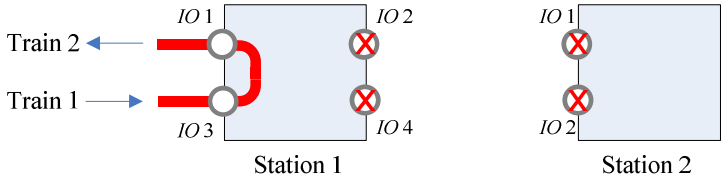


Figure 6.12 *Turn at station 2 moved to station 1 with four deleted events.*

6.4.1 Cancelling an event

Cancelling an event has implications for all types of arcs in the timed event graph. Obviously, running-, trough- and dwell time arcs to and from the event have to be deleted. However, the implications on headway and hindrance constraint arcs are more complicated. In this section, the implications will be investigated, after which the algorithm 'Cancel Event', able to implement these implications in the timed event graph, will be presented.

Implications on headway constraints

The headway constraints have to be changed such that the time separation between the preceding train and the successive train is still ensured. This is illustrated in Figure 6.13. Construction rule 5 describes the necessary changes in the timed event graph.

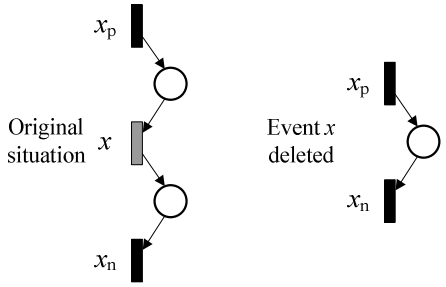


Figure 6.13 *New headway constraint when an event* x *is deleted.*

Construction rule 5

Given: - event x scheduled between its preceding event x_p and its successor event x_n at the same IO-point.

When deleting event x from the timed event graph, the headway arcs have to be changed as follows:

1. delete arc from x_p to x
2. delete arc from x to x_n
3. create new arc from x_p to x_n

Implications on hindrance constraints
When cancelling part of a train trip due the introduction of a short turn, the events modelling this part of the train trip are simply removed from the scheduled order of events. An algorithm that determines the implications on the timed event graph due to this operation has already been discussed in section 6.2.3. This algorithm will be used to determine the implications for hindrance constraints in case of short turning as well.

6.4.2 The algorithm 'ShortTurn'

The algorithm 'ShortTurn' is able to change the timed event graph according to the situation that a train turns earlier than scheduled. The input consists of the timed event graph, represented by the variables *A*, *Adjhead*, *Adjtail*, and *Event*. The vector *Empty*, needed for the algorithms 'Delarc' and 'Addarc', for updating the graph, is input as well. The location of the new turn is determined by *x*, which is the arrival event at the new turning station. In the example of Figure 6.11 and Figure 6.12 this would be the arrival of train 1 at station 1. The output is an updated timed event graph and the matrix *Cancelled*, containing the numbers of all cancelled events, as well as the numbers of the next scheduled events of the same train line. This is used for calculating the passenger delay, since passengers have to wait for the next train if their train is cancelled. The algorithm works as follows:
In line 1, the next event of the train trip is found. In Figure 6.11 this would be the departure from station 1. The number of this event is assigned to the variable *i*. The dwell or through arc running to this event has to be deleted, so it is stored in the vector *Arcs2del* for later deletion. In line 5, the main loop starts. This loop 'walks' along the train trips until the algorithm arrives at the new turning station again, which is checked in line 7. In Figure 6.11 this would correspond to walking via the turn at station 2 until the arrival of train 2 at station 1. On its way along the train trip, the algorithm stores all headway constraint arcs in the vector *Arcs2del* for later deletion according to construction rule 5.1 and 5.2. The new headway arc between the previous and the next train according to construction rule 5.3 is stored in line 22. The arcs of which the train trip itself consists (i.e. running, dwell, through or turn arcs) are deleted as well. Hindrance constraint arcs are investigated in line 23, where the algorithm Delevent_Hindrance, presented in section 6.2.3, is called.
In line 25, the algorithm continues to the next event, which was found while scanning the outgoing arcs (in line 16). Upon arrival at the new turning station, the loop ends and the algorithm continues with lines 27 and 28. A new turn arc, connecting event *x* with the departure after the newly scheduled turn, is created in line 27. The minimal duration of the new turn t_{turn} is the same as the turning time of the original turn, which was retrieved from the timed event graph in line 13. In line 28 the timed event graph is actually updated, using the algorithms presented in sections 5.2.5 and 5.2.6.

Limitations of the algorithm
Two limitations of this algorithm deserve attention. The first limitation concerns the turning time t_{turn} used for the new turn arc. In the real railway system, this time would be called the 'minimum layover time', which is the minimal time that has to elapse between the arrival and the departure at the turning station. In the presented algorithm, the minimum layover time of the original turn is used as t_{turn} for the new turn (e.g. if the turn at station 2 in Figure 6.11 would take at least 10 minutes, then the new turn in Figure 6.12 will take at least 10 minutes as well). The limitation is that in reality the minimum layover time can differ in case of dispatching actions. For example, the minimum layover time in the original schedule can be relatively long (e.g. if the train has to be cleaned, etc.), where this is not always necessary in case of dispatching actions (e.g. if cleaning the trains has a lower priority in case of delays). Still, using the same minimum layover

time for the new turn is considered realistic, since most factors affecting it remain the same in case of dispatching actions. These factors are for example the length of the train (if the train driver has to walk back to the other cab), the rolling stock (whether it takes some time to make the train ready for a trip in the opposite direction), etc.

Algorithm 6.5 (SHORTTURN)

Input:
A = arclist
Adjhead = adjacency list for heads
Adjtail = adjacency list for tails
Event = list of events
Empty = list of empty rows in *A*
x = arrival event where the new turn will take place

Output:
A = updated arclist
Adjhead = updated adjacency list for heads
Adjtail = updated adjacency list for tails
Event = updated list of events
Empty = list of empty rows in *A*
Cancelled = list of cancelled events and next scheduled events of same train line

1. find outgoing arc *r* of *x* such that $type_r$ = 'dwell' or 'through' ;
2. *i* ← successor event *i* of arc *r* ;
3. *Arcs2del* ← *Arcs2del* ∪ *r* ;
4. *continue* ← 1 ;
5. **while** *continue* **do**
6. *Cancelled* ← *Cancelled* ∪ {*i* , next event *n* at IO_n such that $LN_n = LN_i$} ;
7. **if** $type_i$ = 'arrival' & $TTP_i = TTP_x$ **then** % arrival at new turning station
8. *continue* ← 0 ;
9. **for** each outgoing arc *r* of *i* **do** % scan outgoing arcs
10. **if** $type_r$ = 'headway' **then**
11. *Next* ← successor event of arc *r* ; *Arcs2del* ← *Arcs2del* ∪ *r* ;
12. **elseif** $type_r$ = 'turn' **then**
13. *successor* ← successor event of arc *r* ; $t_{turn} \leftarrow t_r$; % store min. duration of turn
14. *Arcs2del* ← *Arcs2del* ∪ *r* ;
15. **elseif** $type_r \neq$ 'hindrance' **then**
16. *successor* ← successor event of arc *r* ;
17. **for** each incoming arc *r* of *i* **do** % scan incoming arcs
18. **if** $type_r$ = 'headway' **then**
19. *Arcs2del* ← *Arcs2del* ∪ *r* ;
20. *Previous* ← preceding event of arc *r* ;
21. **if** *Next* exists & *Previous* exists **then**
22. *Arcs2add* ← {*Next*, *Previous*, *t*, 'headway'} ; % construction rule 5.3
23. [*add _temp*, *del_temp* , *Event*] ← Delevent_Hindrance(x_1) ;
24. *arcs2add* ← *arcs2add* ∪ *add _temp* ; *arcs2del* ← *arcs2del* ∪ *del _temp* ;
25. *i* ← *successor* ;
26. **end**
27. *Arcs2add* = *Arcs2add* ∪ {*i* , *x* , t_{turn} , 'turn'} ; % new turn arc
28. Delarc(*Arcs2del*) ; Addarc(*Arcs2add*) ; % update graph

The second limitation concerns the treatment of hindrance constraint arcs. The short turning manoeuvre introduced in the new turning station (e.g. station 1 in Figure 6.12) can lead to new hindrance conflicts, for instance if the turning train crosses some tracks used by other train lines when changing to a track for the opposite direction. Creating new hindrance constraints in this case is possible only with detailed knowledge about the topology of the station and the interlocking route of the turning manoeuvre, which is not included in the presented algorithm. Note that this is only the case for hindrance caused by the turning manoeuvre *itself*, the arrival and departure from the new turning station are still secured by all headway and hindrance constraints present in the model before the short turn was introduced. This is shown in Figure 6.14, which is based on the example presented in Figure 6.12.

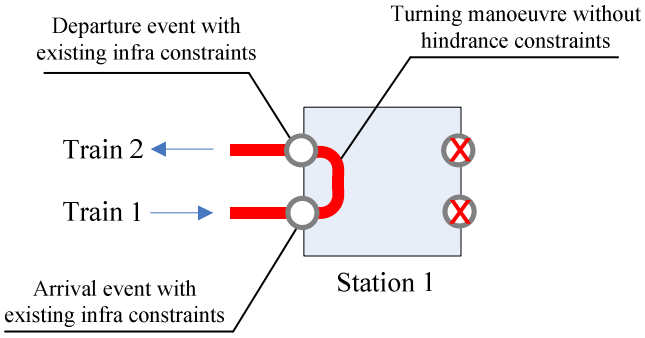


Figure 6.14 *Schematic representation of turning manoeuvre not secured by infra constraints.*

6.5 Optimization framework

In order to find effective dispatching actions, a greedy optimization algorithm has been implemented. The optimization algorithm calculates the effectiveness of each dispatching action from a list of possible dispatching actions created beforehand. In the end, the optimization algorithm returns the dispatching action with the greatest effectiveness. By applying this algorithm several times and storing each optimal dispatching action, a greedy approach yielding effective combinations of dispatching actions is obtained.

6.5.1 Dispatching actions have to be combined with postponements

Often, one dispatching action is not enough to reduce occurring delays. In such cases, a combination of different dispatching actions proves to be much more effective. In particular, a dispatching action is likely to be most effective when it is combined with one or more postponements of events from the delayed train trip. For example, suppose that train 1 has a delay, which is propagated to train 2, running behind it. In this case, changing the order between trains 1 and 2 seems to be an effective dispatching action. But suppose that train 1 has a hindrance conflict with train 3 as shown in Figure 6.15. When the order between trains 1 and 2 is changed, train 3 still has to wait for its departure until train 1 has passed. As a result, delays keep being propagated through the network and the effectiveness of a dispatching action is not maximized. Clearly, the solution to this problem is to let train 3 depart *before* train 1 has passed, i.e. the departure of train 1 has to be postponed (recall the definition of postponing posed in section 6.3.1. In that case, changing the order of trains 1 and 2 is effective indeed. As a consequence, the positive effect of each dispatching action has to be maximized by combining it with appropriate postponements. This has been implemented in the optimization algorithm, as will be described in the next sections.

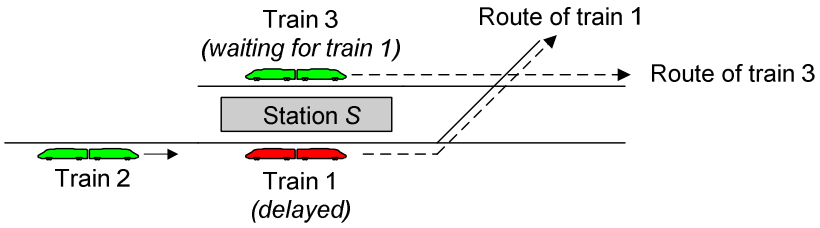


Figure 6.15 *Delayed trains at a station.*

6.5.2 Making an inventory of possible dispatching actions

Before the actual optimization process can start, an inventory has to be made of all dispatching actions to be taken into consideration. To this end, the algorithm 'Generate_Inventory' is used. The input for this algorithm consists of the timed event graph represented by the arclist and its adjacency lists, and the matrix *Event*. The output is a matrix *Choice* containing all possible dispatching actions recognised by the algorithm. This matrix is used as input for the optimization algorithm described in section 6.5.4. Each row of the matrix *Choice* describes a dispatching action as follows:
In case of an order change, a row *r* contains the following elements:

Choice(*r*) = {*type* , *x* , *amount*}

where:
type = 'OrderChange',
x = event from where the order change starts,
amount = amount of IO-points where the order is changed.

The variables *x* and *amount* are input for the ChangeOrder algorithm (see section 6.2.5) when implementing this dispatching action in the timed event graph.

In case of a short turn, a row *r* contains the following elements:

Choice(*r*) = {*type* , *x* }

where:
type = 'ShortTurn',
x = arrival event where the new turn will take place.

The variable *x* is input for the ShortTurn algorithm (see section 6.4.2) when implementing this dispatching action in the timed event graph.
The algorithm works by investigating the possibilities for dispatching actions for each event *x* separately. If event *x* is a departure of a start of a train trip, the possibilities for changing the sequence order are investigated in lines 2 – 11. This is done by following the (events of the) train trip through the graph (line 6). For each arrival at a station where overtaking is possible, the corresponding dispatching action is stored in the *Choice* matrix (line 11). If the train running behind the train under investigation ends or continues in a different direction, changing the order is not possible anymore. This is accounted for by the if-statement in line 10.
If event *x* is an arrival event at the end of a train trip, the possibility of short turning is investigated. If the train is scheduled to turn (which is checked in line 13), the train trip is tracked back through the timed event graph to search for short turning possibilities. Each arrival event at

a timetable point where turning is possible represents a possibility to short-turn a train, and is stored in the *Choice* matrix (line 16). Note that it is assumed that each train turns at a train with the same route (but obviously in the opposite direction), since short turning is impossible if a train continues at a different route after turning.

Algorithm 6.6 (GENERATE_INVENTORY)

Input:
Event = list of events
A = arclist
Adjhead = adjacency list for heads
Adjtail = adjacency list for tails
Output:
Choice = list of possible dispatching actions

```
1.  for each event x ∈ Event do
2.    if type_x = 'departure' or type_x = 'start of train trip' then        % explore order changes
3.      amount ← 0 ;
4.      find train number TN_next of the train scheduled to run subsequent to TN_x …
5.      … by exploring headway constraint arcs ;
6.      for each following arrival event a of train trip TN_x do
7.        amount ← amount + 2 ;
8.        if overtaking at TTP_a is possible
9.          find train number TN_next of the train scheduled to run subsequent to TN_x ;
10.         if TN_next is the same as in the previous step
11.           Choice ← Choice ∪ {'ChangeOrder' , x , amount} ;
12.   elseif type_x = 'end of train trip' then                             % explore short turns
13.     if an outgoing arc r exists such that type_r = 'turn' then
14.       for each preceding arrival event a of train trip TN_x do
15.         if short turning at TTP_a is possible
16.           Choice ← Choice ∪ {'ShortTurn' , a} ;
```

6.5.3 The objective function: total passenger delay

Many considerations play a role when choosing an objective function for the optimization process. In the optimization framework used in this project, the total passenger delay is used as an optimization function. In order to calculate the total passenger delay, a number of passengers is attached to each event in the railway network. For example the number of passengers attached to an arrival event x reflects the number of alighting passengers at TTP_x. The total passenger delay is then calculated by multiplying the delay vector created by the delay propagation algorithm with the vector containing the numbers of passengers attached to each event, as shown by the next formula:

$$Z\_total = Z * P' , \tag{6.1}$$

where:
Z_total = total passenger delay in minutes,
Z = delay vector,
P = vector containing the number of passengers attached to each event.

The main reason for choosing the total passenger delay as an objective function is that the impact of dispatching actions for the passengers is considered more important than the impact on the

trains themselves. For instance introducing a short turn may be very effective to restore the train service to the scheduled situation. For passengers however, this is an unpopular measure causing large passenger delays at the stations where the turning train is cancelled, particularly when the considered train line runs at a low frequency (e.g. hourly).
Since cancelled events are no longer part of the timed event graph, they are not visited by the delay propagation algorithm presented in section 5.4 anymore. Each cancelled event x is registered in the matrix *Cancelled*, along with the next scheduled event of the same train line n. Before formula 6.1 can be used, the delays of all cancelled events $x \in$ *Cancelled* are calculated as follows:

$$Z(x) = d(n) - d(x) + Z(n) \tag{6.2}$$

where:
$Z(x)$ = delay of a cancelled event x,
d = timetable vector,
n = next event scheduled at IO_x such that $LN_n = LN_x$.

This way of calculating the passenger delay incorporates:

- Arrival delays of alighting passengers,
- Departure delays of embarking passengers,
- Delays of passengers who have to wait for the next train since their train is cancelled due to short turning.

This is sufficient for a correct evaluation of the dispatching actions considered in this thesis. However, more elaborate ways of calculating passenger delays exist, for instance including origin/destination matrices for an accurate calculation of the passenger delays caused by broken transfers, etc, a description of which can be found in [10].

6.5.4 Finding the most effective dispatching action

Figure 6.16 shows a flow diagram explaining the optimization algorithm 'Dispatch_optimal', which evaluates all dispatching actions from the list of possible dispatching actions created by the algorithm 'Generate Inventory'. After calling the algorithm, a delay vector is generated from the initial delays (the initial delays are input data). This vector is used to calculate the delay propagation through the network in the situation without any dispatching actions. During the calculation of the delay propagation, a list of delayed trains is produced. After these initial steps, the actual loop starts. To speed up the algorithm, dispatching actions regarding on-time trains are omitted. That is why in the loop each dispatching action is checked for being associated with a delayed train.

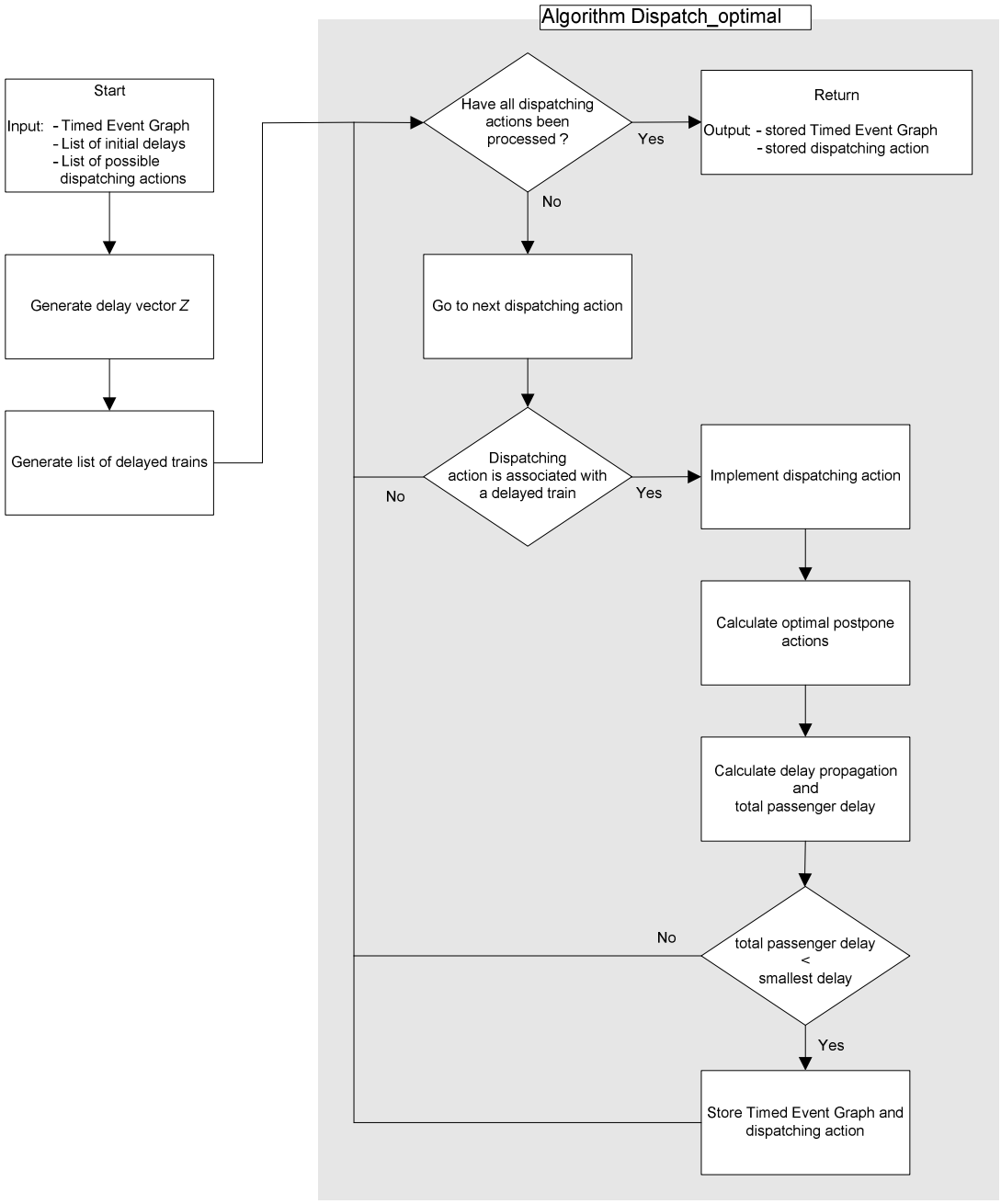


Figure 6.16 *Diagram of algorithm 'Dispatch_optimal' to find the most effective dispatching action.*

When a dispatching action relates to a delayed train, the dispatching action is implemented, using one of the algorithms presented in this chapter. Using the changed timed event graph, the optimal postpone actions are calculated. After implementing the dispatching actions, the delay propagation is calculated. Finally, the calculated delay is compared with the smallest delay found so far. If the calculated delay is smaller, the timed event graph and the performed dispatching actions are stored, after which the loop continues. When no more dispatching actions are available for testing, the loop ends and the most effective dispatching action is returned, along with the changed timed event graph.

This algorithm yields only one optimal dispatching action. In order to get a set of dispatching actions forming a complete tactic to revert to the scheduled situation, several iterations have to be done, which is implemented in the algorithm ‘Dispatch’. The adapted timed event graph according to the dispatching action of iteration one is then used as input for iteration two, etc. A flow diagram of this algorithm is shown in Figure 6.17.

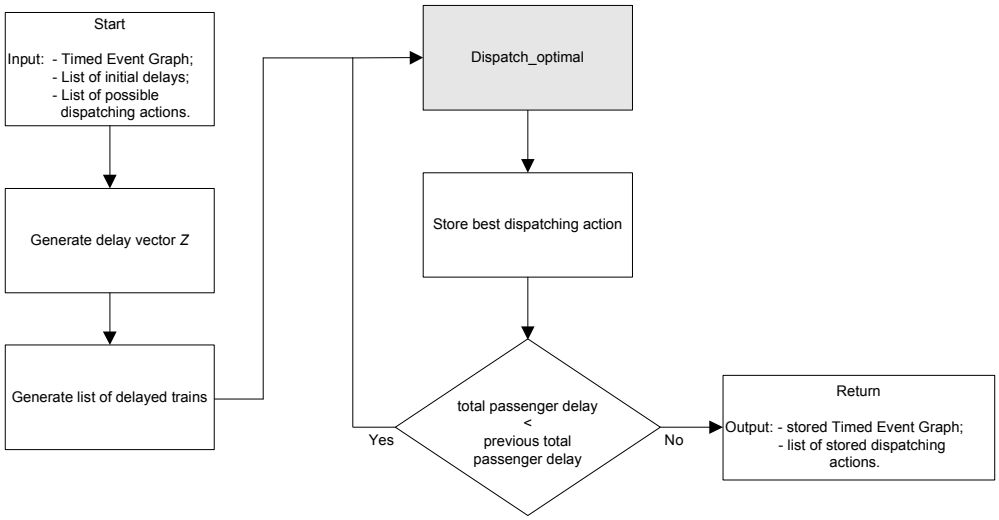


Figure 6.17 *Flow diagram of the algorithm 'Dispatch', calculating a combination of dispatching actions.*

Note that this algorithm contains similarities to a greedy algorithm, as the most effective dispatching actions are stored during the iterative process. Greedy algorithms are used frequently in the literature to solve similar problems, see for instance [5], where a greedy algorithm is used to find effective dispatching actions in a model using the specific properties of an alternative graph formulation. However, the philosophy behind the approach proposed in [5] is aimed at detailed modelling of railway operations (each block signal is modelled), used for calculating accurate speed profiles of the trains. This differs from the idea behind this research project, which is aimed at a quick, network wide, evaluation of the impact of dispatching actions.

Different approaches can be evaluated

The output of the algorithm described here consists of one combination of dispatching actions found during the optimization process. In practice however, situations can occur in which a dispatcher is interested in comparing different approaches to react on the given delays in a network. Particularly with respect to the dispatching actions investigated in this thesis, it may be interesting to compare an approach in which short turning is allowed with another approach in which short turning is not considered as an option to restore a delayed train service. This can be implemented by running the algorithm ‘Dispatch’ twice, where the second time a modified version of the *Choice* matrix, from which the dispatching action ‘short turning’ has been removed, is used. If this case, the dispatcher gets an overview of both options, and he can choose which approach to follow depending on the situation.

Limitations of this optimization approach

Some limitations to the described optimization approach deserve attention. The most important limitation is that the possibility to carry out dispatching actions regarding on-time trains is omitted during the optimization process. The choice for this methodology is based on the fact that all dispatching actions regarded in this thesis (change order, postpone and short turning) are aimed at reducing the delay propagation caused by a delayed train. Hence it is logical, and in

most cases effective, to try applying these dispatching actions to delayed trains only. However, situations in which it is effective to apply one of the aforementioned dispatching actions to an on time train can theoretically exist, which could theoretically lead to a sub-optimal solution. Nevertheless, it is highly unlikely that postponing an on-time train is an effective dispatching action, so it is assumed that this limitation of the algorithm will not lead to a considerable discrepancy between the dispatching actions advised by the algorithm and the dispatching actions that are advisable in reality.
Another limitation is related to the complexity of the problem that the optimization algorithm has to solve. Even in small networks an enormous combination of dispatching actions is theoretically possible. For large scale networks, this leads to a combinatorial explosion of the solutions space for the problem. It is therefore of importance to develop intelligent optimization strategies for finding effective combinations of dispatching actions. Some ideas for this will be discussed in section 8.3.

6.6 Dispatching actions in max-plus notation

Timed event graphs can be translated into systems of max-plus equations and vice versa. Since most theory, algorithms and literature about modelling a railway system as a discrete event system makes use of the max-plus notation, the notations of a switching max-plus system (i.e. with the possibility to represent dispatching actions) will be briefly discussed in this section.

6.6.1 Max-plus algebra: definitions

The following definitions of max-plus algebra were published in [3], while a more detailed description of max-plus algebra for railways can be found in [7].
First, the set of real numbers is extended for use with max-plus algebra. Define:

$$\varepsilon = -\infty$$
$$\mathbb{R}_\varepsilon = \mathbb{R} \cup \{\varepsilon\}$$

The two basic operations in max-plus algebra are the max-plus algebraic addition and multiplication, which are defined as follows for numbers $x, y \in \mathbb{R}_\varepsilon$:

max-plus addition: $x \oplus y = \max(x, y)$,
max-plus multiplication: $x \otimes y = x + y$.

Matrix operations are defined as follows for matrices $A, B \in \mathbb{R}_\varepsilon^{m \times n}$, $C \in \mathbb{R}_\varepsilon^{n \times p}$ and vector $x \in \mathbb{R}_\varepsilon^{n}$:

max-plus addition: $[A \oplus B]_{ij} = a_{ij} \oplus b_{ij} = \max(a_{ij}, b_{ij})$,

matrix – vector multiplication: $[A \otimes x]_i = \bigoplus_{k=1}^{n} a_{ik} \otimes x_k = \max_{k=1,\ldots,n} (a_{ik} + x_k)$,

max-plus multiplication: $[A \otimes C]_{ij} = \bigoplus_{k=1}^{n} a_{ik} \otimes c_{kj} = \max_{k=1,\ldots,n} (a_{ik} + c_{kj})$.

6.6.2 Max-plus linear systems

The dynamics of a discrete-event dynamic system (such as a railway system), modelled so far by timed event graphs, can be described by recursive equations in max-plus algebra [7]. These equations can be put together to a max-plus linear system. First, the following variables are defined:

$x(k)$ = The *state vector*, where $x_i(k)$ denotes the actual time at which event i occurs for the k-th time (recall that a periodic timetable is concerned).

A = The *system matrix*, where $[A]_{ij}$ denotes the time that has to elapse after the occurrence of event j before event i can occur. In the timed event graph, this is equivalent to the arc weight of the arc originating from event j and ending at event i (recall the firing rule for events, explained in section 4.4.2). If no arc (j, i) exists, then $[A]_{ij} = \varepsilon$. Since the timetable is periodic, A is the same for each period k.

$d(k)$ = The *timetable vector*, where $d_i(k)$ denotes the scheduled time for the k-th occurrence of event i.

A *homogeneous* max-plus linear system has no inputs from a timetable. The recursive state equation, with which the event times in period k can be calculated when the event times in period $k-1$ are known, is:

$$x(k) = A \otimes x(k-1), \qquad x(0) = x_0, \tag{6.3}$$

where x_0 is the initial state vector, containing the event times of the first occurrences of all events. Since the timetable is not included in the system, its dynamic behaviour depends on the eigenvalues of A and the initial state vector x_0. In the real railway system, this can be viewed as the situation that each train would depart and arrive as early as possible in the scheduled order, instead of waiting for their scheduled departure times.
Of course, in a scheduled railway system, each train runs according to a timetable. The timetable is introduced in the model as a timetable vector $d(k)$. The periodic timetable with cycle time T (which usually equals 60 minutes) is modelled by:

$$d(k) = d(k-1) \otimes T, \qquad d(0) = d_0, \tag{6.4}$$

where d_0 is an initial timetable vector. A scheduled railway system can be represented by a *scheduled max-plus linear system*, according to the following recursive equation:

$$x(k) = A \otimes x(k-1) \oplus d(k). \tag{6.5}$$

6.6.3 The switching max-plus system

In a *switching* max-plus system, the system matrix A can be different for each period. In the preceding sections, algorithms for changing the timed event graph according to a given dispatching action are discussed. As a timed event graph is equivalent to a max-plus system, the algorithms can in fact be used to adapt the system matrix A, given a set of dispatching actions. The ability to change the system matrix can be included in equation (6.5) by introducing a new input variable $u(k)$, representing the possible changes that can be carried out in A. This can for example be implemented as a binary vector with length n, where n is the number of possible dispatching actions. In that case, $u_r(k) = 1$ means that dispatching action r is implemented in

period k. Since dispatching actions affect the system matrix, A becomes dependent of u. When writing $k + 1$ instead of k, this yields:

$$x(k+1) = A(k, u(k)) \otimes x(k) \oplus d(k+1). \tag{6.6}$$

Note that some dispatching actions could also affect the timetable vector (for example when changing a dwell time). However, in this project dispatching actions affecting the timetable vector are not considered, since the goal of carrying out the dispatching actions is to revert to the original timetable.
Since dispatching actions are carried out with the purpose of reducing the delays in the system, a delay vector $z(k)$ is introduced. The delay vector $z(k)$ is defined as the difference between the scheduled event time and the actual event time, and is never smaller than zero:

$$z(k) = (x(k) - d(k)). \tag{6.7}$$

Note that $z(k)$ cannot be negative, since in equation 6.6 the scheduled event times $d(k + 1)$ are added to the calculated event times $x(k + 1)$.
When no delays are present in the system the system matrix is not changed in that period, so:

$$z(k) = 0, \text{ which implies } u(k) = 0, \tag{6.8}$$

and therefore:

$$A(k, 0) = A. \tag{6.9}$$

6.7 Conclusion

This chapter contains the main result of this thesis, which is description of algorithms able to change the timed event graph according to given dispatching actions. These are:

- ChangeOrder, for changing the sequence order of two trains,
- Postpone, for postponing the arrival or departure of a train at a station with conflicting interlocking routes,
- ShortTurn, for applying a short turn.

Construction rules describing the necessary changes to reflect dispatching actions in the timed event graph have been described and implemented in these algorithms. The algorithms ChangeOrder and ShortTurn make use of two other algorithms for calculating the implications on the hindrance constraints when deleting or inserting an event in the scheduled order of events. Furthermore, a simple optimization algorithm using a greedy approach has been described, which will be used in the case study to calculate effective dispatching actions in a testing network. Finally, the possibility to apply dispatching actions in max-plus algebra has been formally described, yielding a switching max-plus system.
The consistency of the algorithms presented in this chapter will be tested in a case study, which is the subject of the next chapter.

7 Case study

7.1 Introduction

This chapter is dedicated to the application of the theory presented in the preceding chapters to a test case. In many fields of research, the test case is used for the calibration and/or validation of the model. However, since this thesis is aimed at showing how dispatching actions can be implemented in an *existing* model (i.e. the max-plus model), the purpose of the case study in this thesis is to illustrate how the algorithms can be used in practice, and to examine the consistency and plausibility of the results.
The outline of this chapter is as follows: In section 7.2 a fictive testing network with a timetable is presented. Section 7.3 explains the way in which this network is used to test the algorithms. In section 7.4 the results are presented and interpreted. Section 7.5 contains a brief conclusion.

7.2 Testing network

7.2.1 Considerations leading to the used testing network

In order to demonstrate all characteristics of the developed algorithms, a small testing network has been made up. The choice for a hypothetical network instead of (a part of) a real world railway network is based on the following considerations:

- The algorithm 'Postpone' can only be demonstrated when stations with hindrance constraints exist (e.g. stations with crossing interlocking routes). However, in the Dutch railway network such stations typically have a complicated track layout (i.e. Den Haag HS). Stations with a simple track layout are more suitable for easy interpretation of the test results.
- Modelling a real world network requires a thorough calculation of all headways, turning times, running times etc., which is considered not feasible for this research project.

A testing network is considered useful if it provides the possibilities to apply the presented algorithms in their full functionality, suggesting that the following elements have to be present in the test network:

- Trains with different operational speeds (i.e. intercity and local trains) are required since changing the sequence order of such trains is a particularly useful dispatching action.
- A rolling stock circulation modelled by trains turning at their terminal stations is needed to enable short-turning of trains.
- A station with crossing train lines is needed, in order to get hindrance constraints allowing the 'Postpone' algorithm to be useful.
- A longer railway line with one or more stations where it is possible to change the sequence order of trains is needed, so that different (combinations of) order changes can be tested.

Furthermore, the following characteristics for the timetable are required:

- The timetable has to be simple enough for easy interpretation of the test results.
- The timetable has to offer possibilities for dispatching actions, which implies that the traffic has to be dense enough to enable dispatching actions. However, the capacity consumption has to be low enough to offer the flexibility needed to implement dispatching actions.

- Just as in a real railway system, time supplements (i.e. slack time) have to be included in the timetable to enable small delays to settle without the need for dispatching actions.

7.2.2 The used testing network and timetable

Based on the considerations in the preceding section, the testing network shown in Figure 7.1 has been produced. As can be seen, stations 1, 6 and 7 form the end of a railway line. The trains turn here. The track layout of these stations is not shown. Stations 2, 4 and 5 are stations with overtaking possibilities along the railway line. Station 3, where two railway lines merge (without flyover) forms the heart of the testing network.

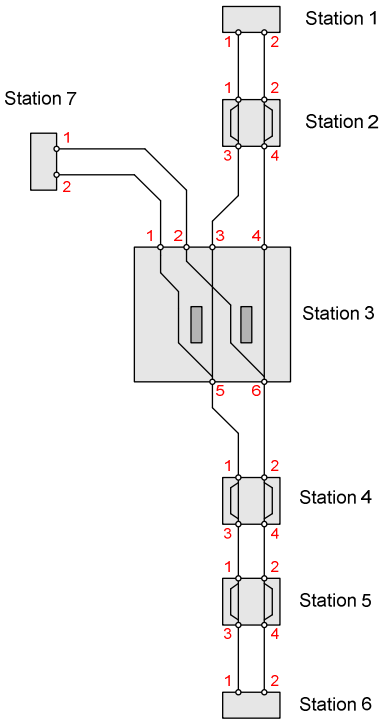


Figure 7.1 *Testing network for the case study.*

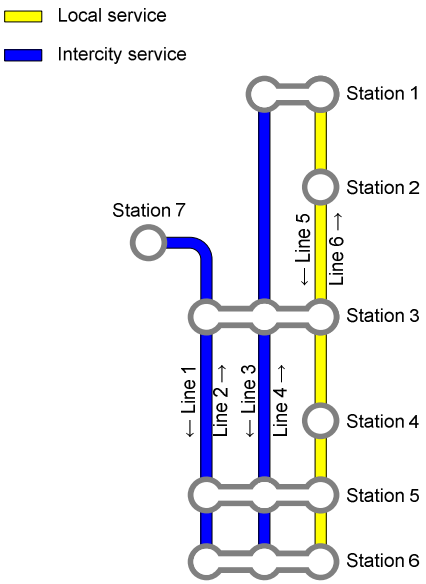


Figure 7.2 *Schematization of the timetable used for the test case.*

Figure 7.2 contains a schematization of the line plan of the test case. A thick line denotes a train line running in a 30 minutes service. One intercity line runs between stations 7 and 6, via station 3. Another intercity line connects station 1 with station 6. A local service runs between stations 1 and 6 as well. Note that all train lines share the same infrastructure between stations 3 and 6.
The hourly timetable of the train lines running in north – south direction are shown in Table 7.1, while the trains in the opposite direction can be found in Table 7.2. A through train is indicated by italic print of the departure time. In the testing timetable, each train line runs 6 times in total; hence, with a 30 minutes service on each line, approximately 3 hours are simulated.
A time-distance diagram of the corridor between stations 1 and 6 is shown in Figure 7.3. The individual train trips are numbered with three-digit train numbers, where the first digit corresponds with the line number. Note that in station 4, direction north – south, line 5 (local service) is scheduled to be overtaken by line 1 (intercity), and in direction south – north, line 6 (local service) is scheduled to be overtaken by line 2 (intercity). Note also that the local trains have longer scheduled running times than the intercity trains (i.e. the local trains are slower).

Table 7.1 *Timetables of trains in north - south direction in test network (d = departure, a = arrival).*

		Line 1: Intercity Station 7 – Station 6		**Line 3: Intercity** Station 1 – Station 6		**Line 5: Local** Station 1 – Station 6	
Station 1	d.			.15	.45	.21	.51
Station 2	a.			|	|	.28	.58
	d.			*.20*	*.50*	.29	.59
Station 7	d.	.00	.30	|	|	|	|
Station 3	a.	.11	.41	.26	.56	.36	.06
	d.	.13	.43	.28	.58	.38	.08
Station 4	a.	|	|	|	|	.45	.15
	d.	*.18*	*.48*	*.33*	*.03*	.51	.21
Station 5	a.	.28	.58	.43	.13	.04	.34
	d.	.30	.00	.45	.15	.05	.35
Station 6	a.	.36	.06	.51	.11	.13	.43

Table 7.2 *Timetables of trains in south - north direction in test network (d = departure, a = arrival).*

		Line 2: Intercity Station 6 – Station 7		**Line 4: Intercity** Station 6 – Station 1		**Line 6: Local** Station 6 – Station 1	
Station 6	d.	.16	.46	.01	.31	.09	.39
Station 5	a.	.22	.52	.07	.37	.17	.47
	d.	.24	.54	.09	.39	.18	.48
Station 4	a.	|	|	|	|	.31	.01
	d.	*.34*	*.04*	*.19*	*.49*	.37	.07
Station 3	a.	.39	.09	.24	.54	.44	.14
	d.	.41	.11	.26	.56	.46	.16
Station 7	a.	.52	.22	|	|	|	|
Station 2	a.			|	|	.53	.23
	d.			*.32*	*.02*	.54	.24
Station 1	a.			.38	.08	.01	.31

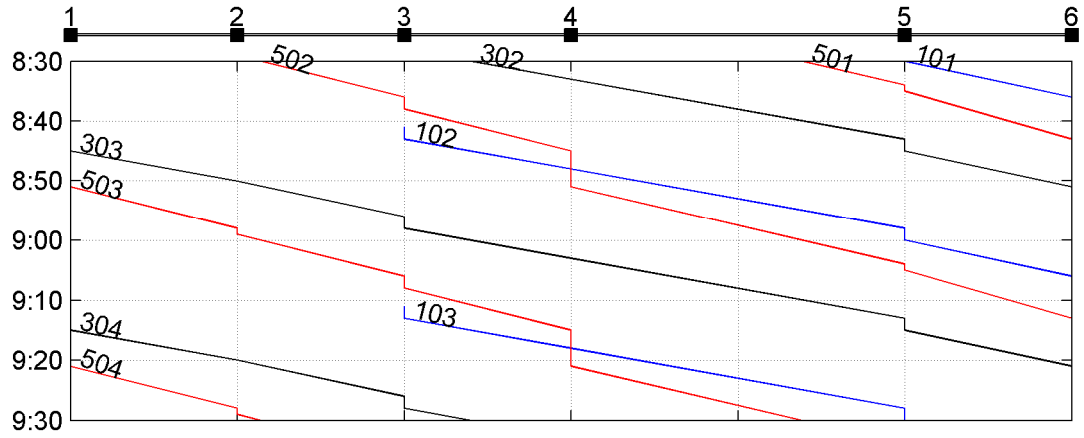


Figure 7.3 *Time-distance diagram of hourly schedule in test network. Local trains are coloured blue. For clarity, only one direction is shown. Note that the trains of line 1 merge with the corridor at station 3.*

7.3 Testing methodology

7.3.1 No real time simulation

Although the algorithms presented in this thesis are meant to operate in the real time environment of the dispatchers' office, no real time simulation is carried out for the test case. Instead, different delay scenarios are used as input, and the evaluation of different dispatching actions, leading to a (set of) dispatching actions yielding the best results is the output. This is schematized in Figure 7.4. The real time implementation of an evaluation tool for dispatching actions is a subject beyond the goal of this thesis. Furthermore, the output of a simulation of a real time environment would be less suitable for interpretation when one is mainly interested in the algorithms used for the *implementation* of dispatching actions in the max-plus model.

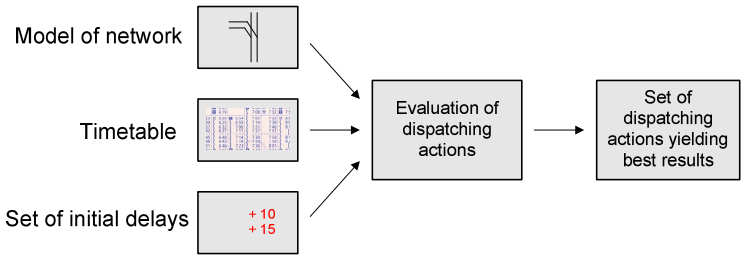


Figure 7.4 *Schematization of one test run.*

7.3.2 Input data

All variables as defined in chapter 5 (Data structure for timed event graphs) are needed for running the test case. This section explains how the required data is generated.

The matrix *Event*

The first 5 columns of the matrix *Event* are created before the actual timed event graph is generated. For each train line 1…6 all events of one train trip are created by hand, after which a small script is used to multiply the result six times (recall from the previous section that each train line runs 6 times in the testing timetable, and recall from section 4.6 that the algorithms only work in an acyclic model). As an example, the manual input for train line 5 is shown in Table 7.3. When multiplying this data 6 times, the train numbers are increased every time.
The linked lists in columns 6 and 7, containing the scheduled order of events at a timetable point, are generated automatically by the 'Generate_TEG' algorithm when building the timed event graph.

Table 7.3 *Example of manual input for one train line.*

TN (Train number)	***LN*** (Line number)	***TTP*** (Timetable point)	***IO*** (IO-point)	***type***
501	5	1	1	4 ('start')
501	5	2	1	1 ('arrival')
501	5	2	3	2 ('departure')
501	5	3	3	1 ('arrival')
501	5	3	5	2 ('departure')
501	5	4	1	1 ('arrival')
501	5	4	3	2 ('departure')
501	5	5	1	1 ('arrival')
501	5	5	3	2 ('departure')
501	5	6	1	3 ('end')

The timetable vector *d*
The same approach is followed when creating the timetable vector *d*: the scheduled event times for one train run are created manually, after which a small script multiplies the input 6 times, thereby increasing the scheduled event times with 30 minutes each time, such that each train line runs in a 30 minutes service.

Amount of travellers
As explained in section 6.5.3, an amount of travellers is attached to each event in the model. Tables containing the used amounts of travellers can be found in appendix 10.2.

Turn data
As in a real railway system, train turns are included in this model as well. The turns can be found in Table 7.4. All turns have a minimum turning time of 6 minutes, but the actual layover times are longer due to the schedule, yielding extra slack time for absorbing delays.

Table 7.4 *Train turns in rolling stock circulation of the test case. The minimum turning time is 6 minutes.*

Feeder train	Connecting train	Scheduled lay-over time (min.)
101	203	10
102	204	10
103	205	10
104	206	10
201	102	8
202	103	8
203	104	8
204	105	8
205	106	8
501	604	26
502	605	26
503	606	26

Feeder train	Connecting train	Scheduled lay-over time (min.)
301	403	17
302	404	17
303	405	17
304	406	17
401	303	37
402	304	37
403	305	37
404	306	37
601	503	20
602	504	20
603	505	20
604	506	20

Hindrance conflict data
As can be seen in the testing network shown in Figure 7.1, conflicting train routings are possible in station 3. In particular, trains departing in the direction of station 7 have to cross the incoming track of the railway line from station 1. When one of the conflicting trains has passed the crossing, the track is entirely free for the other train, which means that the conflict duration is relatively short. A duration of 2 minutes is used for these crossing conflicts. The conflicts to be included in the model are shown in Table 7.5.

Table 7.5 *Conflicting train movements in testing network.*

Train movement	Conflicting train movement	Conflict duration
Departure of line 2 to station 7	Arrival of line 3 from station 1	2 minutes
Departure of line 2 to station 7	Arrival of line 5 from station 1	2 minutes

Recall the implementation of conflict matrices and their notation explained in section 5.2.7. For each combination of line number *LN*, timetable point *TTP* and IO-point *IO* a conflict matrix exists in which each row represents a train movement conflicting with such a combination, containing the following elements:

$Conflicts\{LN, TTP, IO\}(r) = (LN^{conflicting}{}_r, IO^{conflicting}{}_r, t_r)$

where for each row *r*:
$LN^{conflicting}$ = line number of the conflicting line,
$IO^{conflicting}$ = IO-point used by the conflicting line,
t = minimal amount of time the conflicting train of $LN^{conflicting}$ has to wait for *LN*.

For the test network, this yields the following conflict matrices:

$$Conflicts\{2,\ 3,\ 2\} = \begin{bmatrix} 3,\ 3,\ 2 \\ 5,\ 3,\ 2 \end{bmatrix}$$

Denoting that line 2 at timetable point 3 using IO-point 2 is conflicting with:

- line 3 using IO-point 3 with a conflict duration of 2 minutes
- line 5 using IO-point 3 with a conflict duration of 2 minutes

$$Conflicts\{3,\ 3,\ 3\} = [2,\ 2,\ 2]$$

Denoting that line 3 at timetable point 3 using IO-point 3 is conflicting with line 2 using IO-point 2 with a conflict duration of 2 minutes.

$$Conflicts\{5,\ 3,\ 3\} = [2,\ 2,\ 2]$$

Denoting that line 5 at timetable point 3 using IO-point 3 is conflicting with line 2 using IO-point 2 with a conflict duration of 2 minutes.

The other matrices *Conflicts*{*LN*, *TTP*, *IO*} are empty, denoting that no other hindrance conflicts occur in the network. Note that the *headway* conflicts are implemented later by the 'Generate TEG' algorithm.

Generating the timed event graph
With the input data described above, a timed event graph is generated using the 'Generate_TEG' algorithm described in section 5.3.3. This yields the following data:

- The *Event* matrix, now completed with the linked lists.
- The arclist *A*.

After this, the adjacency lists *Adjhead* and *Adjtail* are created using the algorithm 'Arclist2adj', described in section 5.2.4. Now, the timed event graph is ready to use for the case study.

List of possible dispatching actions
As explained in section 6.5.2, a list of possible dispatching actions has been created using the algorithm 'Generate_Inventory'. The output, the *Choice* matrix, was used as input for the optimization algorithm 'Dispatch' described in section 6.5.4, yielding the results presented in the next section.
Note that it is assumed that trains can overtake each other (i.e. the sequence order can be changed) at each station in the testing network. Note also that short turning is considered possible at each station in the testing network, although the switches and tracks needed for this are not shown in the track layout of Figure 7.1.

7.4 Results

Two delay scenarios have been tested. The following sections contain descriptions of the delay scenarios and the respective results. For each scenario, the following parameters are calculated:

- advised dispatching actions when no short turning is allowed, calculated by the 'Dispatch' algorithm,
- advised dispatching when short turning is allowed,
- the delay propagation when no dispatching actions are applied,
- the delay propagation when dispatching actions are applied,
- the first order delay (i.e. the delay resulting from initial delays),
- the consecutive delay (i.e. the delay caused by delay propagation).

In this project, first order delay is defined as follows:

First order delay is the delay resulting from a given set of initial delays. In the timed event graph, the first order delay is transmitted via running time and dwell time.

The consecutive delay is defined as follows:

Consecutive delay is the delay resulting from delay propagation caused by headway constraints, hindrance constraints, transfers and turns.

The results of each scenario are interpreted and discussed using time-distance diagrams.

7.4.1 Delay scenario 1: Departure of train 102 delayed

Reason for this delay scenario
The rolling stock circulation of lines 1 and 2 has relatively short layover times (10 minutes at station 6 and 8 minutes at station 7), which makes these lines interesting to compare the dispatching actions 'change order of trains' and 'short turning' with the delay propagation when no dispatching actions are applied.
Train 102 is the second train trip of line 1 departing from station 7, which is chosen for a delay scenario since a delay of train 101 (the first train) would cause less delay propagation, which is not insightful for a test case.

Expected results
The following results are expected:

- In case of small delays, it should be advised not to carry out any dispatching actions, since slack times and layover times can make up for the delay.
- In case of bigger delays, order changes are expected. In particular, it is expected that the local train, which is scheduled to be overtaken by the delayed train in station 4, is advised not to wait for this train anymore.
- In case of severe delays, the advice to turn before the scheduled turn station is expected

Calculated results

Five different delays ranging from 5 minutes to 25 minutes are tested in this scenario. For each delay, the advised dispatching actions minimizing the total passenger delay calculated using the 'Dispatch' algorithm are described in Table 7.6. The delay propagation in the situations with and without dispatching actions are summarized in Table 7.7.

In case of initial delays of 5 minutes and 10 minutes, it is advised to apply no dispatching actions at all. The delays will settle automatically by consuming the slack times in the timetable and at the turning stations. The first dispatching action is applied when the delay amounts 15 minutes. When a dispatching action is applied, the practical implications for the train service are described between brackets.

Table 7.6 *Advised dispatching actions (no short turning included) and total delays in scenario 1.*

Delay (min.)	**Advised dispatching actions (without short turning)**
+5	Do nothing
+10	Do nothing
+15	1. Switch order train 102 and train 502 between stations 4 and 6. → ('Cancel scheduled overtaking of train 502 by train 102')
+20	1. Switch order train 102 and train 502 between stations 4 and 6. → ('Cancel scheduled overtaking of train 502 by train 102') 2. Switch order train 102 and train 303 between stations 3 and 5. → ('Let train 102 run behind train 303 between stations 3 and 5') 3. Switch order train 204 and train 604 between stations 4 and 3. → ('Cancel scheduled overtaking of train 604 by train 204')
+25	1. Switch order train 102 and train 502 between stations 4 and 6. → ('Cancel scheduled overtaking of train 502 by train 102') 2. Switch order train 102 and train 303 between stations 3 and 6. → ('Let train 102 run behind train 303 until arrival at station 6') 3. Switch order train 204 and train 604 between stations 4 and 3. → ('Cancel scheduled overtaking of train 604 by train 204') 4. Switch order train 105 and train 505 between stations 4 and 5. → ('Move scheduled overtaking of train 505 from station 4 to station 5')

Table 7.7 *Summary of delay propagation with and without dispatching actions.*

	Delay propagation without dispatching actions			**Delay propagation with dispatching actions**		
Initial delay	Total first order delay	Total consecutive delay	Total passenger delay	Total first order delay	Total consecutive delay	Total passenger delay
5	24	7	2600	-	-	-
10	64	31	8700	-	-	-
15	104	128	22010	104	62	15960
20	144	326	42970	144	132	29110
25	184	632	72960	184	229	41190

Short turning is a dispatching action with large consequences for the travellers, since they have to wait for the next train. Therefore, the possibility to use this dispatching action is calculated separately. When short turning is included in the calculation, the advice stays the same for the

delays of 5, 10, 15 and 20 minutes. Only the delay of 25 minutes yields better results when short turning is applied, as shown in Table 7.8.

Table 7.8 *Alternative dispatching action with short turning in scenario 1.*

Delay (min.)	**Advised dispatching actions (with possibility for short turning)**
+25	1. Short-turn train 102 at station 3. → ('Hence, train 204 is cancelled between stations 6 and 3')

Table 7.9 *Summary of delay propagation with and without dispatching actions, when short turns are allowed.*

	Delay propagation without dispatching actions			**Delay propagation with dispatching actions**		
Initial delay	Total first order delay	Total consecutive delay	Total passenger delay	Total first order delay	Total consecutive delay	Total passenger delay
25	376	440	72960	49	0	37700

Discussion of the results

The delay propagation when no dispatching actions are applied in case of a delay of 25 minutes for train 102 is shown in Figure 7.5. Each train is forced to run in the scheduled order due to the lack of dispatching actions, which can be clearly seen from the pattern of train 502. This train is scheduled to be overtaken by train 102 at station 4, and therefore has to wait almost half an hour for the overtaking to take place.

In contrast, the situation when the advised dispatching actions are applied is depicted in Figure 7.6. In this diagram can be seen that train 502 and train 303 are allowed to run *before* the delayed train 102, and that train 102 gets an almost conflict-free path along the track.

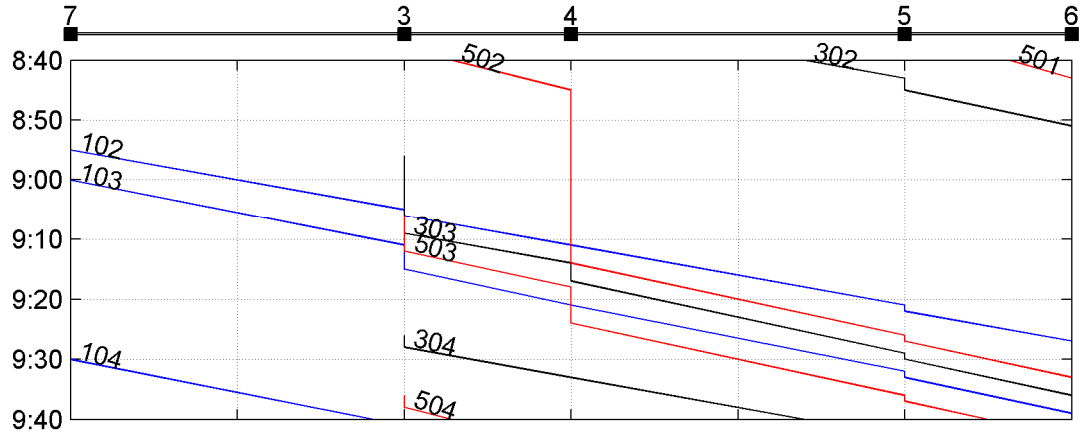


Figure 7.5 *Time-distance diagram of delay propagation without dispatching actions when train 102 has 25 minutes delay.*

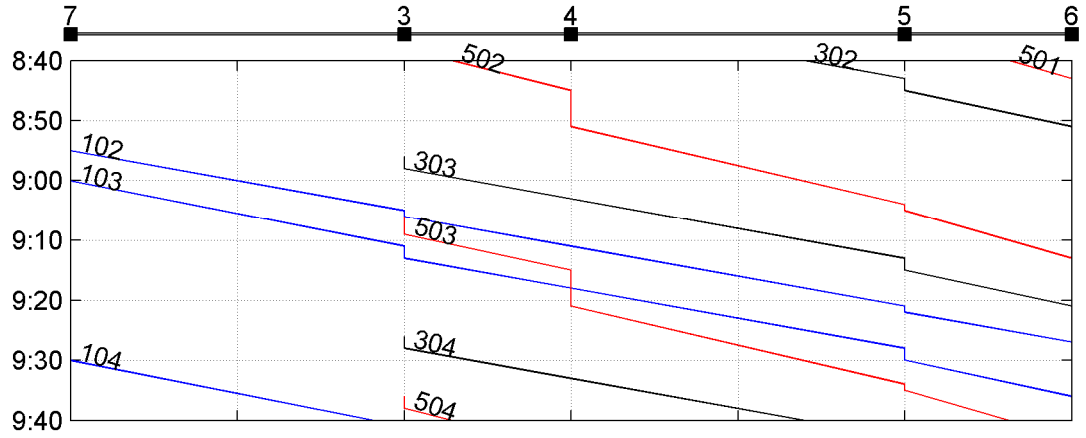


Figure 7.6 *Time-distance diagram of delay propagation with dispatching actions (no short turns allowed) when train 102 has a delay of 25 minutes.*

Some other interesting observations can be made. For instance in case of a delay of 20 minutes, the order of trains 102 and 303 is switched between stations 3 and 5, and not until the terminal station of both trains, station 6, as shown in Figure 7.7. This can be explained when regarding the scheduled layover times at station 6. Train 102 has a relatively short layover time (10 minutes), while train 303 has a longer layover time (17 minutes). Hence, when train 102 can arrive at the terminal station before train 303, less delay is propagated via the turn of train 102. This is a result of the fact that the algorithms evaluate the effectiveness of dispatching actions on network level.

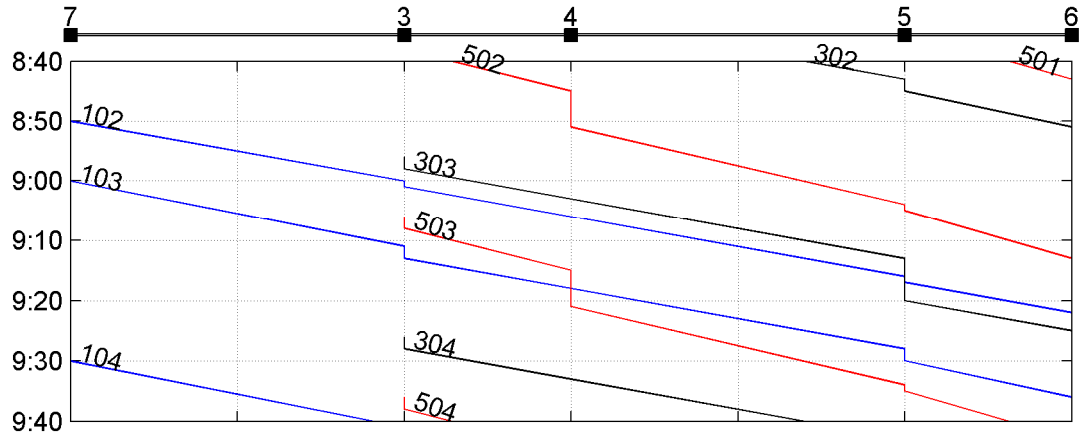


Figure 7.7 *Time-distance diagram of train service with dispatching actions when train 102 has a delay of 20 minutes.*

Influence on the capacity consumption

Note that due to the delayed train 102, between approximately 9:00 and 10:00 more trains run along the track from station 4 to station 6 than in the situation without delays. This leads to a higher capacity consumption, which has been calculated with the algorithm 'ExploitationRate' presented in section 5.5, leading to the following results:

Capacity consumption on the railway track between stations 4 and 6, between 9:00 and 10:00 hours:	
Original schedule:	37 %
In the disrupted situation:	45 %

Note that 45 % is still sufficient for a stable service. Obviously, the delayed train 102 is missing in the hour between 8:00 and 9:00, which leads to a lower capacity consumption during that period.

7.4.2 Delay scenario 2: Departure of train 502 delayed

Reasons for this delay scenario
Train 502 is a local train of line 5 starting at station 1 and ending at station 6. The interlocking route of this train line crosses the routing of line 2 at station 3. Therefore, this delay scenario offers the possibility to study the 'postpone' algorithm, as well as the 'change order' algorithm.

Expected results

- In case of small delays, no advised dispatching actions are expected.
- In case of bigger delays, postponing actions are expected, since conflicting interlocking routes with the delayed train at station 3 exist. Furthermore, order changes are expected.

Calculated results
As in the previous delay scenario, 5 different delays are tested. The advised dispatching actions calculated by the optimization algorithm are shown in Table 7.10. A summary of the delay propagations with and without dispatching actions for this scenario is shown in Table 7.11.

Table 7.10 *Total delays with dispatching actions in scenario 2.*

Delay (min.)	**Advised dispatching actions (without short turning)**
+5	Do nothing
+10	1. Switch order train 502 and train 102 between stations 3 and 4. → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4') 2. Postpone the arrival of train 502 at station 3 until train 202 has departed.
+15	1. Switch order train 502 and train 102 between stations 3 and 4. → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4') 2. Postpone the arrival of train 502 at station 3 until train 202 has departed.
+20	1. Switch order train 502 and train 102 between stations 3 and 4. → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4') 2. Postpone the arrival of train 502 at station 3 until train 202 has departed. 3. Switch order train 502 and train 303 between stations 4 and 6. → ('Let train 303 overtake train 502 at station 4.')
+25	1. Switch order train 502 and train 102 between stations 3 and 4. → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4') 2. Postpone the arrival of train 502 at station 3 until train 202 has departed. 3. Switch order train 502 and train 303 between stations 4 and 6. → ('Let train 303 overtake train 502 at station 4.')

Table 7.11 *Summary of delay propagation with and without dispatching actions.*

	Delay propagation without dispatching actions			**Delay propagation with dispatching actions**		
Initial delay	Total first order delay	Total consecutive delay	Total passenger delay	Total first order delay	Total consecutive delay	Total passenger delay
5	19	4	1230	-	-	-
10	50	62	8330	50	5	2870
15	98	198	24540	98	0	5080
20	148	460	52290	148	25	8930
25	198	910	96430	198	74	16930

When the possibility to schedule short turns (i.e. before the scheduled turning station) is included, the advice stays the same. This can be explained by the fact that train 502 has a relatively long layover time (26 minutes) so that a delay of 25 minutes can be absorbed almost entirely at the turning station (the minimum layover time is 6 minutes).

Discussion of the results

A difference with scenario 1 is that dispatching actions are advised already for the relatively small delay of 10 minutes in scenario 2, while this delay did not require dispatching actions in scenario 1. Figure 7.8 shows a time-distance diagram of the delay propagation for this situation (train 502 delayed by 10 minutes). It can be seen that train 102 is hindered by train 502 on the track between stations 3 and 4. Figure 7.9 shows the situation with dispatching actions (i.e. order change of trains 502 and 102 between stations 3 and 4). This is very effective, since train 102, an intercity train, is not hindered anymore, and 5 minutes of the delay of train 502 are absorbed at station 4 since the time-consuming overtaking is cancelled. The graph shows clearly that the dwell time of train 502 is reduced to the minimum of 1 minute, instead of the scheduled 6 minutes including the overtaking action.

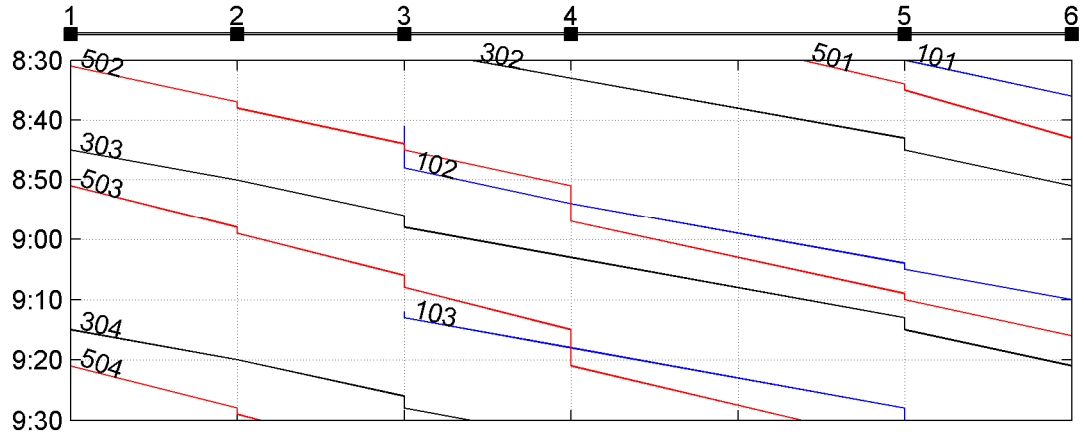


Figure 7.8 *Time-distance diagram of delay propagation without dispatching actions when train 502 has a delay of 10 minutes.*

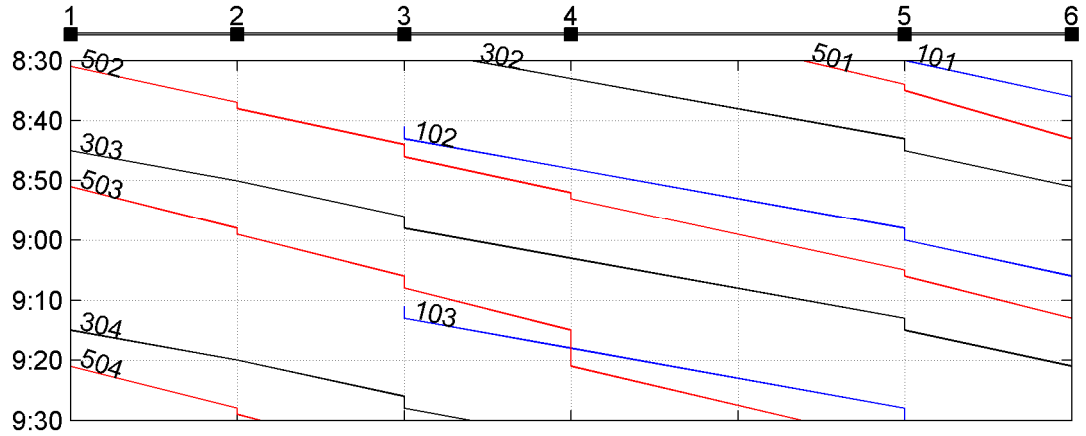


Figure 7.9 *Time-distance diagram of delay propagation with dispatching actions when train 502 has a delay of 10 minutes.*

The results for the initial delay of 20 minutes are shown in Figure 7.10 and Figure 7.11, for the situations without and with dispatching actions respectively. As with scenario 1, a fixed sequence order of the trains causes heavy delay propagation (in this case from the initially delayed train 502 to trains 102 and 303).
Other causes of delay propagation are the hindrance constraints at station 3. The delay of train 103 is caused by the hindrance conflict between trains 502 and 202 at station 3. The delay of train 202 caused by this conflict is propagated to train 103 via the turn at station 1. In Figure 7.11 can be seen that train 103 runs on time when the correct postponing actions are applied.
Finally note that the scheduled overtaking of train 502 by train 102 is in the rescheduled situation replaced by an overtaking of train 502 by train 303.

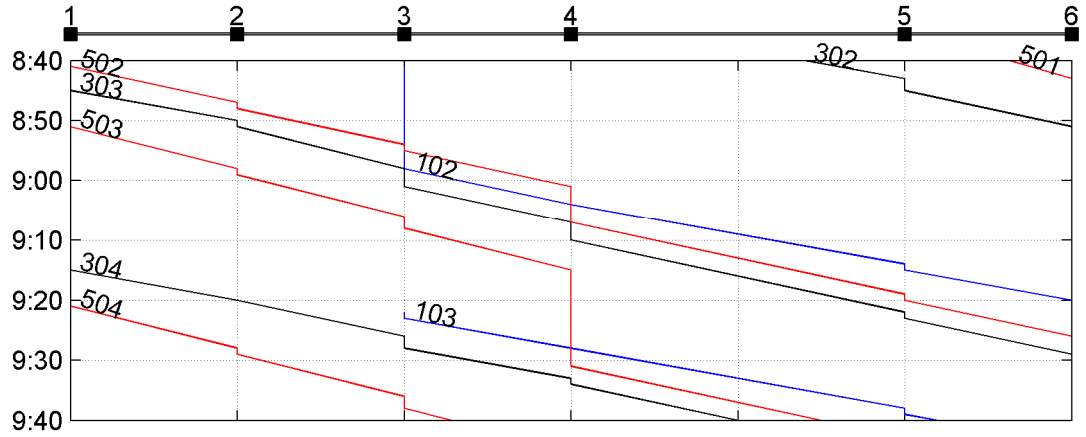


Figure 7.10 *Time-distance diagram of delay propagation without dispatching actions when train 502 has a delay of 20 minutes.*

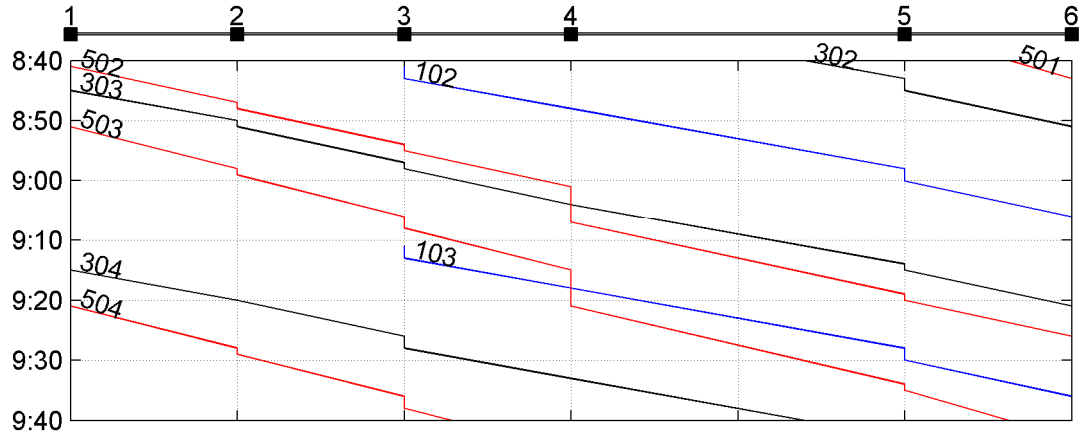


Figure 7.11 *Time-distance diagram of delay propagation with dispatching actions when train 502 has a delay of 20 minutes.*

7.4.3 Influence of the number of passengers

As explained in the description of the input data in section 7.3.2, amounts of passengers are attached to each event in the model of the test case. In this section, the influence of the number of passengers on the advised dispatching actions is briefly displayed using delay scenario 1.
In order to show the influence of the number of passengers on the results, it is investigated whether the advised dispatching actions change if the amount of passengers in trains 102 and 204 is lowered.
The amounts of passengers used in the test case can be found in appendix 10.2, but for completeness the amounts of passengers for trains 102 and 204 are shown here as well, see Table 7.12. In the table can be seen that a relatively large amount of passengers use train 102 between stations 5 and 6 (200 passengers embarking in station 5, 300 passengers disembarking at station 6, which is the terminal station). Train 204, running in the opposite direction, is used by 200 passengers, embarking at station 6.

Table 7.12 *Amounts of passengers for trains 102 and 204 in test case.*

Train 102		**Train 204**	
Station	**Amount of embarking / alighting travellers**	**Station**	**Amount of embarking / alighting travellers**
7, departure	200	6, departure	200
3, arrival	50	5, arrival	50
3, departure	50	5, departure	50
4 (through)	0	4 (through)	0
5, arrival	100	3, arrival	100
5, departure	200	3, departure	100
6, arrival	300	7, arrival	200

It is expected that short turning becomes more favourable when train 102 is delayed if less passengers use trains 102 and 204 between stations 5 and 6. To investigate this, the amounts of passengers are changed into the values shown in Table 7.13, where only 100 passengers disembark from train 102 at its terminal station, and only 50 passengers use train 204 from its start at station 6.

Table 7.13 *Modified amounts of passengers for trains 102 and 204.*

Train 102		**Train 204**	
Station	**Amount of embarking / alighting travellers**	**Station**	**Amount of embarking / alighting travellers**
7, departure	200	6, departure	50
3, arrival	100	5, arrival	20
3, departure	50	5, departure	20
4 (through)	0	4 (through)	0
5, arrival	100	3, arrival	20
5, departure	50	3, departure	120
6, arrival	100	7, arrival	150

A test run using the algorithm 'Dispatch' shows that the advices have changed indeed, as can be seen in Table 7.14, where the results with the lowered passenger amounts for trains 102 and 204 are shown. Note the two differences with the original test case of scenario 1 described in section 7.4.1:

1. When train 102 is delayed 10 minutes, an order change is advised, while in the original test case 'do nothing' was advised.
2. When train 102 is delayed 20 minutes, a short turn is advised, while in the original test case order changes were advised.

Difference 1 can be explained by the time-distance diagrams of Figure 7.12 and Figure 7.13. The former shows the delay propagation when no dispatching actions are carried out (which is the advice in the original test case for a delay of 10 minutes of train 102). It can be clearly distinguished that train 502 has to wait more than 10 minutes at station 4 for train 102 to come through.

When the amount of passengers in train 102 is lowered, train 502 gets more relative importance in the calculation of the most effective dispatching actions, which leads to the new advice of moving the overtaking to station 5, causing less delay for train 502. This is visualized in Figure 7.13.

Table 7.14 *Advised dispatching actions with lowered passenger amounts for trains 102 and 204 (short turning is allowed).*

Delay (min.)	**Advised dispatching actions (with short turning)**
+5	Do nothing
+10	Switch order train 102 and train 502 between stations 4 and 5. → ('Move scheduled overtaking of train 502 by train 102 from station 4 to station 5')
+15	Switch order train 102 and train 502 between stations 4 and 6. → ('Cancel scheduled overtaking of train 502 by train 102')
+20	Short-turn train 102 at station 3.
+25	Short-turn train 102 at station 3.

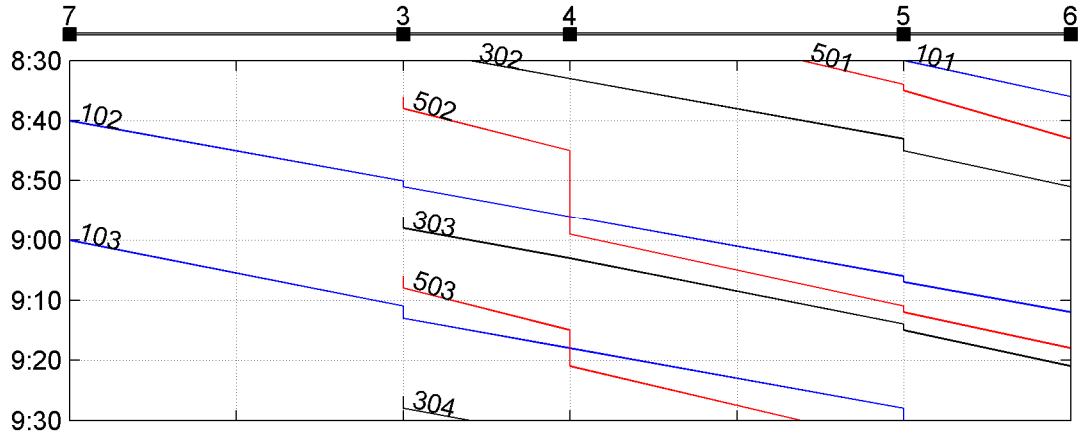


Figure 7.12 *Time-distance diagram of delay propagation when train 102 as a 10 minute delay and no dispatching actions are carried out.*

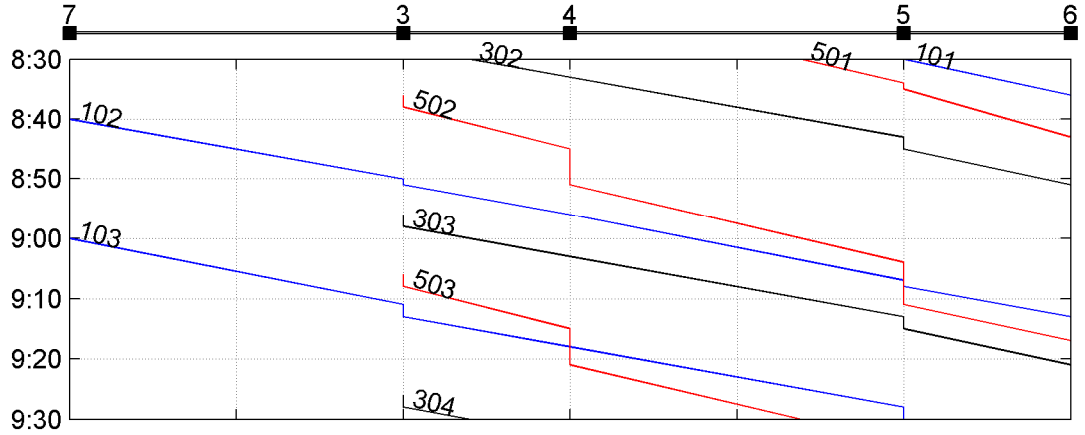


Figure 7.13 *Time-distance diagram of advised dispatching actions when train 102 has a 10 minute delay and a lowered amount of passengers.*

The comparison between the delays of the individual trains in Table 7.15 shows the difference. With the lowered amount of passengers in trains 102 and 204, slightly more delay for these trains is accepted, while the delay of train 502 is lowered considerably.

Original amounts of passengers	
Train number	**Delay (min.)**
102	64
204	3
502	27
303	1

Less passengers in trains 102 and 204	
Train number	**Delay (min.)**
102	67
204	9
502	10
604	1

Table 7.15 *Comparison of individual train delays with different amounts of passengers.*

The second difference can be explained directly from the fact that less passengers are affected by a short turn, leading to the outcome that this dispatching action becomes more favourable when the turning trains carry less passengers.

7.5 Conclusion

This chapter contains a case study in which the algorithms for evaluating dispatching actions, presented in chapter 6, are tested on a fictive testing network to investigate the consistency and plausibility of the results. As an objective function, the total passenger delay has been used.

The expectation was that in case of small delays, no dispatching actions can reduce the passenger delay. When the delay gets higher, order changes are expected to be effective to reduce delay propagation. Short turns are expected to be effective only in case of big delays (i.e. more than 20 minutes), since the objective function takes into account the fact that passengers have to leave the train if a short turn is applied. The results of two delay scenarios met these expectations, showing that:

1. The developed algorithms implement the dispatching actions in a consistent way.
2. The algorithms can be used effectively to evaluate dispatching actions.

One delay scenario has been used to test the effect of the chosen objective function on the dispatching actions generated by the optimization algorithm. This showed that the amount of passengers influences the results in such a way that the generated dispatching actions show an emphasis on reducing the trains carrying passengers.

8 Conclusions

This chapter contains the conclusions of the research subject presented in this thesis. The main goal was to implement railway dispatching actions in a max-plus model, and to create an algorithm which can be used to evaluate dispatching actions.
Section 8.1 presents the main conclusions of this thesis. Section 8.2 offers an outlook on the applicability of the developed theory in practice, where a distinction will be made between offline and online applications. Some recommendations for future research will be discussed in section 8.3.

8.1 Main conclusions

A dispatching action is an intervention in the rail traffic system with the purpose of solving a conflict between train runs, thereby aimed at reducing delays and their propagation. Dispatching actions in which the line routes and sequence orders of trains remain unchanged are called traffic control actions. The other dispatching actions, involving changes in the line routes and/or sequence orders of trains, are called rescheduling actions.

Over the last decades, research has been carried out to develop tools able to calculate the most optimal approach for solving conflicts in railway networks. Yet, a tool for quick evaluation of dispatching actions on a network scale level acting as a real time decision support system for dispatchers does still not exist.

In this thesis, algorithms that can be used effectively to evaluate dispatching actions on network level using max-plus theory have been presented. It has been shown that the timed event graph representation of a max-plus system is suitable for the development of such algorithms. Particularly the advantage of being able to visualize the implications of dispatching actions on the model has been proven useful for the development of the algorithms in this thesis. The detail level of the model is aimed at a network wide evaluation, which means keeping the model and the algorithms simple and time efficient. Using this model, the following dispatching actions have been implemented as algorithms:

- Change the sequence order of trains.
- Short-turn a train (i.e. at a station before its terminal station).
- Postpone the departure or arrival of trains at stations with conflicting interlocking routes.

For the development of these algorithms, so-called ‘construction rules’ have been specified for each dispatching action. These construction rules describe the changes in the timed event graph necessary to represent the corresponding dispatching action, and follow immediately from the implications of the dispatching action on the model. In practice, more different dispatching actions can be evaluated with the theory developed in this thesis. Changing the sequence order of trains can for instance be used to move a scheduled overtaking to another station, which is technically a different dispatching action. Furthermore, the described construction rules can be used to develop other algorithms, for instance for cancelling train runs, introducing new train runs, etc.

In order to use the algorithms for finding optimal dispatching actions for a given network and a set of initial delays, they can be implemented in an optimization algorithm. In this project, a

greedy approach has been used. To maximize their effectiveness, order changes of trains have to be evaluated in combination with the appropriate postponements of arrivals and departures of the involved trains when conflicting interlocking routes are present in the network. The total passenger delay in the network served as the objective function, thereby including the negative effects for passengers when train trips are partly cancelled due to short turning of delayed trains. Although global optimality has not been checked, a test case with two delay scenarios showed that the generated dispatching actions are plausible and consistent.

8.2 Applicability in practice

The developed algorithms can change an existing max-plus model, represented by a timed event graph, in order to evaluate dispatching actions. This can be used for offline and online applications, as will be briefly described in this section.

Offline applications
Offline applications for the evaluation of dispatching actions are particularly useful for analysis of railway operations and in the timetable design process. They enable a timetable designer for example to:

- Assess the flexibility of a timetable with regard to the application of dispatching actions,
- Assess the stability of possible dispatching actions,
- Design emergency schedules in case of disruptions,
- Use the results of offline evaluations to decide whether certain timetable paths can be inserted in an existing timetable or not.

For such applications, the developed algorithms could for instance be used in PETER, an offline timetable evaluation tool described in [7], to calculate the delay propagation in situations where dispatching actions are applied. PETER is originally intended to evaluate periodic timetables and therefore makes use of a periodic max-plus model. However, an acyclic model is more suitable for the operations needed for the evaluation of dispatching actions. Therefore, PETER is to be expanded with the possibility to 'unfold' the cyclic timed event graph as described in section 4.6.2 before algorithms for modelling dispatching actions can be implemented.

Online applications
The ultimate goal of the research subject dealt with in this thesis is to enable real time evaluation of dispatching actions to support the dispatcher. The algorithms developed in this thesis can be used for this if the following requirements are fulfilled:

- A max-plus model of the existing railway system has to be available, which implies that detailed knowledge of the timetable, minimum running and dwell times, and conflicting routes of train pairs with headway times has to be available.
- The online application has to be provided with the actual delays in the network. Data streams that can be used for this are already available in the TNV system at the Dutch dispatchers' offices.
- The online application has to be able to cope with all possibly occurring dispatching actions and disruptions in the network, since the used max-plus model has to be kept consistent with the real situation.

When the aforementioned requirements are fulfilled, the following online applications are possible:

- Implementation as a tool for the dispatcher to evaluate and compare the effectiveness of different dispatching actions when he has to choose between different approaches to react

on a disruption (e.g. one approach including short turns and another approach where short turns are not allowed).
- Implementation as an online tool calculating the most effective approach for the dispatcher to react on a disruption.
- Visualization of the train traffic including the dispatching actions in time-distance diagrams to show dispatchers that the proposed dispatching actions are effective. A visual representation makes it easier for employees to take a decision since they can immediately recognize the impacts of the proposed dispatching actions, giving them the knowledge and confidence needed to decide quickly in case of disruptions.
- The max-plus model which has been changed and updated according to the dispatching actions during the day can be stored for later analysis, as it contains a model of the train service as actually carried out.

8.3 Recommendations for future research

Carrying out this research project has brought about many ideas for future research. The most important suggestions are discussed in this section.

Scale level of the used model

In the current model, stations are modelled as a black box (a timetable point), so what happens inside is unknown. When dispatching actions regarding the interlocking routes of trains have to be implemented (such as: changing the platform track of a train), the stations have to be modelled in more detail. However, this can have implications on the running time of the algorithms, endangering their ability to run in real time.

A possibility to avoid this problem could be to use a less detailed model to calculate dispatching actions on network level, and to use a more detailed model to calculate dispatching actions on station level afterwards when necessary. A more detailed model can for instance be obtained by:

- Modelling all different platform tracks of a station separately, instead of modelling one timetable point as a black box,
- Modelling each switch and crossing separately, so that interlocking routes and the conflicts between them can be modelled in great detail,
- Modelling all block signals separately.

The purpose of the model thereby determines which scale level should be used. As shown in this thesis, control and analysis on network level does not require the aforementioned level of detail. However, it is recommended for all future research on this subject to carefully consider the level of detail of the model in relation with its purpose.

Modelling dispatching actions at network level

The dispatching actions implemented in this thesis mainly regard switching the sequence order of trains, or letting a train turn before its scheduled turning stations. However, as described in chapter 3, more dispatching actions are possible.

Dispatching actions with implications at a higher level, (i.e. at network level) can be effective in case of bigger disruptions such as a blocked track, etc. It is recommended to investigate the implications of such dispatching actions (e.g. re-routing a train on network level) on the max-plus model, since the model has to be capable of representing these bigger dispatching actions when used in practice. The construction rules and algorithms in this thesis may form a start for this.

Optimization strategy

In this thesis, a simple greedy approach has been used to find the most effective dispatching actions for a given railway network, timetable and delay scenario. It is recommended to conduct research to more powerful optimization strategies, while keeping in mind the online application and the short calculation times required for this. The following approaches may be examined:

- Use the delay propagation algorithm in an optimization approach to return information about occurring conflicts in order to choose dispatching actions that are likely to be effective (e.g. when a delay is transmitted by a headway arc, changing the order between the involved trains is likely to be effective). Note that this approach has already been illustrated in this thesis by trying dispatching actions related to delayed trains only, but could be improved further.
- Aim at the near-optimum instead of the overall optimum. When looking for effective dispatching actions, near-optimality may be sufficient when this yields a substantial improvement of the current situation.
- Use characteristics of the max-plus model, such as the cycle time, to get an indication of the stability and effectiveness of selected dispatching actions, which can be used by the optimization algorithm to work effectively and intelligently.
- Use the delay propagation algorithm to detect conflicts and resolve them within the delay propagation algorithm using a branch and bound approach (branching on different dispatching actions).

Case study on a (part of a) real railway network

It is recommended to carry out a case study on a (part of a) real railway network, such as for example the Dutch railway line Den Haag – Dordrecht. It is expected that this yields valuable insight in the requirements needed for the implementation in practice, such as:

- Which data is required, and how can this data be obtained?
- How should the model be adjusted or extended in order to work on a real railway network?

For this analysis, the outcomes of the model could be compared to real-world data from situations in which dispatching actions were carried out in reality.

Relate results to the duration of a disruption

In this research project, the dispatching actions were mainly aimed at reducing the effects of a given set of initial delays in the network. However, in case of bigger disruptions, lasting for example 8 hours, different strategies to reduce the effects on the train service can be adopted. Offline investigation using real-world data and/or the algorithms presented in this thesis may reveal a correlation between the duration of a disruption and the effectiveness of dispatching actions. Such results can be used to adopt effective strategies that differ in case of shorter or longer disruptions, which could be beneficial for the optimization algorithm. During the different stages of a disruption, the optimization strategy could for instance be aimed on:

1. Reducing the effects on the train service during the disruption and,
2. Get the train service back to the scheduled situation when the disruption is over.

Note however that this is only useful when the theory and algorithms in this thesis are extended with more dispatching actions to enable investigating the full range of possibilities from which a real dispatcher can choose.

9 Bibliography

[1] Adenso-Díaz, B., Oliva González, M., González-Torre, P., "On-line timetable re-scheduling in regional train services", *Transportation Research Part B*, Vol. 33, pp. 387 – 398, 1999.

[2] Ahuja, R.K., Magnanti, T.L., Orlin, J.B., *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, New Jersey, 1993.

[3] Boom, T.J.J. van den, De Schutter, B., "Modelling and control of discrete event systems using switching max-plus-linear systems," *Control Engineering Practice*, Vol. 14, Issue 10, pp. 1199 – 1211, 2006.

[4] Cormen, T., Leiserson, C., Rivest, R., *Introduction to Algorithms*, The MIT Press, London, England, 1990.

[5] D'Ariano, A., Pranzo, M., Hansen, I., "Conflict Resolution and Train Speed Coordination for Solving Real-Time Timetable Perturbations", *IEEE transactions on intelligent transportation systems*, Vol. 8, No. 2, 2007.

[6] Goverde, R.M.P., "Synchronization Control of Scheduled Train Services to Minimize Passenger Waiting Times", In: *Proceedings of the 4th TRAIL Year Congress*, part 2, TRAIL Research School, Delft, 1998.

[7] Goverde, R.M.P., *Punctuality of Railway Operations and Timetable Stability Analysis*, TRAIL Thesis series no T2005/10, Delft, 2005.

[8] Heller, S., Schaer, T., "DisKon – Disposition und Konfliktlösungsmanagement der DB AG", *Eisenbahningenieur* Vol. 55, Issue 9, pp. 102 – 122, 2004.

[9] Jacobs, J. "Reducing delays by means of computer-aided 'on-the-spot' rescheduling", In: J. Allan, C. A. Brebbia, R. J. Hill, G. Sciutto & S. Sone (Eds.), *Computers in Railways IX* (pp. 603–612). Southampton, UK: WIT Press, 2004.

[10] Koopman, D.J., *Vervoersprestatieverbetering van spoorwegen tijdens calamiteiten*, Delft University of Technology, Faculty of Civil Engineering and Geosciences, 2007.

[11] Mazzarello, M., Ottaviani, E., "A traffic management system for real-time traffic optimisation in railways", *Transportation Research Part B*, Vol. 41, pp. 246–274, 2007.

[12] Pachl, J., *Railway Operation and Control*, VTD Rail Publishing, Mountlake Terrace, WA, USA, 2004.

[13] Törnquist, J. "Railway traffic disturbance management—An experimental analysis of disturbance complexity, management objectives and limitations in planning horizon", *Transportation Research Part A: Policy and Practice*, Volume 41, Issue 3, pp. 249-266, 2007.

[14] UIC 406R, *Capacity*, 2004.

10 Appendix

10.1 Algorithm 'ChangeList'

The algorithm 'ChangeList' changes the linked lists containing the scheduled order of events when x and E, scheduled directly subsequent to each other, are swapped. This is the case when event x is postponed after E. Recall from section 5.2.1 that for each event x the preceding event P_x and the next event N_x scheduled at the same timetable point are stored (thus forming a linked list). This is illustrated in the table below.

Algorithm 10.1 (CHANGELIST)

Input:
Event = list of events
x, E = subsequently scheduled events that will be swapped
Output:
Event = list of events with updated linked lists

1. $P_E \leftarrow P_x$;
2. $\boldsymbol{N_E} \leftarrow x$;
3. $P_x \leftarrow E$;
4. $N_x \leftarrow N_E$;
5. **if** $P_x \neq 0$ **then**
6. $N(P_x) \leftarrow E$;
7. **if** $N_E \neq 0$ **then**
8. $P(N_E) \leftarrow x$;

Linked lists in original order:

Events:	P_x		x		E		N_E	
Linked list:	$P(P_x)$	$N(P_x)$	$P(E)$	$N(E)$	$P(x)$	$N(x)$	$P(N_E)$	$N(N_E)$
Value:	-	x	P_x	E	x	N_E	E	-

The linked list as they are updated when events x and E are swapped is shown in the next table:

Events:	P_x		E		x		N_E	
Linked list:	$P(P_x)$	$N(P_x)$	$P(E)$	$N(E)$	$P(x)$	$N(x)$	$P(N_E)$	$N(N_E)$
Value:	-	E	P_x	x	E	N_E	x	-

10.2 Amounts of travellers used in test case

Line 1

Station	**Amount of embarking / alighting travellers**
7, departure	200
3, arrival	50
3, departure	50
4 (through)	0
5, arrival	100
5, departure	200
6, arrival	300

Line 2

Station	**Amount of embarking / alighting travellers**
6, departure	200
5, arrival	50
5, departure	50
4 (through)	0
3, arrival	100
3, departure	100
7, arrival	200

Line 3

Station	**Amount of embarking / alighting travellers**
1, departure	300
2, (through)	0
3, arrival	100
3, departure	100
4 (through)	0
5, arrival	100
5, departure	200
6, arrival	400

Line 4

Station	**Amount of embarking / alighting travellers**
6, departure	200
5, arrival	50
5, departure	100
4 (through)	0
3, arrival	50
3, departure	100
2, (through)	0
1, departure	300

Line 5

Station	**Amount of embarking / alighting travellers**
1, arrival	100
2, arrival	40
2, departure	40
3, arrival	50
3, departure	50
4, arrival	30
4, departure	30
5, arrival	40
5, departure	40
6, departure	100

Line 6

Station	**Amount of embarking / alighting travellers**
6, departure	100
5, arrival	40
5, departure	40
4, arrival	30
4, departure	30
3, arrival	50
3, departure	50
2, arrival	40
2, departure	40
1, arrival	100

10.3 Delay propagation in test case, scenario 1

Delay 5 minutes:

Delay propagation without dispatching actions:
Total first order delay: 24 minutes.
Total consecutive delay: 7 minutes.
Total delay of travellers: 2600 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	24	3,00
502	7	0,70

Advice without possibility for short turning:
No dispatching actions advised.

Advice with possibility for short turning:
-

Delay 10 minutes:

Delay propagation without dispatching actions:
Total first order delay: 64 minutes.
Total consecutive delay: 31 minutes.
Total delay of travellers: 8700 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	64	8,00
204	3	0,38
303	1	0,10
502	27	2,70

Advice without possibility for short turning:
No dispatching actions advised.

Advice with possibility for short turning:
-

Delay 15 minutes:

Delay propagation without dispatching actions:
Total first order delay: 104 minutes.
Total consecutive delay: 128 minutes.
Total delay of travellers: 22010 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	104	13,00
105	1	0,13
204	40	5,00
303	22	2,20
405	1	0,10
502	47	4,70
604	17	1,70

Advice without possibility for short turning:

1. Switch order train 102 and train 502 between stations 4 and 6.
   → ('Cancel scheduled overtaking of train 502 by train 102')

Delay propagation with dispatching actions:

Total first order delay: 104 minutes.
Total consecutive delay: 62 minutes.
Total delay of travellers: 15960 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	104	13,00
105	1	0,13
204	40	5,00
303	4	0,40
604	17	1,70

Advice with possibility for short turning:

-

Delay 20 minutes:

Delay propagation without dispatching actions:

Total first order delay: 144 minutes.
Total consecutive delay: 326 minutes.
Total delay of travellers: 42970 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	144	18,00
105	32	4,00
204	80	10,00
303	52	5,20
405	36	3,60
502	67	6,70

505	11	1,10
604	47	4,70
605	1	0,10

Advice without possibility for short turning:
1. Switch order train 102 and train 502 between stations 4 and 6.
   → ('Cancel scheduled overtaking of train 502 by train 102')
2. Switch order train 102 and train 303 between stations 3 and 5.
   → ('Let train 102 run behind train 303 between stations 3 and 5')
3. Switch order train 204 and train 604 between stations 4 and 3.
   → ('Cancel scheduled overtaking of train 604 by train 204')

Delay propagation with dispatching actions:
Total first order delay: 144 minutes.
Total consecutive delay: 132 minutes.
Total delay of travellers: 29110 minutes.

Delayed trains:

Train number	**Delay (min.)**	**Average delay (min.)**
102	144	18,00
105	32	4,00
204	80	10,00
303	9	0,90
505	11	1,10

Advice with possibility for short turning:
-

<u>Delay 25 minutes:</u>

Delay propagation without dispatching actions:
Total first order delay: 184 minutes.
Total consecutive delay: 632 minutes.
Total delay of travellers: 72960 minutes.

Delayed trains:

Train number	**Delay (min.)**	**Average delay (min.)**
102	184	23,00
103	18	2,25
105	72	9,00
204	120	15,00
205	17	2,13
303	82	8,20
305	1	0,10
306	4	0,40
405	86	8,60
502	87	8,70
503	14	1,40

505	31	3,10
604	77	7,70
605	23	2,23

Advice without possibility for short turning:
1. Switch order train 102 and train 502 between stations 4 and 6.
   → ('Cancel scheduled overtaking of train 502 by train 102')
2. Switch order train 102 and train 303 between stations 3 and 6.
   → ('Let train 102 run behind train 303 until arrival at station 6')
3. Switch order train 204 and train 604 between stations 4 and 3.
   → ('Cancel scheduled overtaking of train 604 by train 204')
4. Switch order train 105 and train 505 between stations 4 and 5.
   → ('Move scheduled overtaking of train 505 from station 4 to station 5')

Delay propagation with dispatching actions:
Total first order delay: 184 minutes.
Total consecutive delay: 229 minutes.
Total delay of travellers: 41190 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	184	23,00
105	72	9,00
204	120	15,00
305	1	0,10
405	25	2,50
503	1	0,10
505	10	1,00

Advice with possibility for short turning:
1. Let train 102 turn in station 3.
   → ('Hence, train 204 is cancelled between stations 6 and 3')

Delay propagation with dispatching actions:
Total first order delay: 49 minutes.
Total consecutive delay: 0 minutes.
Total delay of travellers: 37700 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	49	24,50

10.4 Delay propagation in test case, scenario 2

Note: when short turns are allowed, the advised dispatching actions remain the same.

Delay 5 minutes:
Delay propagation without dispatching actions:
Total first order delay: 19 minutes.
Total consecutive delay: 4 minutes.
Total delay of travellers: 1230 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	3	0,38
502	20	0,70

Advice:
No dispatching actions advised.

Delay 10 minutes:
Delay propagation without dispatching actions:
Total first order delay: 50 minutes.
Total consecutive delay: 62 minutes.
Total delay of travellers: 8330 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	32	4,00
103	3	0,38
202	9	1,13
502	68	6,80

Advice:
1. Switch order train 502 and train 102 between stations 3 and 4.
   → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4')
2. Postpone the arrival of train 502 at station 3 until train 202 has departed.

Delay propagation with dispatching action:
Total first order delay: 50 minutes.
Total consecutive delay: 5 minutes.
Total delay of travellers: 2870 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
502	55	5,50

Delay 15 minutes:

Delay propagation without dispatching actions:

Total first order delay: 98 minutes.
Total consecutive delay: 198 minutes.
Total delay of travellers: 24540 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	62	7,75
103	40	5,00
202	19	2,38
204	24	3,00
303	12	1,20
502	118	11,80
503	15	1,50
604	6	0,60

Advice:

1. Switch order train 502 and train 102 between stations 3 and 4.
   → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4')
2. Postpone the arrival of train 502 at station 3 until train 202 has departed.

Delay propagation with dispatching actions:

Total first order delay: 98 minutes.
Total consecutive delay: 0 minutes.
Total delay of travellers: 5080 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
502	98	9,80

Delay 20 minutes:

Delay propagation without dispatching actions:

Total first order delay: 148 minutes.
Total consecutive delay: 460 minutes.
Total delay of travellers: 52290 minutes.

Delayed trains:

Train number	Delay (min.)	Average delay (min.)
102	92	11,50
103	80	10,00
105	16	2,00
202	29	3,63
204	64	8,00
205	16	2,00

303	42	4,20
304	8	0,80
405	16	1,60
502	168	16,80
503	35	3,50
505	4	0,40
604	35	3,50
605	3	0,30

Advice:
1. Switch order train 502 and train 102 between stations 3 and 4.
   → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4')
2. Postpone the arrival of train 502 at station 3 until train 202 has departed.
3. Switch order train 502 and train 303 between stations 4 and 6.
   → ('Let train 303 overtake train 502 at station 4.')

Delay propagation with dispatching actions:
Total first order delay: 148 minutes.
Total consecutive delay: 25 minutes.
Total delay of travellers: 8930 minutes.

Delayed trains:

Train number	**Delay (min.)**	**Average delay (min.)**
303	5	0,50
502	168	16,80

Delay 25 minutes:
Delay propagation without dispatching actions:
Total first order delay: 198 minutes.
Total consecutive delay: 910 minutes.
Total delay of travellers: 96430 minutes.

Delayed trains:

Train number	**Delay (min.)**	**Average delay (min.)**
102	122	15,25
103	120	15,00
105	56	7,00
106	9	1,13
202	39	4,88
204	104	13,00
205	56	7,00
303	91	9,10
304	34	3,40
405	66	6,60
406	9	0,90
502	218	21,80
503	57	5,70

505	23	2,30
506	1	0,10
604	65	6,50
605	38	3,80

Advice:
1. Switch order train 502 and train 102 between stations 3 and 4.
   → ('Let train 502 run behind train 102 and cancel the overtaking of train 502 at station 4')
2. Postpone the arrival of train 502 at station 3 until train 202 has departed.
3. Switch order train 502 and train 303 between stations 4 and 6.
   → ('Let train 303 overtake train 502 at station 4.')

Delay propagation with dispatching actions:
Total first order delay: 198 minutes.
Total consecutive delay: 74 minutes.
Total delay of travellers: 16930 minutes.

Delayed trains:

Train number	**Delay (min.)**	**Average delay (min.)**
303	53	5,30
502	218	21,80
503	1	0,10