Minimizing the waiting time in a large-scale network of an advanced planning system

L. Plantagie

9 December 2009

Supervisors:
Dr. S. van Dijk
Prof. Dr. R.H. Bisseling

co-reader:
Dr. P.A. Zegeling

ORTEC
PROFESSIONALS IN PLANNING

Contents

Preface

This thesis is written to conclude my master study *Mathematical Sciences* at Utrecht University. It describes my work at ORTEC bv, a software company that provides advanced software to solve many types of planning problems.

I could not have written this thesis without my supervisor Steven van Dijk. I would like to thank him for his patience, his guidance, for the many 'moments' he was available when I asked him for advice, and for the pleasant time I have had at ORTEC bv. I would also like to thank Oscar Verzaal for his advice. I also thank Janneke Vos for her support and for improving my English.

Furthermore, I would like to thank Rob Bisseling from the department Mathematics at Utrecht University for his guidance.

Last but not least, I would like to thank my family and friends for all the support I have received.

Linda Plantagie

1 Introduction

Many companies in the transport sector distribute products throughout an area. They all have to transport a set of orders using the available resources, such as drivers, trucks and trailers. If this problem is complex, for example if the set of orders is large compared to the set of resources, or if there are many constraints on the orders or on the resources, then software is used to allocate the orders to the resources. ORTEC bv provides advanced software for many different problems related to planning, including the optimization of distribution processes. This thesis analyses and improves a part of this planning software.

Given a set of orders that should be planned together with a set of resources, a schedule is generated that specifies which resources are used, which actions are performed (for example picking up a package, driving to the next destination or delivering the package), and the finish instant of every action. This thesis concerns the part of the software that receives such a schedule and minimizes the amount of waiting time that is contained in this schedule. Throughout the optimization, neither the order of the actions nor the allocation of the orders to the resources is allowed to be changed. Only the times of the actions can be changed, as long as no constraints on the schedule are additionally violated due to these changes.

The method binary search is used to minimize the amount of waiting time in the schedule. For this method, a search domain is defined that contains the optimal start instant of some action. Under some conditions, this search domain can become bifurcated, and it is known that the software does not handle this situation correctly.

The two main goals of this thesis are stated below.

- *Analyse the software that is used to minimize the total waiting time in the schedule. Identify the conditions under which the search domain becomes bifurcated.*
- *Based on the results of the analysis in the previous point, either improve the existing algorithm or develop a novel algorithm.*

In part I, the terminology and basic principles used throughout this thesis are explained. Part II describes the problem and the algorithm used by ORTEC bv, together with the analysis of this algorithm. Part III contains the improvements that have been achieved for the algorithm, and the conclusion and subjects for further investigation are given in part IV.

Part I
Preliminaries

2 Plan

2.1 Structure of a plan

A typical problem in transport is the assignment of orders to resources such that the total costs are minimal. The solution to this problem is called a *plan* and contains the *schedule* with all the details of the planning.

2.1.1 Trips, sequences and actions

The schedule does not only contain information on which order is assigned to a particular resource, but also the start time of the pickup of the order, the address of the pickup, the address of the delivery and the driving time needed to travel from the pickup address to the deliver address. All this information is contained in smaller elements, called *actions*. An action is an operation that is performed during a consecutive period of time. The schedule consists of a collection of actions. It is possible that an action contains other actions. If it does, then this action is called the *parent* of the action(s) it contains. The contained actions are called the *children* of this action. If two actions have the same parent, they are called *brothers*. It may be counterintuitive, but it is possible that a child action has ended before his parent action starts. This will be explained in detail in section 2.2. The most frequently used actions are

Couple The couple is an action that adds one or more resources to the already coupled resources. Usually it contains the coupling action, information on the resource(s) to couple, possibly a travel action, a wait action or one or more acquire actions.

Coupling The coupling action is the actual couple of the resources.

Acquire Every resource that is coupled in the couple action, is acquired. The acquire action visualizes the need for the resource before the coupling can start. It is possible to couple more than one resource in the same couple action and some kinds of resources need another resource to be able to travel. Suppose, for example, a trailer, a truck and a driver are all coupled in the same couple action and they are currently all on a different location. Then the couple contains an acquire action that acquires all three resources. However, to move the trailer to the couple location, a truck and a driver are acquired. Hence the acquire action contains another acquire action for the truck and the driver. The truck needs the driver to travel, hence the second acquire contains a third acquire for the driver. This results in a tree with the acquire for the driver as a leaf. An acquire can contain a wait.

Stop The stop action is a container of consecutive actions at the same address with the same resources. It can contain pickups, delivers, drive throughs and waits. It can also contain a travel if the preceding action of the stop was at a different address.

Pickup The pickup is an action where at a specific address cargo is loaded into the resource combination.

Deliver The deliver is an action where the cargo is unloaded from the resource combination.

Drive through A drive through is planned when a resource combination is requested on an address for a different reason than a pickup or a deliver. It is uncommon to schedule drive throughs in transport, but they are used in other areas. For example if a carpenter is requested in a building for a task of one hour, a stop action is planned with a drive through of 1 hour.

Decouple The decouple is an action that removes resources from the already coupled resources. It contains the decoupling and possibly a wait or a travel from the address of the previous action to the address of the decoupling.

Decoupling The decoupling action is the actual decouple of the resources.

Wait The wait is a period of time between two other actions. A wait action can be a child of any of the above mentioned actions.

Travel The travel is an action where a resource combination moves from one address to another. If there is a limit on the duration of a consecutive period of driving, then it is possible that the travel not only contains periods of driving, but also periods of non-driving, called pauses or rests depending on their duration. The travel is always a child of one of the following actions: a couple, an acquire, a stop or a decouple.

The resources that are coupled at the start or end of an action are called the *start* resp. *finish* resources of that action. All couples, stops and decouples in the solution are divided into groups which are called *trips*. A trip consists of a series of actions, where the finish address and finish resources of the previous action equal the start address and resources of the next action. Two brothers in a trip cannot overlap in time, i.e. an action has to be completed before its succeeding brother starts. An exception are acquires. If two drivers are acquired in the same couple, then the two acquire actions are brothers and they can overlap. An example of a trip is given in Figure 1.

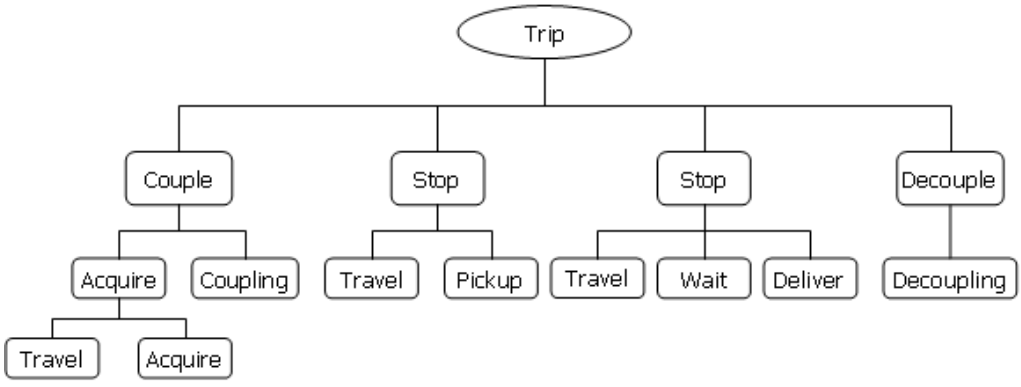

Figure 1: Example of a trip.

Although brother actions in a trip do overlap, children of an action can, and often will, overlap with their parents. For example, a pickup under a stop action starts and finishes within the start and finish instant of the stop action.

Every trip in the schedule is divided into smaller parts, called *sequences*. A sequence is the largest possible chain of consecutive actions in a specific order and with a special relationship. To describe this relationship, the notion of dependent actions is needed; two actions in a schedule are called *dependent* if there exists some relation between them. Examples of dependency relations are sharing the same resource or the same cargo. The dependency relations with the other actions in the schedule are assumed to be the same for every action in a sequence.
There are two actions that are only used to ensure that two consecutive sequences in a trip are consistent and contain enough information to stay consistent if anything in the trip is changed. These actions are called obtain and release.

Obtain The obtain action contains information concerning the resources already present in the trip, together with their availability at the start of the sequence. If a sequence is not the first sequence in the trip, then the start time of the obtain equals the finish time of the previous sequence.

Release The release action contains information on the resources which are not decoupled at the end of the sequence.

The actions obtain, acquire and release differ from the other actions. They do not represent an action a planner can work with, like a travel or a pickup. All three actions are not external, i.e. they are not visible for a driver in the field. The obtain and release are the first resp. last action of a sequence and are necessary to be able to work with sequences in a trip.
The actions of a sequence occur in the following order;

$$| \, O \mid C_1 \mid \cdots \mid C_n \mid S_1 \mid \cdots \mid S_m \mid D_1 \mid \cdots \mid D_k \mid R \, |,$$

where O is exactly one obtain, $C_1, \ldots, C_n$ are n couples ($n \geq 0$), $S_1, \ldots, S_m$ are m stops ($m \geq 0$), $D_1, \ldots, D_k$ are k decouples ($k \geq 0$), and R is exactly one release. So, given the order of all actions in a trip, the sequences of that trip can be determined unambiguously.

2.1.2 Dependent sequences and layering

Two sequences are called *dependent* if one of them contains an action with a dependency on an action from the other sequence. Dependencies occur for example when in two sequences the same resource is used or when two sequences belong to the same trip. The dependencies can be shown in a graph, called the *dependency graph*. In the graph, all sequences are represented by nodes and the dependencies are represented by directed arcs. If a trip consists of two sequences, then there is a directed arc in the dependency graph from the node representing the first sequence of this trip to the node representing the second sequence of this trip.
All the sequences in the schedule are placed in *layers*. A layer is a collection of sequences that have no dependencies among them. All the layers are numbered. The sequences are distributed over the layers in such a way that if there exists a directed arc between two nodes in the dependency graph, the sequence represented by the node at the tail of the arc is placed

in a lower numbered layer than the sequence represented by the node at the head of the arc. When the layers are placed from left to right starting with the lowest numbered layer, all the dependencies are from left to right. This implies that a change in sequence s in layer $i \in \mathbb{N}_{>0}$ can only affect the actions in sequence s and actions in higher numbered layers. Therefore, calculating from left to right is an effective way to recalculate every sequence that should be updated exactly once.

In Example 2.1, the sequences and dependencies of a schedule are determined, and the principle of layering is demonstrated.

Example 2.1. Consider a schedule with three trips, as shown in Figure 2. The couple and decouple actions are shown. A number of stops can be planned between every succeeding couple and decouple action. The resources that are (de)coupled are denoted between brackets. Hence trip 1 starts with coupling the resources 1, 2 and 3 followed by a number of stops and decoupling the resources 1 and 2.
Notice that some resources are used in different trips, resulting in dependency relations between actions of different trips.

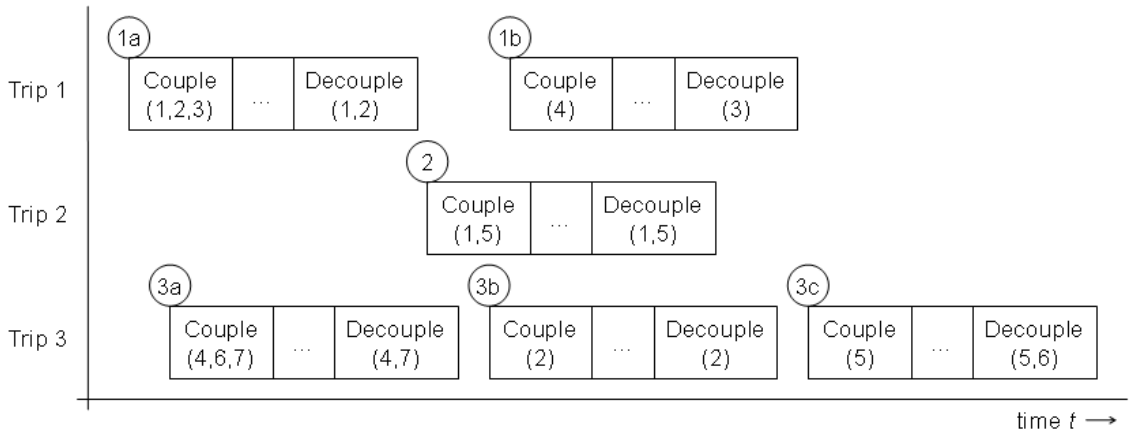

Figure 2: A schedule with three trips.

Since the order of the actions in a sequence is fixed, the sequences of trip 1 can be determined unambiguously. The first sequence of this trip, referred to as sequence $1a$, is
$\mid O \mid C\ (1,2,3) \mid S_1 \mid \ldots \mid S_n \mid D\ (1,2) \mid R \mid$, where $n \geq 0$. Here O is the obtain and R is the release. Since sequence $1a$ is the first sequence of the trip, the obtain has no start resources. Since resource 3 is coupled but not decoupled, the release contains resource 3 as its finish resource. The second sequence of trip 1, referred to as sequence $1b$, is
$\mid O \mid C\ (4) \mid S_1 \mid \ldots \mid S_m \mid D\ (3) \mid R \mid$, where $m \geq 0$, O contains start resource 3 and R contains finish resource 4. Likewise, it is easily seen that trip 2 consists of one sequence, referred to as sequence 2, and trip 3 consists of the three sequences $3a$, $3b$ and $3c$.

There are many dependencies between the sequences. For example, sequence $1a$ and $1b$ are dependent, since they belong to the same trip. Since sequence $1a$ must be planned before sequence $1b$, the dependency is represented in the dependency graph by a directed arc from

sequence $1a$ to sequence $1b$. Likewise, the sequences $3a$ and $3b$, $3a$ and $3c$, and $3b$ and $3c$ are dependent. Also sequence $1a$ and 2 are dependent, since they have a common resource. The same is true for sequences 2 and $3c$, $1a$ and $3b$, and $3a$ and $1b$. The dependency graph of this schedule is shown in Figure 3.

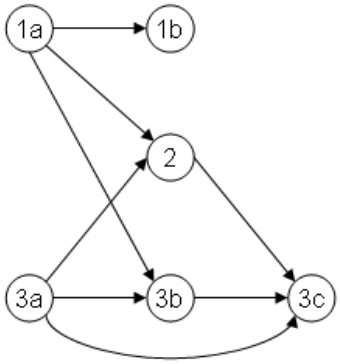

Figure 3: Dependency graph of a schedule.

Since the tail of an arc in the dependency graph is placed in a lower numbered layer than the head of that arc, the only sequences that can be placed in the first layer are the sequences that are not a head of any arc in the dependency graph. An example of a correct layering is shown in Figure 4.

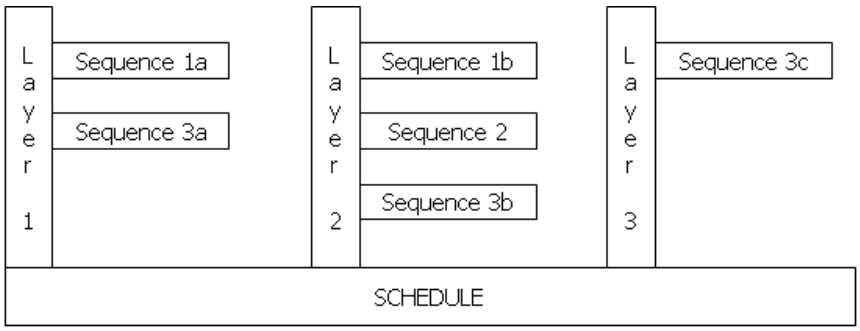

Figure 4: The structure of a schedule.

Remark When a trip is removed from a schedule, the layers are not determined again. As a result, the distribution of sequences over the layers is not unique and it is not guaranteed that a sequence is contained in the layer with the lowest possible number.

2.2 Updating the finish instants in a plan

For every action in the schedule a finish instant is determined. The start instant of an action is not calculated and if, for some reason, a start instant is requested, the finish instants of other actions are used to deduce it. If the finish instant of an action is changed, the finish instants of other actions may need to be changed as well. As discussed in section 2.1.2, only actions that belong to the same sequence or to a sequence in a layer with a higher number may need to be updated. Since a planner cannot continue before the calculations are completed, it

is important to update only those actions that can be affected by the change. The mechanism that ensures and maintains the consistency of a schedule is referred to as the *engine*.

2.2.1 Network of observers

To determine which actions in the schedule should be updated, *observers* are used. An observer belongs to an action and can observe, for example, other actions. If such an action changes, the observer is triggered and can execute some code to update the action it belongs to. Each action in the plan has its own network of observers. Observers typically watch the children of the action (if they exist), the parent of the action, the previous brother and the previous action(s) of the coupled resource(s). Since only the observers of actions that could change are triggered, unnecessary calculations are avoided as much as possible.

2.2.2 The duration of an action

Given the start of an action, its finish instant cannot always be determined as the sum of the start instant and the duration of the action, since the duration of an action is not always known. The durations of the following actions are known: coupling, decoupling, pickup, deliver and drive through. The duration of a travel can differ due to a pause or rest, but the total driving time is known. The duration of an action containing children is not always known. For example, the duration of a stop can change if it contains a deliver and a pickup with a wait action between them. If an action contains other actions, then the finish instant of the parent action equals the finish instant of its last child. Hence given the start instant of an action, its finish instant can be determined.

2.2.3 Generating wait actions

To determine the start instant of an action, often the finish instant of the preceding action in time is used. However, there are situations where this finish instant cannot be used as the start instant. Suppose for example that the start instant of a pickup needs to be recalculated because its preceding travel ends an hour earlier. Then it is possible that the address of the pickup is not open at the earlier finish instant of the travel. In this situation, a wait is generated from the finish of the travel until the next opening of the address. This wait has the special name *wait for opening time*. If the address is open, but the cargo is not yet ready to be loaded, another wait action called *wait for task available* is generated. The same is true for couple actions: if the previous action of a couple starts earlier but the couple itself cannot start earlier because its resources are not yet available, a *wait for resource available* is generated. If a planner manually states that the pickup should start at a particular time, and the travel ends earlier, then a wait is generated as well.

2.2.4 Updating a parent action

As stated in section 2.2.2, the duration of a parent action, i.e. an action containing other actions, is not always known, but its finish instant equals the finish instant of its last child. Although it might be expected, it is not always true that the start instant of the parent action equals the start instant of its first child action. There are several situations where this is not correct; the most common are

- the first child is a travel or a wait action.
- the first child that is not a travel or a wait action contains a wait action.

These situations are considered in Example 2.2.

Example 2.2. Consider a trip that starts with a couple, followed by three stops (they are all parent actions) and a decouple, as shown in Figure 5.

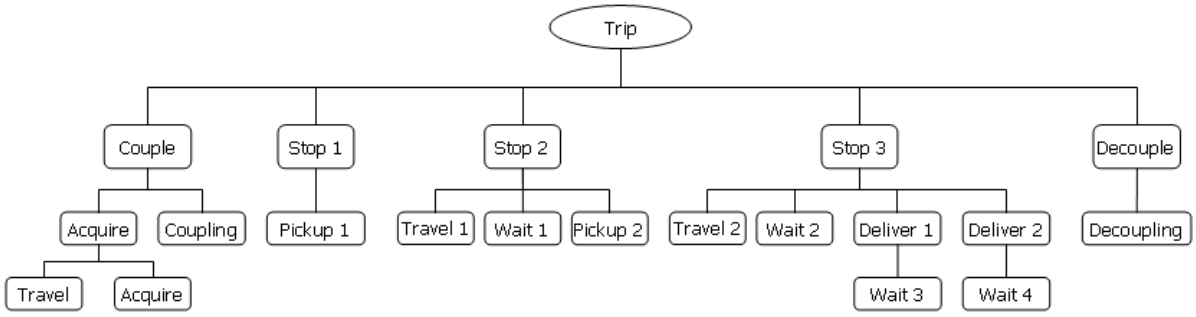

Figure 5: Visualization of the trip in Example 2.2.

The relations in time between the actions are not visible in Figure 5. For example, travel 1 is a child of stop 2, but the stop cannot start before the travel is completed. The same is true for wait action 1. The relations in time are shown in Figure 6, where the acquire tree is represented by one action (Acquires) for simplicity.

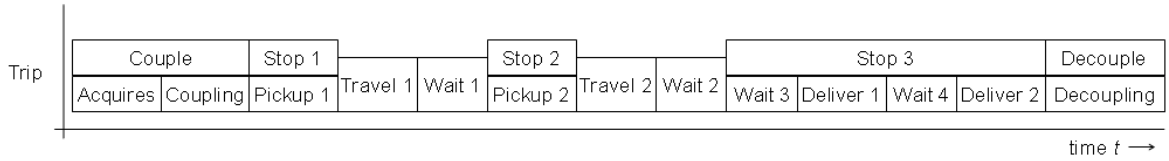

Figure 6: Visualization of the trip in Example 2.2 using the start and finish times of its actions.

Note that wait action 3 is a child of deliver action 1 and that its start equals the start of stop action 3, while wait action 2 must be completed before stop action 3 can start.

The observers in the networks of the actions are used to update the finish instants of the actions if anything changes in the schedule. In this example, an observer in the network of any action except a parent action is triggered if the finish instant of the preceding action in time is changed. For example, the preceding action of wait 1 is travel 1 and if the finish of this travel is changed, then the finish of wait 1 is recalculated. If the preceding action is a parent action, then the finish of the parent action is observed by an observer in the network of the succeeding action.

This mechanism ensures that the instants in the schedule remain consistent if anything changes in the schedule, for example if an extra order is planned or an order is unplanned, i.e. deleted, in an already existing trip.

3 Restrictions

A planner cannot just randomly put actions somewhere in a schedule, since every schedule has to fulfill certain constraints. These constraints are translated to *restrictions* which can be tested on the schedule. Most restrictions are tested on a single action, like a restriction that requires that a pickup on a specific address should start and end within the opening times of that address. If a restriction should be tested on an entire trip or schedule instead of on a single action, then this restriction is called a *schedule* restriction. Examples of schedule restrictions are a limit on the maximum driving time and a limit on the maximum number of stops in a trip.
Restrictions can be divided into two groups.

- The most important group consists of all the *necessary* restrictions, which should always be fulfilled. An example of such a restriction is requiring that at least one finish resource is present at the end of every stop. Requiring that a pickup is planned before the corresponding deliver is also a necessary restriction.
  If in an existing plan a planning command is given, for example the planner wants to have an extra stop planned, and a necessary restriction is violated, then the extra stop will not be planned.

- The second group consists of all *optional* restrictions. If during a planning command an optional restriction is violated, then a warning message is generated to inform the planner of the violation. Examples of optional restrictions are a finish time window on a task and a restriction on capacity.
  If the planner decides to allow the violation of an optional restriction, then the planning command is executed. The tolerated value of the violation is stored within the action in the schedule. If the restriction is violated in the future, then the value of the violation is compared with the tolerated value. The planner will not be notified if the violation is less then the tolerance and in that case the tolerance is replaced by the current violation.

An example of tolerating a violation of an optional restriction and updating this tolerance is given in Example 3.1.

Example 3.1. Suppose a trip is planned with a pickup action, but the capacity of the coupled trailer is not sufficient to transport the order. Then the restriction on capacity will generate a warning. If the planner decides to tolerate the violation, then replacing the current order by an order that uses slightly less capacity will not result in a new warning. If instead the order is replaced by an order using (slightly) more capacity, then a warning is generated to inform the planner of the increased violation. If the planner decides to couple a different trailer with sufficient capacity, then the restriction is no longer violated and the tolerated value is removed. If, afterwards, the planner decides to switch back to the original situation with the trailer with insufficient capacity, then the warning is generated again to notify the planner of the new violation and the planner has to tolerate it again.

3.1 Time-concerning restrictions

Some restrictions concern resources or addresses, others are related to time or capacity. If a restriction is related to resources, it has the label ‘resource’. Likewise, there are labels for the

other categories, and a restriction can have more than one label. All restrictions related to time have the label 'instant', and these are the restrictions discussed in this section.

Examples of restrictions that concern time are address opening, time window on a task and time window on the start and finish instants of the trip. A *time window* contains an approved interval. For example, a time window on a task specifies the interval in which the task is allowed to be carried out. A time window on an address specifies the opening times of that address. A time window is limited in use, since it only specifies one allowed interval. If open or closed intervals occur frequently, *calendars* are used. A calendar consists of one or more *versions*. A version has a start time and can have an end time, and can contain

- one or more days of the week. On such a day, one or more open intervals can be specified. If a Monday is specified with an open interval from 8 am until 5 pm, then every Monday between the start and end of the version has this open interval.
- additional opening periods that are not reoccurring. They are denoted as *extra opens*.
- excluded opening periods, denoted by *extra closed*. They are included in the calendar if an open interval in a reoccurring series should be excluded.

If a restriction that uses a time window or calendar is tested, it is checked if the action starts and ends within an open interval.

3.1.1 Behavior of time-concerning restrictions

Let r be a time-concerning restriction, and suppose that this restriction is tested on an action A. Let the function $f_A^r(t)$ be defined as

$$f_A^r(t) = \begin{cases} 1, \text{if the violation of restriction } r \text{ on action } A \text{ is within the tolerance,} \\ 0, \text{otherwise,} \end{cases} \tag{1}$$

where t is the start instant of action A. For an arbitrary restriction that concerns time, the graph of f_A^r can contain many discontinuities, as shown in Figure 7.

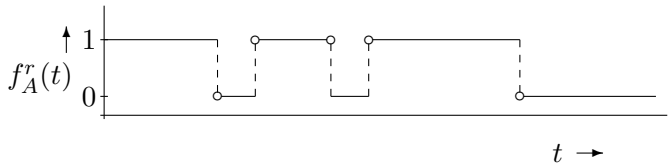

Figure 7: Graph of f_A^r as a function of the start instant of action A.

Let R be the number of time-concerning restrictions, and let $f_A(t)$ define the function that returns 1 if action A is allowed to start at time t, i.e. no restriction is additionally violated, and returns 0 otherwise. Then

$$f_A(t) = \prod_{1 \leq r \leq R} f_A^r(t). \tag{2}$$

If the schedule is optimized to reduce the amount of waiting time, then no additional violation of a restriction is allowed. This implies that the value of f_A is not allowed to decrease when the start instant of an action is changed during the optimization.

3.2 Driver legislation

An important group of restrictions that concern time arises from constraints on the duration of the work and travel time of a driver. By law, a driver is not allowed to travel long distances without taking any breaks. Short breaks are called *pauses*, while long breaks are called *rests*. As for any regular job, also the total time a driver is allowed to work without rest is limited. The total work consists of both driving time and non-driving time, for example loading and unloading the truck. By *driver legislation* we denote the rules concerning the amount of allowed driving time, allowed working time, compulsory pause time and compulsory rest time. The following three parameters are used to monitor the status of the driver: *working time till rest* (WTR), *driving time till rest* (DTR) and *driving time till pause* (DTP). Based on the regulation of the European Parliament [1], the values used in this thesis are

$$\begin{aligned} DTP &= 4.5\,\text{hours}, & P &= 3/4\,\text{hours}, \\ DTR &= 10\,\text{hours}, & R &= 9\,\text{hours}, \\ WTR &= 15\,\text{hours}, \end{aligned}$$

where P is the *minimum duration of a pause* and R is the *minimum duration of a rest*.
Every action where the driver is involved, contains a *driver status* (DS) with the most recent values of the parameters. At the end of the action, the driver status is updated and the next plan action will receive this updated driver status.
If during an action any of the values in the driver status reaches or drops below zero, the driver should pause or rest at the start of the first travel that is encountered after this action, unless the action itself is a travel, then the driver pauses or rests immediately.

Plan actions can be divided into four categories.

1. Plan actions that only count as working time. Examples of such actions are couples, decouples, drive throughs, pickups and delivers.

2. Plan actions that may count as a rest or pause. A typical example is a wait. Based on its duration a wait can count as a pause or a rest. An action in this category also counts as working time.

DS at start of A	DS after action A with duration T		
	$T < P$	$P \leq T < R$	$R \leq T$
DTP	DTP	4.5	4.5
DTR	DTR	DTR	10
WTR	WTR - T	WTR - T	15

Table 1: Change in driver status for an action in category 2

3. Plan actions that count both as working time and as driving time. Only travels belong to this category. Figure 8 shows the calculation of the actions in a travel. TLD denotes the total time that still needs to be driven in this travel. The parameter Time is initialized at the start of the travel. After every action in the travel, this parameter is increased by the duration of this action. Hence at the end of the travel, Time denotes the finish time of the travel.

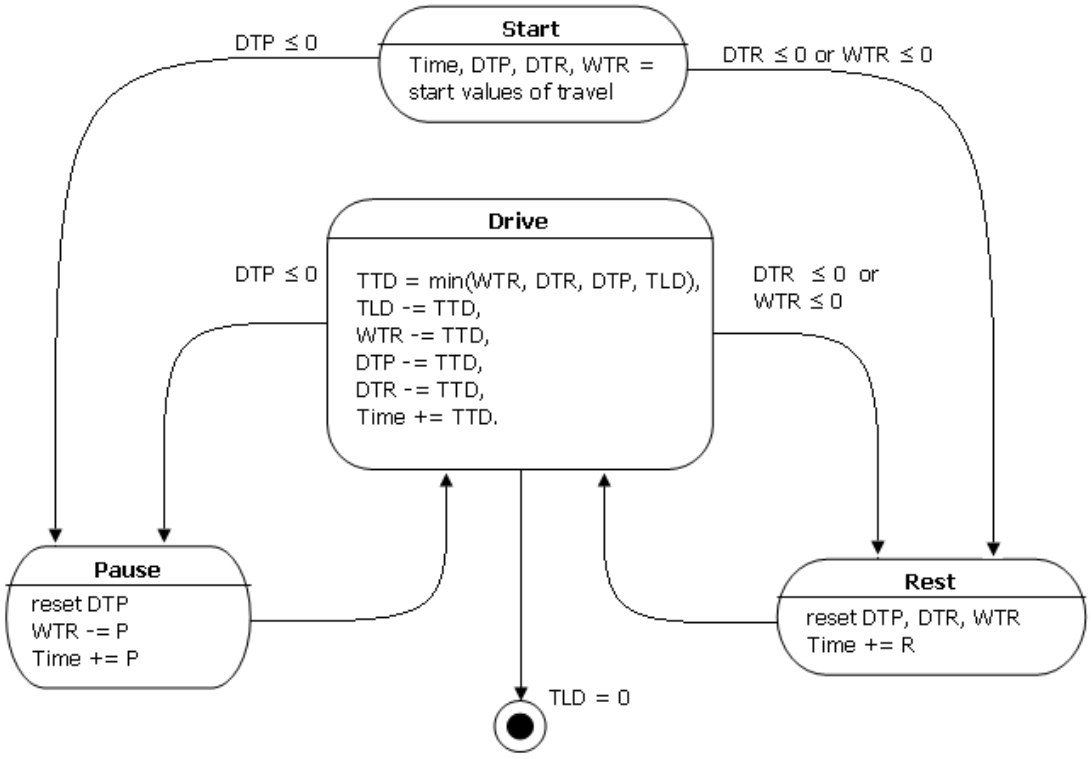

Figure 8: Calculation of the actions in a travel

If the value of DTR or WTR drops at or below zero, a rest is planned. If these values are positive and the value of DTP is at or below zero, a pause is planned. Otherwise a drive action is planned with a maximum duration, i.e. until a rest or pause should be planned or until the value of TLD equals zero. The driver status at the end of the travel is given by the last calculated values for DTP, DTR and WTR.

4. Plan actions that have no effect on the driver status, i.e. all plan actions that do not belong to one of the other three categories. Examples are an acquire, an obtain and a release.

In Example 3.2, the effect of actions on the driver status is demonstrated for actions that belong to different categories. This example also demonstrates the use of the driver status to calculate the different parts of a travel action, i.e. driving, pausing and resting.

Example 3.2. Consider the trip shown in Figure 9, where
C = couple, Dl = deliver, P = pickup,
D = decouple, Dr = drive, Ps = pause.
⊙ = corresponds to a DS in Table 2.

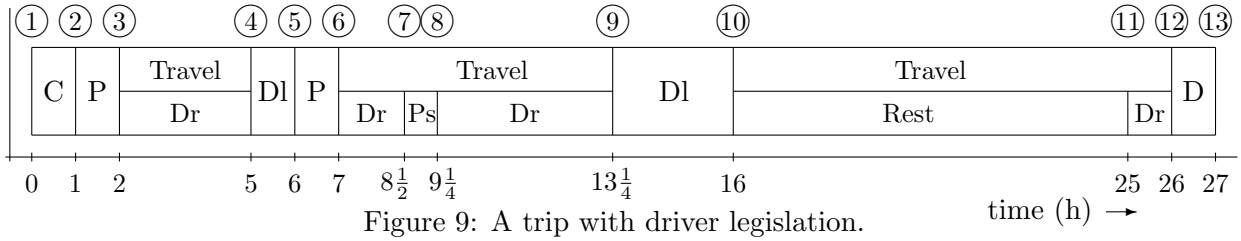

Figure 9: A trip with driver legislation.

The numbers in the circles above the trip correspond to the numbers in the first row of Table 2 and represent a driver status. Driver status 7, 8 and 11 are not really created in the schedule, but are included here to show what happens to the parameters at different stages of a travel. The actions in the trip belong to three different categories. Their effect on the driver status is therefore not identical. Before and after every action, the driver status is given in Table 2.

DS	1	2	3	4	5	6	7	8	9	10	11	12	13
DTP (h)	4.5	4.5	4.5	1.5	1.5	1.5	0	4.5	0.5	0.5	4.5	3.5	3.5
DTR (h)	10	10	10	7	7	7	5.5	5.5	1.5	1.5	10	9	9
WTR (h)	15	14	13	10	9	8	6.5	5.75	1.75	-1	15	14	13

Table 2: The driver status at several moments in the trip.

Part II
Analysis

4 Description of the problem

4.1 Mathematical description

Assume we are given a schedule with N trips. An important way of reducing the cost of this schedule is to reduce the sum of all waiting times in the schedule, since this reduces the total time the resources are working.

- Let S_n denote the start instant of trip n, i.e. the start instant of the first external action of trip n.
- Let F_n denote the finish instant of trip n, i.e. the finish instant of the last external action of trip n.
- Let $\mathcal{F}_n$ denote the finish instant of trip n in the original schedule.
- Let T_n denote the duration of trip n, defined by

$$T_n = F_n - S_n.$$

Minimizing the sum over all waiting times in the trip is equivalent to minimizing the sum over all trip durations. During the optimization, it is not allowed that a restriction is violated more than before the optimization started. Hence,

★ the constraints on the schedule are constraints on the minimization problem.

To formulate the constraints, the following notations are used.

- Let $\mathcal{A}_n$ denote the set of all actions in trip n that are not wait actions.
- Let $\mathcal{R}$ denote the set of restrictions that are tested on all actions with a changed start instant.
- Let $V_{(R,A)}$ denote the violation for restriction $R \in \mathcal{R}$ on action $A \in \mathcal{A}_i$ $(1 \leq i \leq N)$.
- Let $\mathcal{V}_{(R,A)}$ denote the tolerated violation for restriction $R \in \mathcal{R}$ on action $A \in \mathcal{A}_i$ with $(1 \leq i \leq N)$ in the original schedule.
- Let $V_{(R,S_n,A)}$ denote the violation for restriction $R \in \mathcal{R}$ on action $A \in \mathcal{A}_i$ $(1 \leq i \leq N)$ if trip n has start instant S_n.

The minimization of the waiting times in the schedule can be formulated as

$$\begin{aligned} & \min \quad \sum_{n=1}^{N} T_n, \qquad \text{s.t.} \\ V_{(R,A)} \quad & \leq \quad \mathcal{V}_{(R,A)}, \quad \forall R \in \mathcal{R},\ \forall A \in \mathcal{A}_i, \forall_{1 \leq i \leq N}. \end{aligned} \tag{3}$$

The algorithm used to solve this problem is called the *beautify*. The approach of the beautify is to reduce this minimization problem to N smaller maximization problems, one for each trip. The fundamental idea is that the duration of a single trip will be minimized if its finish instant is fixed and the start instant is delayed as much as possible without causing additional violations.

★ An additional constraint for the maximization problem for trip n is that the finish instant of trip n is not allowed to be delayed.

The maximization problems are given by (4).

Let $1 \leq n \leq N$ be fixed.

$$\begin{aligned} & \max \quad S_n, \qquad \text{s.t.} \\ F_n \quad & \leq \quad \mathcal{F}_n, \\ V_{(R,S_n,A)} \quad & \leq \quad \mathcal{V}_{(R,A)}, \quad \forall R \in \mathcal{R},\ \forall A \in \mathcal{A}_i, \forall_{1 \leq i \leq N}. \end{aligned} \tag{4}$$

The beautify solves the minimization problem (3) by solving at most N times the maximization problem (4). Every trip is allowed to be optimized at most once when the beautify is applied to the schedule.

4.1.1 Remarks

- There is a second reason for bounding the finish instant of the trip that is optimized. If other trips depend on trip n, then a delay of the finish of trip n due to the later start of the trip can result in a high number of actions in these dependent trips whose start instants need to be recalculated. An upper bound on the finish instant of the trip prevents recalculating a major part of all actions in the schedule after changing the start instant of a single trip.

- Although the upper bound on the finish instant of trip n reduces the number of actions whose start instant will change during the optimization of trip n, it is still possible that actions in trip $k \neq n$ are delayed when the start instant of trip n is delayed. Therefore, the restrictions need not only be tested on the actions in trip n, but on every action in the schedule that has changed.

- Also note, that the constraint on the finish instant of the trip only holds for trip n, hence it is possible that the finish of trip k is delayed during the optimization of trip n.

- The problem of finding a schedule with minimal waiting time is now rewritten into two subproblems: finding the optimal order for solving the N smaller maximization problems, and optimizing a single trip.

4.2 Related work

The optimization problem (3) is part of the larger problem of finding an optimal allocation of resources to a given set of orders. This problem is related to vehicle routing problems (VRPs) using pickup and deliver actions. In VRPs, introduced by Dantzig and Ramser [2], clients are supplied by a fixed fleet of vehicles. As argued by Bodin et al. [3], VRPs are NP-hard. Many heuristics have been developed for the basic VRPs, as well as for a generalization using time windows on the deliver actions, for example by Laporte et al. [4], and Baker et al. [5]. Contrary to the VRPs studied in these works, multiple depots can be used in the optimization problem in this thesis. Furthermore, the capacity of the vehicles is limited and varies per vehicle. These generalizations of the vehicle routing problem are, amongst others, studied by Savelsbergh et al. [6]. The heuristics for solving VRPs often use genetic algorithms or local search techniques, such as tabu search and simulated annealing, to optimize an initial feasible solution. These heuristics obtain a solution by changing the structure of the scheduling. This means that actions can be assigned to other trips or that the order of the actions within a trip is changed. During the optimization in this thesis, however, only the instants of an action are allowed to change.
Secondly, the complexitiy of the allocation problem is highly increased by the constraints on the working hours of a driver. Although these constraints are important for many real-life scheduling problems, it has received very little attention in the literature. Heuristics for a more general version of the VRP with time windows on the deliveries and regulations on the drivers' working hours are presented in Goel et al. [7] and Bartodziej et al. [8]. Again, structure changes are used to optimize an initial feasible schedule. Therefore, these heuristics cannot be used for solving problem (3).

Although job scheduling and VRPs are closely related, it is not possible to convert optimization problem (3) to a job scheduling problem. An attempt would be to associate trips with machines and actions with jobs. The processing times of jobs that represent travel actions depend on the start instants of preceding jobs with the same driver. Although literature is known on allocating manpower in job shops, for example Daniels et al. [9], the processing times of the jobs in these problems varies with the amount of allocated resources instead of the amount and location of idle time on a machine. Results of these studies can therefore not be applied in this thesis.

4.3 The order to optimize the trips

To solve optimization problem (3), the beautify optimizes the trips in the schedule one by one. If there are no dependencies between the trips, the order in which the trips are optimized does not affect the solution of the maximization problems. However, if dependencies exist between the trips, the order can affect this solution. This is illustrated in Example 4.1.

Example 4.1. Consider the schedule consisting of two trips as shown in Figure 10. The arrow indicates a dependency relation between deliver action Dl_1 and pickup action P_2. The action at the tail (Dl_1) must be completed before the action at the head of the arc (P_2) can start.
First optimizing trip 1 (by solving maximization problem (4)) and then optimizing trip 2 will remove all the wait actions in the schedule. Optimizing the trips in Figure 10 in the reversed

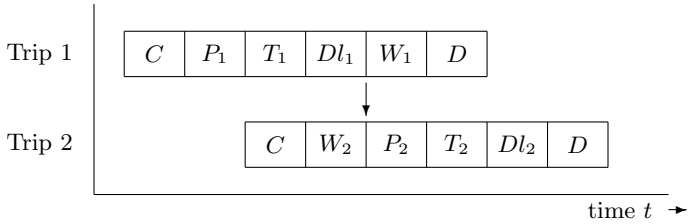

Figure 10: Two trips with one dependency relation.

order will remove wait action W_2 and W_1, while a new wait action will be created after the couple action in trip 2. This new wait action has the same duration as the just removed wait action W_1. Hence the optimal order to optimize the trips in this schedule is $1 \rightarrow 2$.

Hence the order in which the trips in the schedule are optimized can affect the solution of the minimization problem (3).

4.4 Optimizing a single trip

Suppose trip n is chosen for optimization. Since the finish of trip n is fixed, an obvious approach to maximization problem (4) is calculating backwards.

4.4.1 Calculating backwards

The actions of a trip can be place in chronological order, as is shown in Figure 6 of Example 2.2. This order will be used in the method of calculating backwards.

- Let S_A resp. F_A denote the start resp. finish instant of action $A \in \mathcal{A}_n$.

The finish instant of the trip that is optimized is fixed. Therefore, the latest finish instant of the last action A of the trip is known. If the duration of this action can be determined, then its latest start instant $S_A^{\max}$ can be calculated. It is possible that the action cannot start at this instant, due to an additional violation of a restriction. Then a lower bound and an upper bound for S_A are known and finite, since S_A is at least the value of the start instant of A in the original schedule and $S_A \leq S_A^{\max}$. The latest allowed start instant of A can be found using a search algorithm, for example binary search. This latest allowed start instant of A is the latest possible finish instant of the action that precedes action A. By repeating the above steps, the latest start instant of every action in the trip can be calculated. The latest possible start instant of the trip will then equal the latest possible start instant of the first external action of the trip.

Remark To determine the start instant of a parent action using this method, first determine the start instants of all its child actions, starting with the last child. The start instant of the parent action can then be deduced from the start instants of the children.

Problem with calculating backwards
Calculating backwards assumes that, after optimizing the start instant of action A, this action will not change during the optimization of any action $A' \in \mathcal{A}_n$ with $F'_A < F_A$. However, this assumption is not always true when the restrictions are included in the set of constraints of

the optimization problem. Since the duration of a travel depends on the preceding actions, including wait actions, a decrease of such a wait action will change the driver status at the start of the travel. It is possible that the driver suddenly needs to pause or rest during the travel while this was previously not necessary. But since the travel and all succeeding actions were already planned as late as possible, an extra pause in the travel will result in a delay of the finish instant of the trip. Hence, the trip is suddenly not feasible any more, and there is no guarantee that the method of calculating backwards will finally result in a feasible solution. Therefore, this method cannot be used by the beautify.

4.4.2 Assumptions

As described in section 4.4.1, the method of calculating backwards cannot be used in the beautify to find the optimum of the maximization problem (4). The algorithm that is used in the beautify to find the optimum of (4) is binary search (see [10]). It can only be used if the following assumptions are made.

Assumption 4.1 (Behavior of restrictions)**.** *Let $H(x)$ be the Heaviside function, i.e.*

$$H(x) = \begin{cases} 0, & \text{if } x < 0, \\ 1, & \text{if } x \geq 0. \end{cases}$$

Let A be an action in the schedule and let r be a restriction on the schedule. Then

$$\exists\, \hat{t}_A^r \text{ s.t. } f_A^r(t) = 1 - H(t - \hat{t}_A^r).$$

The constant $\hat{t}_A^r$ is the maximum start instant of action A such that there is no additional violation of restriction r. Assumption 4.1 immediately leads to Corollary 4.1.

Corollary 4.1. *Let t be the start instant of action A. Then*

$$\exists\, \hat{t}_A \text{ s.t. } f_A(t) = 1 - H(t - \hat{t}_A).$$

Proof. Take $\hat{t}_A = \min_{1 \leq r \leq R} \hat{t}_A^r$. □

The graph of f_A is shown in Figure 11.

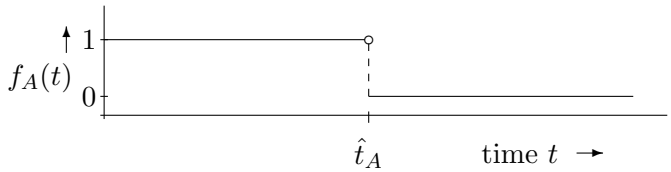

Figure 11: Assumed behavior of f_A as a function of the start instant of action A.

Hence, if a restriction is violated for start instant t of action A, then this restriction will be violated by at least the same amount for every start instant $t' > t$.

The assumption on the behavior of restrictions is not enough to be able to apply binary search in the beautify. Also a relation is needed between the start instants of the actions in the schedule. This relation is stated in Assumption 4.2, where the following notation is used;

- Given trip i, let $S_{i,A}$ denote the start instant of action $A \in \mathcal{A}_i$.

Assumption 4.2 (Start instants of actions)**.** *Given a schedule $\mathscr{S}$ with N trips. Suppose trip $n \in N$ is optimized. Let S_n and S'_n denote two start instants of trip n. Then*

$$S'_n > S_n \qquad \Longrightarrow \qquad S'_{(i,A)} \geq S_{(i,A)}, \quad \forall A \in \mathcal{A}_i \ , \quad \forall 1 \leq i \leq N.$$

Hence, if the start of a trip is delayed, then it is assumed that every action of the schedule starts at the same time or later than it started before the start of the trip was delayed.

4.4.3 Binary search

To start a binary search, the *search domain* needs to be determined, i.e. the range of all start times that are a candidate for the optimum.

- Let LB denote the lower bound of these candidates, i.e. the left boundary of the search domain.
- Let UB denote the upper bound of these candidates, i.e. the right boundary of the search domain.

During a binary search step j, a start time S_n^j of trip n is chosen in the interior of the search domain. Given this start instant, the start instants of all the actions in the schedule are updated. When the constraints are checked, the following situations can occur;

- The finish of the trip is delayed. In this situation, the start instant of the trip is rejected. By Assumption 4.2, any later start instant of the trip will result in a start instant of the last external action which is at least as late as its current start instant. Hence a later start instant of the trip will also result in a delay of the finish instant of the trip. Therefore, S_n^j is a better right boundary of the search domain.
- The finish of the trip is not delayed. Then the restrictions are tested on all the actions that are updated.
  - If there is a constraint that is additionally violated on action A with start instant S_A, then S_n^j is also rejected. Due to Assumption 4.2, a start instant $S'_n > S_n^j$ will result in a start instant S'_A of action A with $S'_A \geq S_A$. By Assumption 4.1, the same constraint will be additionally violated by at least the same amount as for S_n^j. Therefore, also in this situation is S_n^j an upper bound of the search domain for the largest possible start instant of trip n, i.e. $UB = S_n^j$.
  - If all the constraints of the maximization problem (4) are satisfied, then the start instant is accepted and all earlier start instants can be removed from the search domain since they are worse than S_n^j. Hence the start instant is a better left boundary for the search domain, i.e. $LB = S_n^j$.

Hence, the search domain is decreased by every step in the binary search and, since the search domain is discrete, eventually there is only one start instant left in the search domain. This instant is the solution to the maximization problem (4).
The steps 1-4 of a binary search are shown in Figure 12.

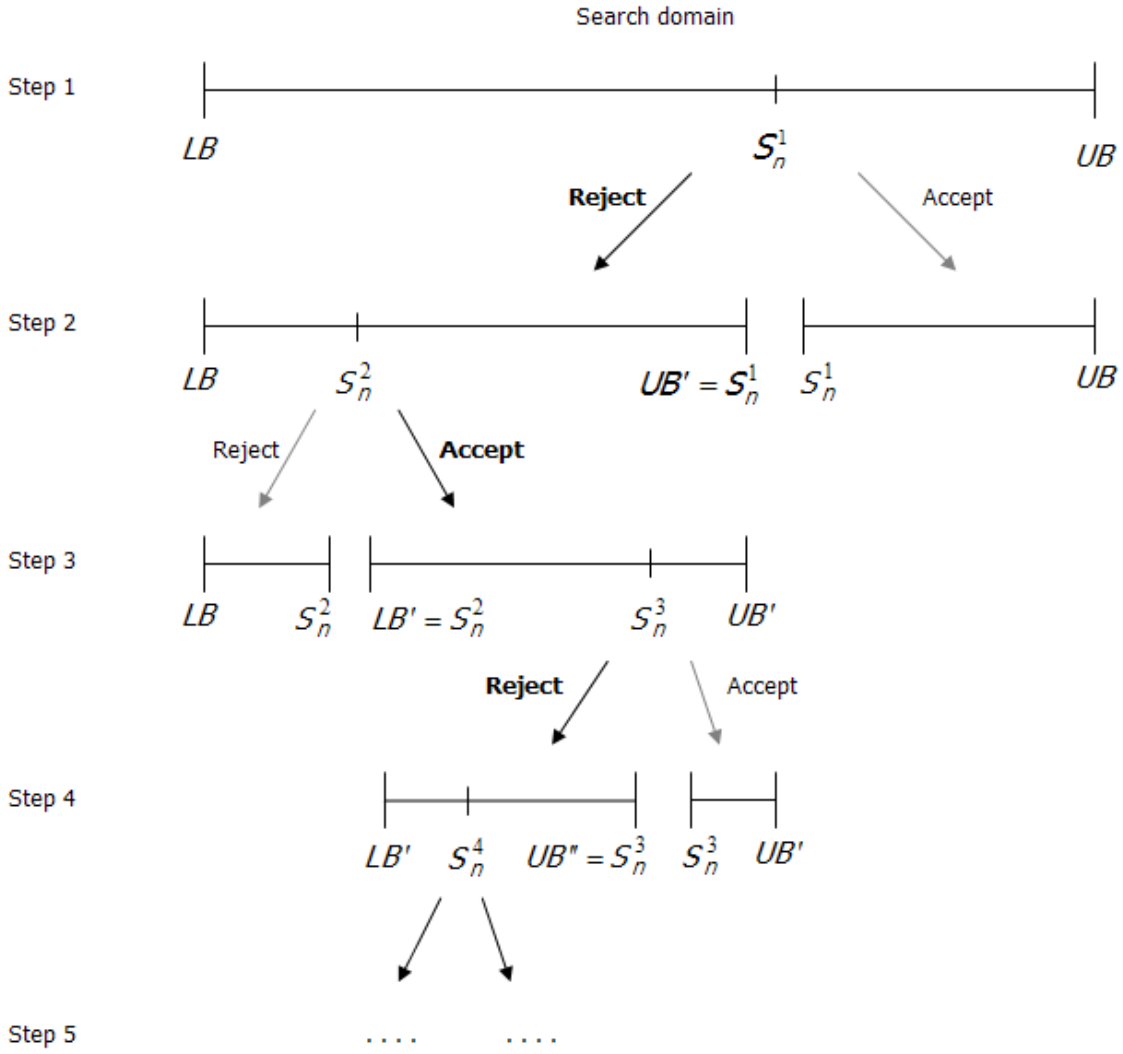

Figure 12: Visualization of the binary search algorithm.

If every new start instant S_n^j is chosen at the middle of the search domain, i.e.

$$S_n^j = \lfloor \frac{UB - LB}{2} \rfloor,$$

then the size of the domain is decreased by a factor 2 in every iteration. Hence, the algorithm is $\mathcal{O}(\log_2 (UB - LB))$.

5 The beautify

The algorithm used to reduce the sum of the waiting times in a given schedule is called the beautify. It is a heuristic for the minimization problem (3) and solves the maximization problem (4) for the trips in the schedule. It is possible to specify a filter to narrow down the number of trips for which problem (4) is solved. In the transport situation, problem (4) is solved for every existing trip. For some clients, however, it is desirable to maximize only the start instant of trips that exceed work time, i.e. trips with a duration that exceeds the total work time allowed in a single trip. This filter will not be discussed in this thesis.

5.1 Selecting a trip for optimization

The first step is to select a trip to solve maximization problem (4). There are three ways to determine which trips are candidates for selection.

1. A list can be specified with trips that should be optimized.
2. It can be specified that all trips should be optimized.
3. If nothing is specified, then only the trips in which something has changed since the last optimization are a candidate.

When nothing is specified, a *delta keeper* is used to determine which trips are changed since the last optimization. This keeper contains all trips that are changed or depend on another trip that has changed. Examples of events that cause a change of a trip are planning an extra order in the trip or changing the start instant of the trip. Then this trip, and every other trip that has changed due to this modification, is placed in the delta keeper. All trips in the delta keeper are candidate for beautify, except the ones that have been beautified already. The sequences of all candidate trips are then sorted according to their place in the layers, starting with the sequence in the lowest numbered layer (see section 2.1). The trip belonging to the first sequence in this sorted list is optimized next. Thereon, a new trip is selected from all trips that are then in the delta keeper. These steps are repeated until there are no candidates left for beautify.

5.2 Optimizing a single trip

Let trip n be the selected trip for optimization. The optimization of the trip consists of two parts.

- The start instant of the trip, i.e. the start instant of the first external action of the trip, will be optimized using binary search to find the maximum start instant of the trip without a delay of the finish instant of the trip, i.e. the finish instant of the last external action of the trip, and without any additional violation of a relevant, i.e. time-concerning, restriction.
- If the trip contains any couples, then the acquire tree for every couple is also optimized. The start instant of every leaf of the acquire tree is delayed as much as possible using binary search. Optimizing the acquires is called *beautify ways to acquire*. In most cases, a driver is acquired in the leaf of an acquire tree, and beautify ways to acquire finds the

latest possible start instant of the driver to reach the couple location in time together with possible other resources that acquire the driver.

These two parts will be discussed in the remainder of this section.

5.2.1 Optimizing the start instant of the trip

The method binary search is used to find the optimum of the maximization problem (4). Every time a start instant for the trip is chosen in the search domain, all actions in the trip are recalculated. When the last external action is delayed, all actions in other trips which depend on the last action of the beautified trip will update their instants. But a delay of the finish of the trip is not allowed, hence these calculations are unnecessary, since the current start instant will not be accepted. To prevent all these calculations, an override of the finish instant is added to the last external action. The observers only see the override finish instant, hence a delay of the finish instant of the last external action during the optimization will not cause every dependent action to update its instant. Remember, however, that this does not mean that there are no actions outside the optimized trip that need to be recalculated.
If the first obtain action in the trip contains a predefined value for its start instant, or the couple contains predefined start information for its leaves of the acquire tree, then these values are removed from the trip and the start instants of all actions in the trip are recalculated. If any action in another trip has an observer that is triggered, then this action is also recalculated.

The search domain
The left boundary T_l of the search domain is defined to be the start instant of the first external action of the trip. This implies that if an obtain or acquire is the first child of the trip, then its start instant is not used as start instant of the trip. If there is no start information available, then the left boundary equals the foundation date of ORTEC bv, T_l = 1981-04-01 at midnight. The right boundary T_r is the finish instant of the last external action of the trip. The best known start at this moment, denoted by T_b, is the left boundary.
Note that the size of the search domain is finite. It is decreased by at least one second (= one unit of time) in every step, hence the size of the search domain in seconds at the start of the binary search is an upper bound for the number of steps needed in the algorithm to find the optimal solution.

Choose the next start instant to try
If the first obtain action in the trip contained a predefined value for its start, and this value is larger than the lower bound of the search domain, then this value is the first value to try as a better start instant of the trip.
Thereafter it is checked if the current lower bound is already the best possible start instant of the trip, since detecting this in an early stage increases the performance of the beautify. Therefore, the next start instant is a delay of the lower bound by one second, since this is the smallest unit of time in the beautify. If this new start instant T_n is rejected, then T_b is the latest possible time to start this trip without delaying the finish of this trip or causing additional violations of the restrictions and the binary search is done. Otherwise, the binary search continues.
If there is no original start time found, then the search domain is large (T_l = 1981-04-01 at midnight), and it is likely that the best start time will be close to the right boundary. Therefore,

the next start instant T_n is chosen at 95% of the search domain, i.e. $T_n = \lfloor 0.95 * (T_r - T_l) \rfloor$. If this T_n is accepted, then $T_l = T_n$ and the process is repeated.
After the first rejection of a start instant, every next start instant is chosen as the middle of the search domain, i.e. $T_n = \lfloor 0.5 * (T_r - T_l) \rfloor$, until the size of the search domain equals one step size (= one second).
When the difference between T_l and T_r is one step size, and the last tried start instant is T_r, then T_r was rejected, hence $T_b = T_l$ is the solution to the maximization problem (4). If the last tried instant is not T_r, then T_r is the only possible start instant that is better than T_l. Therefore, the last start instant to try is $T_n = T_r$.

5.2.2 Optimizing the leaves in the acquire trees

A couple is beautified to determine the best feasible start time for all acquire actions that are leaves of the acquire tree under this couple, see section 2.1 for acquires. Every leaf of the tree should start as late as possible to minimize the waiting times under the couple.
The optimization of the start instant of a leaf of the acquire tree is done in a very similar way as for the start instant of the trip, see section 5.2.1. In this case, the left boundary of the search domain equals the start instant of the acquire leaf. The right boundary equals the finish instant of the couple. The first start instant to try is the original start instant of the acquire leaf. In the remainder, the start instants are chosen in exactly the same way as for the start instants of the trip.
When every leaf of the acquire tree is optimized, the optimization of the couple is completed. After the optimization of every couple in the trip, the selected trip is completely optimized and the next trip is selected to be optimized.

6 Analysis of the beautify

The beautify is a heuristic for solving the optimization problem (3). In this section, some important decisions that have been made while designing the beautify will be analyzed.

6.1 Heuristic

The beautify solves the optimization problem (3) by reducing it to N maximization problems given by (4), where N equals the number of trips in the schedule. If there are dependency relations between the trips, then optimizing the start instant of a trip can increase the amount of waiting time in other trips, and it can even increase the duration of the total schedule. Therefore, minimization problem (3) and the maximization problems (4) are not equivalent. However, for schedules with unidirectional dependencies and even for many schedules with more complex dependency relations, solving the maximization problems is a good heuristic for solving the minimization problem.

Although the minimization problem is rewritten into N maximization problems, not every maximization problem is necessarily solved during the beautify. When a trip is created or something inside the trip is changed, the trip is placed in the delta keeper and will be optimized when the beautify is applied to the schedule. Hence if a trip is not in the delta keeper at any moment during the beautify, then re-optimizing this trip will not reduce the amount

of waiting time in the schedule. Therefore, only trips that are in the delta keeper can be selected for optimization during the beautify.

6.2 Selecting a trip for optimization

The order in which the trips are optimized can affect the solution of the maximization problem as explained in section 4.3. As shown in Example 4.1, the dependency relations between actions in different trips should be taken into account when a trip is chosen for optimization. If the dependency arrows between two trips n_1 and n_2 all point to trip n_2, then n_1 should be optimized before n_2 is optimized. This will enable the beautify to remove or reduce any wait actions in trip n_2 that are created during the optimization of trip n_1, and no waits will be generated in trip n_1 during the optimization of trip n_2. Hence the best order to optimize the trips with such simple dependency relations can be deduced by following the dependency relations between the trips.

When the dependencies between the trips are not unidirectional, then it is not possible to determine the optimal order to optimize the trips only using the dependency relations. This is shown in Example 6.1.

Example 6.1. Consider the two trips in Figure 13.

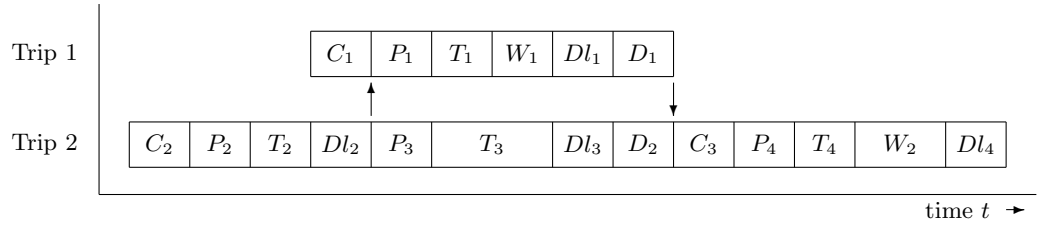

Figure 13: Two trips with two dependency relations.

Deliver action Dl_2 must be completed before pickup action P_1 can start, and some of the resources coupled in couple action C_3 are also used in trip 1. Suppose that $|W_1| < |W_2|$, where $|W_i|$ denotes the duration of wait action W_i. Both orders to optimize the trips in this schedule are examined below;

$1 \rightarrow 2$ **:** Optimizing trip 1 will remove wait action W_1, and the actions in trip 2 are not affected by any changes in trip 1. When trip 2 is optimized afterwards, wait action W_2 will be removed, and a wait action with duration $|W_2| - |W_1|$ is created before P_1.

$2 \rightarrow 1$ **:** Optimizing trip 2 will remove the wait actions W_2 and W_1 and will create a new wait action with duration $|W_2|$ between C_1 and P_1 in trip 1. When trip 1 is optimized afterwards, the wait action in this trip will be removed and no wait actions are created in trip 2.

Clearly, the optimal order to optimize the trips in Figure 13 is $2 \rightarrow 1$.

Now consider the two trips in Figure 14.
The dependency relations between the trips are very similar to the relations in the schedule in Figure 13. Note that P_1 and P_4 cannot start before Dl_3 resp. Dl_2 are completed. Therefore,

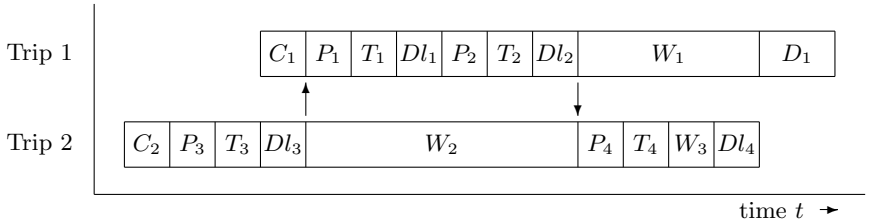

Figure 14: Another schedule with two trips and two dependency relations.

it is not possible to reduce the duration of wait action W_2. Again, both orders to optimize the trips are examined below;

$1 \rightarrow 2$ **:** Optimizing trip 1 will remove W_1 and extend wait action W_2 in trip 2 by $|W_1|$. This will delay the actions succeeding W_2 and remove wait action W_3 since its duration is smaller that the duration of W_1. Optimizing trip 2 afterwards will remove the additional wait in W_2 from the schedule and will not affect the actions in trip 1.

$2 \rightarrow 1$ **:** Optimizing trip 2 will remove wait action W_3, and create a wait action with duration $|W_3|$ after couple action C_1 in trip 1. The wait action W_1 is reduced by the same amount of time. Wait action W_2 cannot be reduced due to the dependency relations. When trip 1 is optimized afterwards, the wait actions in trip 1 are removed and wait action W_2 is extended by the new duration of W_1.

Therefore, the optimal order to optimize the trips in Figure 14 is $1 \rightarrow 2$, while the optimal order for the schedule in Figure 13 with similar dependency relations is $2 \rightarrow 1$.

When the dependency relations between the trips are not unidirectional, optimizing a trip can result in additional waiting time in an already optimized trip. This waiting time will not always be removed during the optimization of the other trips in the schedule. An algorithm that determines the optimal order to optimize the trips in the schedule should take the amount of waiting time into account that is created in (already optimized) trips. The duration of the wait actions that are created during the optimization of a trip does not only depend on the amount of waiting time in that trip, but does also depend on the calendars of tasks and the opening times of addresses used in these trips. This means that if an action is delayed by half an hour and its address is closed for lunch at the new start instant, this action will be delayed even more until the next open period of this address. Even the driver statuses of the actions in these trips should be taken into account, since changing the amount of waiting time in the trip can alter the need for a pause or rest in a travel. Handling all this information will make the algorithm very complex and many calculations will be needed to determine an optimal order to optimize the trips in the schedule.

The algorithm in the beautify that selects the next trip for optimization only uses the layers of the schedule, which are closely related to dependency relations as explained in section 2.1.2. The algorithm sorts all candidate trips by the place of their first sequence in the layers, and the first trip in this list is selected for optimization. Some properties of this algorithm are listed below.

- Execution of the algorithm is fast. Ordering the trips takes minimal time, since the sequences are sorted using the already created layers.

- In theory, the algorithm will not always find the best possible order to optimize the trips.
  - The algorithm only uses the layers, which are created using the dependency relations. The trips in Figure 14 will be optimized in the beautify in the order $2 \rightarrow 1$ since the first sequence of trip 2 will always be in a lower numbered layer than the first sequence of trip 1, while the optimal order is $1 \rightarrow 2$ as shown in Example 6.1.
  - Even in the case of unidirectional dependencies does this algorithm not guarantee to find an optimal order to optimize the trips. Consider a schedule consisting of two trips n_1 and n_2. Suppose trip n_1 consists of $s \in \mathbb{N}_{>2}$ sequences with the first sequence placed in layer 1, while every succeeding sequence is placed in the lowest possible layer. Furthermore, suppose that trip n_2 consist of just one sequence and that the last sequence of trip n_1 depends on the sequence of trip n_2. A correct layering is shown in Figure 15.

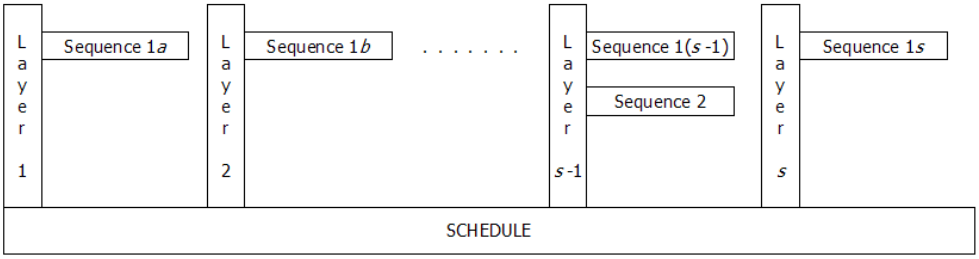

Figure 15: A correct layering for a schedule with two trips, with sequence 2 placed in a lower numbered layer than sequence $1s$.

The order in which the trips will be optimized is $1 \rightarrow 2$, but as shown before the optimal order for such a schedule is $2 \rightarrow 1$.

- In practice, this algorithm provides a good order to optimize the trips. This can be deduced from the fact that applying the beautify a second time on a representative schedule consisting of more than 250 trips does not alter the schedule. Hence there are no wait actions created in already optimized trips that would not have been there if a different optimization order was chosen.

Since trips can be placed in the delta keeper during the beautify, the selection algorithm is applied every time a trip has to be chosen for optimization. It is also possible to determine the collection of all the trips that can become available as a candidate for optimization at the start of the beautify, since it only contains those trips that are accessible by a path in the (directed) dependency graph starting at the first sequence of a trip that is in the delta keeper at the start of the beautify. Only these trips can be placed in the delta keeper during the optimization. Applying the selection algorithm on this collection at the start of the beautify will give the order to optimize all the trips in the beautify. However, since the execution of the selection algorithm in the beautify is fast and provides a good ordering, determining all candidate trips at the start of the beautify will not improve the resulting schedule or significantly decrease the amount of time needed to apply the beautify to the schedule. Therefore, the selection algorithm that is used in the beautify will not be changed.

To examine the effect of the selection algorithm on the solution, the selection criteria are changed such that not the first but the last sequence in the layers is selected. The resulting solution for a representative schedule with more than 250 trips does not significantly differ from the solution that is obtained using the first sequence in the layers.
As mentioned before, applying the beautify twice to this representative schedule does not alter the solution to the minimization problem (3). Besides showing that the selection algorithm in the beautify is accurate, it also justifies the decision to allow the optimization of a trip only once during the beautify, since the benefits are so small that they do not justify the increase in calculation time.

Since the selection algorithm used in the beautify works well in many cases, is very fast, and the optimization order is not found to have a huge effect on the quality of the solution, this part of the minimization problem will not be addressed any further in this thesis.

6.3 Optimizing a single trip

This section will analyze some of the decisions made in the beautify to optimize a single trip.

6.3.1 The assumption on the restrictions

The best start instant for a trip is obtained using binary search. Binary search only guarantees to find the best possible start instant for a single trip, if Assumption 4.1 (Behavior of restrictions) and 4.2 (Start instants of actions) are valid. However, Assumption 4.1 is not always valid.

There are constraints that are specified using *rules*. An example of such a constraint is a limit on the number of resources that is allowed at an address. In a corresponding rule, the planner specifies the maximum number of resources allowed. A time window can be used to specify when the rule is valid. If a trip starts at time t, then every rule with t in its time window is valid. If the restriction for the maximum number of resources is tested on an action in this trip, then all corresponding valid rules are tested. If more than one corresponding rule is valid, then the tightest maximum is used for every kind of resource.
For restrictions using rules, Assumption 4.1 is often incorrect. This is demonstrated in Example 6.2.

Example 6.2. Consider a trip with decouple D at address a_d. Suppose there are two rules, rule 1 and rule 2, that contain a list L_1 resp. L_2 with allowed addresses for the decouple action. Let

- rule 1 be valid from 1 January 2009 until 31 December 2009, and let $a_d \in L_1$,
- rule 2 be valid from 1 June 2009 until 31 August 2009, and let $a_d \notin L_2$.

This situation might occur for example if an address is not available during the summer. Suppose restriction r tests if the address of the decouple action is allowed. The graph of f_D^r is given in Figure 16, where T_i denotes the start of month i of the year 2009. Note that from 1 June 2009 until 31 August 2009 both rules are valid. Since the strongest rule is leading, in this case rule 2, the decouple action is not allowed at address a_d and the restriction is violated.

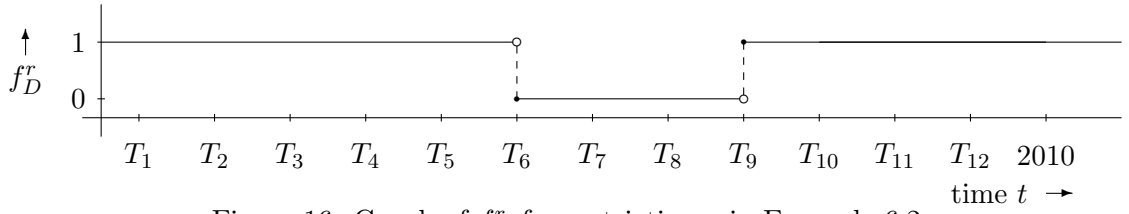

Figure 16: Graph of f_D^r for restriction r in Example 6.2.

If, during a binary search, the decouple action D starts in this summer period for start instant S of the trip, then the restriction is additionally violated. Every start instant of the trip S' with $S' \geq S$ is eliminated from the search domain, while starting after the summer period could be a feasible solution. Therefore, restrictions can lead to the elimination of an optimal solution from the search domain during binary search.

For most time-concerning restrictions, especially rule restrictions, it is possible to validate Assumption 4.1 by postponing the start of an action until an instant for which the restrictions are not additionally violated (if it exists). If the deliver action in Example 6.2 would originally start in the summer during a binary search in the beautify, then it would now be delayed until (at least) 1 September 2009. Hence a restriction will only be violated for an action if there is no later start instant without violations, which validates Assumption 4.1. This principle will be further explored in chapter 8.

6.3.2 The assumption on the start instant of an action

There also exist situations in which Assumption 4.2 (Start instants of actions) is not valid. Such a situation is described in Example 6.3.

Example 6.3. Consider trip 1 shown in Figure 17. Only the interesting part of the trip is shown; there are other actions before P_m and there might be actions after the decouple D.

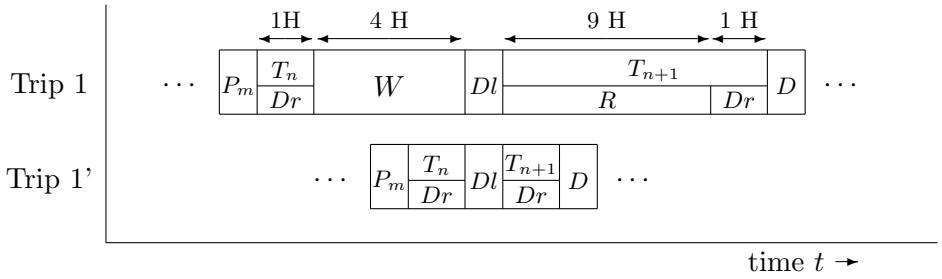

Figure 17: Visualization of the trip in Example 6.3.

Suppose the driver status at the start of travel T_n, W and Dl are shown in Table 3.

DS	T_n	W	Dl
DTP (h)	4.5	3.5	3.5
DTR (h)	5	3.5	3.5
WTR (h)	3	2	-2

Table 3: The driver statuses at the start of some actions in Figure 17.

During the wait action, the value of WTR drops below zero, hence at the start of the next travel, T_{n+1}, a rest is planned. After this rest, the driver has to drive for 1 hour. Then the decouple action can start. Suppose that, due to a delay of the start instant of the trip, the wait is removed from the trip and the driver status at the start of travel T_n is not changed, see trip 1' in Figure 17. Then the driver does not need to rest during travel T_{n+1}, since the values of all three parameters at the start of travel T_{n+1} are larger than 1 hour. Hence the decouple action can start 9 hours earlier, while the start instant of the trip is delayed. This clearly contradicts Assumption 4.2.
Suppose that trip 1 is obtained in step j of the binary search and has start instant S_1^j, and suppose that the decouple starts 4 hours too late. Then S_n^j would be rejected, and every start instant larger than S_n^j would be excluded from the search domain. Let S_n' denote the start instant of trip 1'. Then $S_n' > S_n^j$, but the decouple action D would not start too late and the start instant would be accepted. Hence, the optimal start instant of trip 1 is not found by the binary search, because Assumption 4.2 is not valid.

Remark In Example 6.3, the rest action at the start of travel T_{n+1} in trip 1 is required because the maximum period of working without a rest is exceeded in the deliver action. In most situations in transport, however, the driver needs to rest because otherwise he exceeds the maximum driving time without a rest.

It is also possible that Assumption 4.2 is not valid for an action in a different trip than the trip that is optimized. This is explained in Example 6.4.

Example 6.4. Consider a schedule consisting of two trips, as shown in Figure 18. Pickup action P_3 cannot start before deliver action Dl_1 is completed.

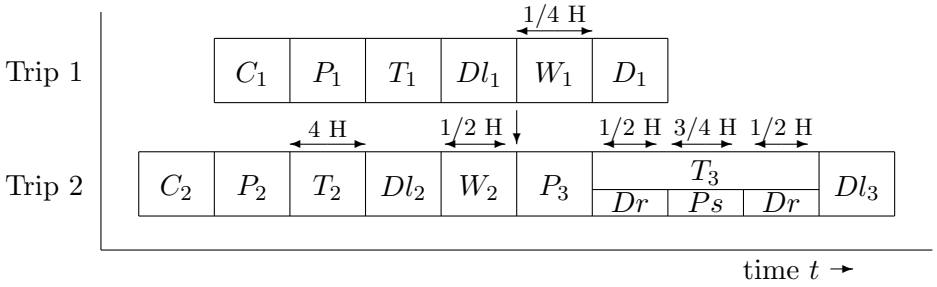

Figure 18: A schedule consisting of two dependent trips.

If trip 1 is optimized, then wait action W_1 is removed and every action in trip 1 except D_1 is delayed by 1/4 hour. Since Dl_1 is delayed, also P_3 is delayed by 1/4 hour and W_2 is extended by 1/4 hour. The duration of W_2 now equals the duration of a pause, hence the value of DTP in the driver status at the start of T_3 is maximal again, i.e. 4.5 hours. The pause in the travel will be removed and deliver action Dl_3 can start 1/2 hour earlier because the start instant of a different trip is delayed. If such a situation occurs during beautify, then optimizing trip 1 using binary search might not return the optimal start instant of trip 1.

If Assumption 4.2 is not regarded to be valid, it is nearly impossible to predict the behavior of the actions in the schedule when the start instant of an action is changed. If a start instant of

trip n results in an infeasible schedule, i.e. either there are additional violations or the finish of trip n is delayed, then it is possible that a later start instant results in a feasible schedule again. It is even possible that changing the start instant of a different trip will make the schedule feasible again.
In practice, it is not likely that an infeasible schedule will become feasible again for a later start instant of a trip. Therefore, this assumption is regarded to be valid.

6.3.3 Bounding the finish instant of a trip

The amount of waiting time in a trip is reduced by delaying the start instant of the trip without delaying its finish instant. There are three reasons to place an upper bound on the finish instant of the trip.

1. The fixation on the finish instant of the trip guarantees that with every accepted step in the binary search the duration of that trip is decreased.

2. The fixation on the finish instant of the trip reduces the number of calculations for a new start instant. It only prevents triggering observers in the networks of actions in other trips that observe the finish of trip n. However, if the driver in trip n is decoupled and then coupled in some other trip, then every action in trip m succeeding this couple action has a driver status that can change if some wait is removed in trip n. If such a driver status changes, then the action is updated and the driver status of the next action will also change, etc.
   In many schedules these dependency relations do not exist between the trips, hence bounding the finish instant of the trip will reduce the number of calculations for a new start instant in the binary search.

3. Without the fixation on the finish instant of the trip there is no guarantee that the process of delaying the start instant of trip n will end in a reasonable time. To prevent moving the start of the trip to the right even if it does not contain any wait actions, the search domain in the binary search for the best possible start instant of the trip is bounded on the right by the original finish instant of the trip. This also guarantees that the start instant of the trip is delayed by a finite amount of time.

Although bounding the finish instant of the trip guarantees that the duration of the optimized trip decreases for every accepted start instant in the binary search, it does not guarantee that the total schedule duration is decreased since the amount of waiting time in other trips can be increased.

Bounding the finish instant of the trip induces an upper bound on the search domain for the start instant of the trip, while the start instant of the trip in an optimal solution of the minimization problem (3) could be larger than its current finish instant. This solution cannot be found by the beautify. Hence, allowing the finish instant of the trip to be delayed can result in a schedule with less waiting time, but it will increase the number of calculations and therefore the total time needed to apply the beautify to the schedule. In the remainder of this thesis, the finish instant of the trip that is optimized is not allowed to be delayed.

6.4 Testing the restrictions

A large part of the time needed to apply the beautify to a schedule is used to update the actions in the schedule when a new start instant of a trip or acquire is tried. Great efforts have been made to reduce the time needed to update the actions, but the way the restrictions are tested during the beautify has not received much attention. Based on the wrong assumption that approximately 30% of the time needed to apply the beautify to a schedule is used to test the restrictions, the way restrictions are tested will be examined more closely in this section. However, further analysis shows that only 3% of the time to apply the beautify to a schedule is used to test the restrictions.

6.4.1 Warnings and tolerated values

As explained at the beginning of chapter 3, a warning message is generated if during a planning command the violation of a restriction is larger than the tolerated violation. The planner has to decide if (s)he accepts or rejects this violation. A rejection will cause all changes made for this planning command to be reversed. If the violation is accepted, the execution of the planning command will be continued and the (new) tolerated value is stored within the action. This is shown in Figure 19.

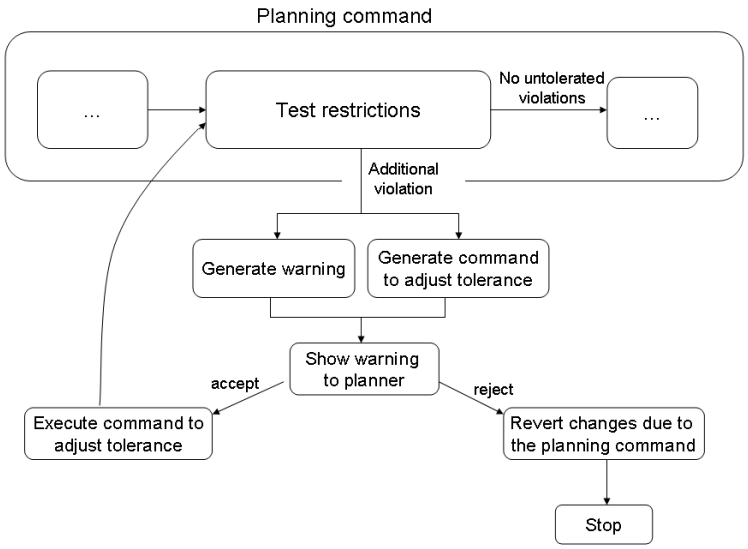

Figure 19: Execution diagram of a command for which the restrictions are tested.

The restrictions are not tested using a simple function call, but a command is used that encapsulates the calls to the functions that are used. Within this command, there is a single function that checks if a restriction is violated on an action, compares the violation with the tolerated value, and, if necessary, generates commands to warn the planner and to adjust the tolerance. Hence when a new start instant of a trip or acquire results in an additional violation of an action, the commands to warn a planner and to adjust the tolerance are automatically generated when the restrictions are tested. But the planner should not be notified of this violation and the tolerated value is not allowed to be adjusted during the execution of the beautify. Therefore, if a restriction is additionally violated in the beautify, the generated

warning is intercepted and destroyed before it is shown to the planner, and the command that is generated to tolerate the current violation will not be executed.

When restrictions are tested in the beautify, the only steps should be to obtain the current violation, compare it with the tolerated violation, and continue testing the next restriction if there is no additional violation or returning the information that there is an additional violation for some restriction. Generating warnings and a command to adjust the tolerated value is not necessary when a new start instant is tried in the binary search. The result of this improvement will be discussed in section 7.1.

6.4.2 Ordering the restrictions

When a restriction is additionally violated in the beautify, the tried start instant is rejected without testing the remaining restrictions. Therefore, the time needed to test the restrictions depends on the order in which the restrictions are tested. There are two logical ways to order the restrictions;

- Sort the restrictions starting with the least time consuming restriction, i.e. the restriction that needs the least time to be tested. Then the most time-consuming restrictions are not tested at all if a less time-consuming restriction is additionally violated.
- Sort the restrictions on the probability of an additional violation. This will reduce the average number of restrictions that is tested.

In the beautify, however, no order is explicitly determined in which the restrictions should be tested. As a result, the restrictions are tested in alphabetical order using their names. The results of testing the restrictions in a different order will be discussed in section 7.2.

Part III
Improvements

7 Improving the command to test the restrictions

The command that is used to test the restrictions during a binary search in the beautify is more complicated than necessary. The knowledge that an additional violation is never tolerated during the binary search can be used to reduce the time that is needed to test the restrictions in the beautify.

7.1 Warnings and tolerated values

If a step in the binary search is rejected because a restriction is additionally violated, then a command has been generated to warn a planner and to adjust the tolerated value of that restriction, while these commands will never be executed. To decrease the time needed to test the restrictions, a new function `CheckIsViolated` is implemented for every time-concerning restriction. This function accepts an action A as input variable and returns a boolean[1] to indicate if there is an additional violation of that restriction for action A. There is no command generated to warn the planner or to adjust the tolerance. See Appendix A for some details of the implementation of the originally used function and of the function `CheckIsViolated`.

To examine the effect of testing restrictions using the function `CheckIsViolated`, the restrictions are tested on a schedule with more than 1000 actions using the original implementation of the restrictions and using the function `CheckIsViolated`. The results are shown in Figure 20, where the bars indicate the standard deviation. The restrictions with the largest reduction of the time needed to test them are `Task time window` (58%), `Task planned according to description` (37%) and `Driving duration` (32%). For restrictions without a changed violation no reduction of the time needed to test them is established, since no command was originally generated to alter the tolerated value or to warn the planner.
The reduction of the time that is needed to test the restriction `Task planned according to description` is not entirely established by not generating commands in case of a changed violation. This restrictions checks five properties of a task, including the allowed duration of the task and the address where the task should be executed. Every property has its own tolerated value, and in the original implementation the five properties are always checked, even if a previously checked property has an additional violation. Since a start instant in the binary search will already be rejected if one of these properties is additionally violated, the function `CheckIsViolated` only continues to test the properties if the previously tested property was not additionally violated.

The total time needed to test the restrictions in the original implementation is $3.0 \pm 0.1\%$ of the total time needed to apply the beautify to a schedule. With the function `CheckIsViolated` this is reduced to $2.6 \pm 0.1\%$. Therefore, the total time needed to apply the beautify to a schedule is reduced by approximately 0.4%.

[1] A variable with value `true` or `false`.

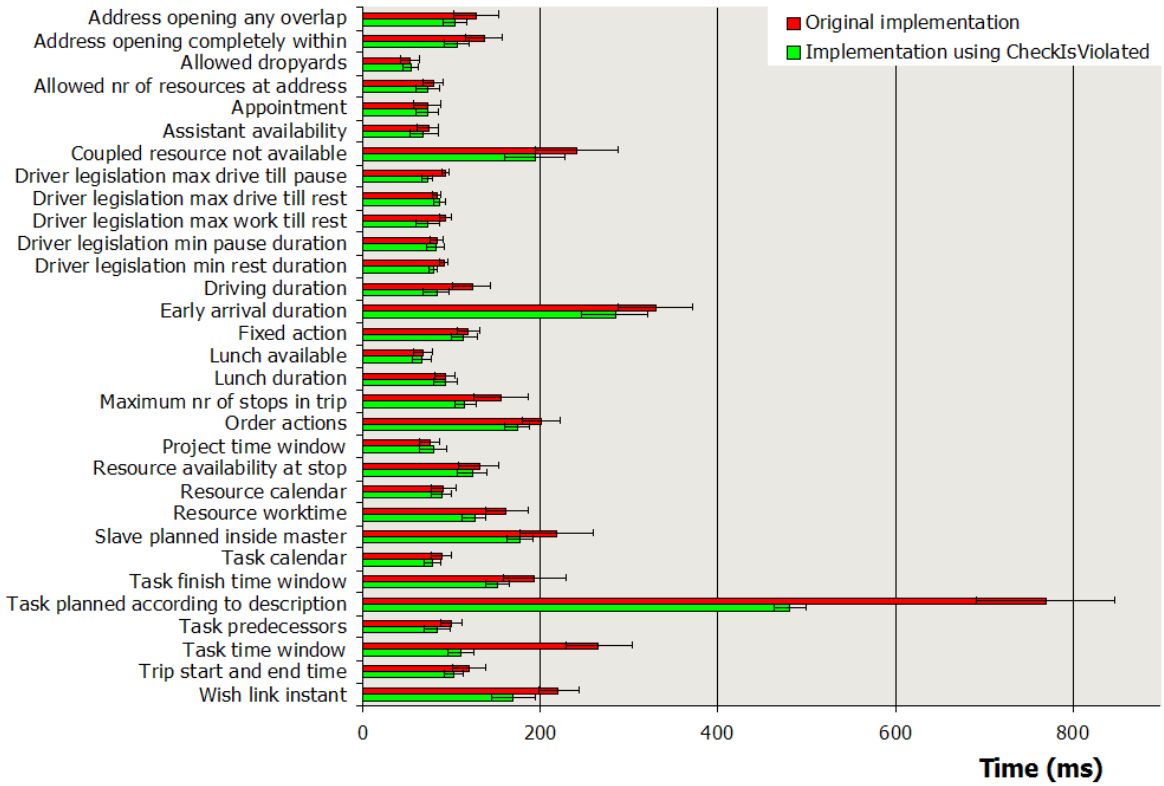

Figure 20: The time needed to test a restriction 10.000 times with the original implementation and with the new function `CheckIsViolated`.

The second approach to reduce the time needed to test the restrictions is to reduce the average number of restrictions that is tested for a step in the binary search. This will be discussed in the next section.

7.2 The order to test the restrictions

For a planning command, every restriction needs to be tested and every additional violation must be tolerated to continue the command. As a result, the order to test the restrictions in a planning command is not important. As mentioned in section 6.4.2, this is not true for the beautify. This section will examine two different orderings of the restrictions.

7.2.1 Sorting on the time needed to test a restriction

If a restriction is additionally violated during a step in the binary search, then the remaining restrictions will not be tested. If every restriction has the same probability of an additional violation, then the average time that is needed to test the restrictions is minimal if they are ordered on increasing time needed to test them.
The timings of the restrictions with the function `CheckIsViolated` are used to divide the restrictions into three categories to indicate how time-consuming they are. The results are shown in Table 4. The restrictions in the class 'Fast' are tested before the restrictions in

the class 'Normal'. The restrictions in the class 'Slow' are only tested if no restriction in the other two classes is additionally violated. Within each class, the restrictions are ordered alphabetically.

FAST	NORMAL	SLOW

- Allowed dropyards,
- Allowed nr of resources at address,
- Appointment,
- Assistant availability,
- Driver legislation
  * min rest duration,
  * min pause duration,
  * max drive till pause,
  * max drive till rest,
  * max work till rest,
- Driving duration,
- Lunch available,
- Project time window,
- Resource calendar,
- Task calendar,
- Task predecessors.
	- Address opening any overlap,
	- Address opening completely within,
	- Fixed action,
	- Lunch duration,
	- Maximum nr of stops, in trip,
	- Order actions,
	- Resource availability, at stop,
	- Resource work time,
	- Slave planned inside, master,
	- Task finish time window,
	- Task time window,
	- Trip start and end time,
	- Wish link instant.
		- Coupled resource not available,
		- Early arrival duration,
		- Task planned according to description.

Table 4: The restrictions divided into three categories based on the time that is needed to test them.

The total time needed to test the restrictions with this ordering is not significantly reduced. This can be explained by the fact that many restrictions in the class 'Fast' have names starting with A-D. Therefore, they were in the original implementation also tested before most of the other restrictions.

7.2.2 Sorting on the probability of an additional violation

The second ordering that is implemented for the restrictions is based on the probability of an additional violation. The time needed to test the restrictions in case of an additional violation is minimal if only the restriction that is additionally violated is tested. Therefore, the restrictions are ordered starting with the restriction that has the highest chance to be additionally violated.

The chance that a restriction is additionally violated depends strongly on the schedule that is used. It is not possible to assign a probability of an additional violation to a restriction that is accurate for every schedule. Instead, this probability is determined using the results of previous tests for that restriction. Let $N_{R,T}$ denote the number of times restriction R is tested, and let $N_{R,AV}$ denote the number of times this restriction is additionally violated. After every test of a restriction, these numbers are updated, and the ratio $N_{R,AV}/N_{R,T}$ equals the probability that restriction R is additionally violated.

The restrictions are tested many times when the beautify is applied to a schedule $\mathscr{S}$. Hence after the beautify is applied, the ratio $N_{R,AV}/N_{R,T}$ for every restriction R can be used to sort the restrictions. If the beautify is then applied to the original schedule $\mathscr{S}$ using the order based on the probability that a restriction is violated, the total time needed to test the restrictions is increased to approximately 7% of the time that is needed to test the beautify. The reason for this increase is the additional time needed to update the counters $N_{R,AV}$ and $N_{R,T}$ after every test of restriction R. The reduction of the time needed to test the restrictions due to the decrease in the average number of restrictions that is tested per step in the binary search is not enough to compensate this additional overhead.

The way the restrictions are tested during the binary search of the beautify is in theory not the most efficient method since unused commands are generated in case of an additional violation and the restrictions are almost randomly tested. In practice, however, testing the restrictions is so fast that it has little value to try to improve it even further.

8 Improving the mechanism for generating wait actions

The optimal solution $\hat{t}$ of maximization problem (4) can be eliminated from the search domain during the binary search, if for start instant $t < \hat{t}$ a restriction is additionally violated that is not additionally violated at start instant $\hat{t}$. This situation is possible, since Assumption 4.1 (Behavior of restrictions) is not always valid. Assumption 4.2 (Start instants of actions) is assumed to be valid in this section. This section presents a framework to constrain the set of possible start instants of an action in the time-calculations of the engine. This framework prevents the elimination of the optimal solution from the search domain during the binary search.

8.1 Constraining the set of start instants of an action

The general idea behind the framework is to eliminate those start instants of an action A that are responsible for the non-monotonicity of the function $f_A^r(t)$. Recall from formula (1) that

$$f_A^r(t) = \begin{cases} 1, & \text{if the violation of restriction } r \text{ on action } A \text{ is within the tolerance,} \\ 0, & \text{otherwise.} \end{cases}$$

Let I_r^{pos} denote the set of start instants that are not eliminated by restriction r from the domain of all start instants of action A, hence

$$I_r^{\text{pos}} = \{\ t \mid f_A^r(t) = 1\ \} \cup \{\ t \mid f_A^r(t) = 0 \ \wedge \nexists\ t' > t \text{ s.t. } f_A^r(t') = 1\ \}. \tag{5}$$

Although the function $f_A^r(t)$ need not be monotonically decreasing, the restricted function $f_A^r(t)|_{I_r^{pos}}$ is monotonically decreasing on its domain. Hence if restriction r is additionally violated for action A with start instant $t \in I_r^{pos}$, then it will be additionally violated by at least the same amount for every start instant $t' \in I_r^{pos}$ with $t' > t$.

Theorem 8.1. *If a start instant of action A is always chosen in the set I_r^{pos} for every time-concerning restriction r, and Assumption 4.2 is valid, then the optimal solution cannot be eliminated from the domain of the binary search.*

Proof. Suppose that the optimal solution of the binary search is start instant $\hat{t}$. Let start instant t of the trip be chosen during a step in the binary search such that $t < \hat{t}$ and t is rejected. As a result, $\hat{t}$ will be eliminated from the domain of the binary search. Instant t can be rejected for two reasons;

1. The finish instant of the optimized trip is delayed.
2. A restriction r_1 is additionally violated for an action A with start instant t_A.

Suppose that start instant t is rejected because of reason 1. By Assumption 4.2, if the finish instant of the trip is delayed for start instant t, then this finish will also be delayed for start instant $\hat{t} > t$. But $\hat{t}$ is a feasible solution, hence the finish of the trip cannot be delayed. $\lightning$
Suppose that start instant t is rejected because of reason 2. Let the start instant of action A for start instant $\hat{t}$ of the trip be given by $\hat{t}_A$. Since $\hat{t} > t$, it follows from Assumption 4.2 that $\hat{t}_A \geq t_A$. By construction, both $\hat{t}_A$ and t_A are in I_r^{pos} for every restriction r, and I_r^{pos} is such that $f_A^r(t)|_{I_r^{pos}}$ is monotonically decreasing on its domain. Since $f_A^{r_1}(t_A) = 0$ and $f_A^{r_1}(t) \in \{0, 1\}\ \forall t \in I_{r_1}^{pos}$, it follows that $f_A^{r_1}(\hat{t}_A) = 0$. Hence restriction r_1 is also additionally violated for start instant $\hat{t}_A$. But $\hat{t}$ is a feasible solution. $\lightning$
Hence no such start instant t exists in the domain of the binary search. Therefore, it is no longer possible to eliminate the optimal solution from the search domain of the binary search. □

The impact of this framework on the solution of the beautify is illustrated in the Example 8.1.

Example 8.1. Suppose that trip n is optimized. Let t be the start instant of trip n before optimization, and let $\hat{t}$ be the optimal solution of maximization problem (4) for trip n. Furthermore, let $t' < \hat{t}$ be in the search domain of the binary search such that action A starts at instant t_A and restriction R is additionally violated for action A. This is shown in Figure 21.

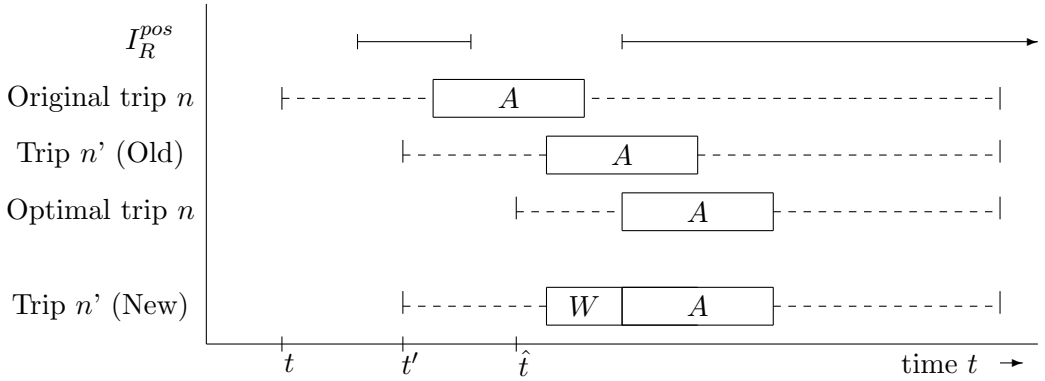

Figure 21: Three steps of a binary search during the optimization of trip n. Trip n' has start instant t' and is shown twice, first calculated with the original time-calculation of the engine and secondly calculated using the new framework with I_R^{pos} for generating wait actions.

Trip n' in Figure 21 is trip n with start instant t'. There is no wait action generated before action A and restriction R is additionally violated. Although this restriction is not violated

for the later start instant $\hat{t}$, this instant will not be found by the beautify, since it is eliminated after instant t' is tried. With the new framework, action A in trip n' will be delayed until the restriction is not additionally violated. Therefore, instant $\hat{t}$ is not eliminated from the search domain and will eventually be found as the optimal solution of the optimization of trip n.

The new framework is designed to improve the time-calculations that are necessary for the beautify, since it should prevent that the optimal solution is eliminated from the search domain of a binary search. However, if the framework is not used during time-calculations outside the beautify, then it is possible that wait actions that were generated during the time-calculations of the beautify are removed if the actions are updated outside the beautify. If this occurs for an action in the optimal solution, then it is possible that the optimal solution suddenly becomes infeasible, since restrictions can be additionally violated due to the earlier start instant of the action. The new framework should therefore be used for every time-calculation of the engine.

One of the problems with this framework is to find the set I_r^{pos}. In section 2.2 the mechanism to determine the finish instant of an action is briefly described. In this mechanism there are also some start instants eliminated from the search domain. In the next section this mechanism is considered again.

8.2 The original mechanism for generating wait actions

In the original implementation, a wait action is generated if the finish instant of an action and the start instant of its succeeding action in the same trip cannot coincide. This can occur if for example a package is transported first in trip n_1 and then in trip n_2. Then it is possible that the pickup in trip n_2 has to wait until the deliver action in trip n_1 is completed. A wait action is generated from the finish of the action preceding the pickup in n_2 until the finish of the deliver action in trip n_1. The dependency relations between the actions are used to ensure that they do not overlap. If only the dependency relations are used to determine the need for a wait action, and if a trip spans several days, then the restrictions concerning the opening times of an address, the calendar of a task or the calendar of a resource would be violated for many actions, since calendars are often closed during the night. Many warnings would be generated and the planner would have to decide for every warning if the additional violation should be tolerated. To avoid this situation, also calendars of addresses, tasks and resources are used to determine the need for and the duration of a wait action. This is illustrated in Example 8.2.

Example 8.2. Consider an action A. The address of A has its open periods specified in a calendar. Suppose that the task and the resources are always available. Furthermore, suppose that the duration of action A is 15 minutes. If the action that precedes action A finishes within a closed period of the calendar of the address, then a wait action is automatically generated until the start of the next open period of the calendar of the address of action A. Therefore, action A will not start within a closed period of the calendar of its address if there exists a later start instant for which the address is open. The same principle holds for the calendar of a task and the calendars of the resources.

The mechanism for generating wait actions illustrated in Example 8.2 eliminates certain start instants from the set of all start instants of action A. The engine uses its own mechanism to

determine which start instants are eliminated, while the restrictions have their own mechanism to determine if they are violated for the action. These two mechanisms are not equal and it can occur that start instants are eliminated for which no restriction is violated, or that an instant is *not* eliminated while there are restrictions violated. This is illustrated in Example 8.3.

Example 8.3. Let t denote the start instant of the action A in Example 8.2 and let r_1 and r_2 denote the restriction `Address opening any overlap` resp. the restriction `Address opening completely within`. If the address is every day open from 6 AM until 8 PM and there are no tolerated values, then the graph $f_A^{r_1}(t)$ is shown in Figure 22.

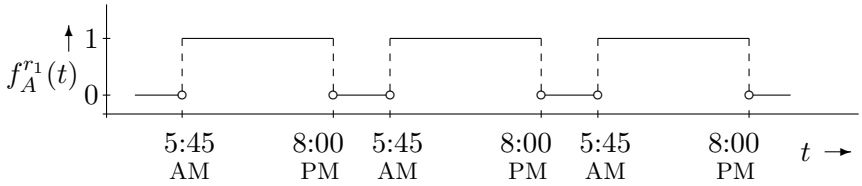

Figure 22: Graph of $f_A^{r_1}$ as a function of the start instant of action A.

If action A would start between 8 PM and 6 AM, then a wait action is automatically generated until the start of next open interval of the address. Consider the following two situations;

- If only the restriction r_1 is tested on action A, then the start instants in the interval [5:45, 5:59] AM would be eliminated from the set of possible start instants, since a wait action is generated until 6 AM to prevent action A from starting outside an open period of the calendar of the address. However, for every start instant in the interval [5:45, 5:59] AM there is overlap between the action and the next open interval of the calendar of the address. Therefore, these start instants are eliminated even though $f_A^{r_1}(t) = 1\ \forall t \in$ [5:45, 5:59] AM. Hence for this restriction too many instants are eliminated from the set of start instants.

- If only restriction r_2 is tested, then $f_A^{r_2}(t) = 0$ for every start instant that is eliminated, since every start instant outside an open period of the calendar of the address will result in a violation of r_2. However, there are too few instants eliminated, since $f_A^{r_2}(t) = 0$ for every instant in the interval [7:46, 8:00] PM while these start instants are not eliminated.

Hence the set of start instants of an action is constrained, but the obtained set of possible start instants is in both situations not equal to the right hand side of formula (5). Furthermore, the engine does not use any tolerated values in its calculations. Therefore, any start instant t that results in a tolerated violation is eliminated even though $f_A^r(t) = 1$.

Hence the original mechanism does constrain the set of start instants of an action, but the obtained set of possible start instants differs from the set $f_A^r(t)$ because both the restrictions and the engine have their own method to determine if the start of an action is allowed. A second shortcoming of this mechanism is that there are restrictions for which no construction exists in the engine to extend the duration of a wait action until the next allowed start instant for those restrictions. Examples of these restrictions are restrictions that use rules, as shown in Example 6.2 in section 6.3.1.

The solution for obtaining the correctly constrained set of all possible start instants of an action is found in the role of the restrictions. This will be the main topic of the next section.

8.3 The role of the restrictions

The engine is the mechanism that (amongst others) generates the wait actions and determines finish instants of actions in the schedule. The constraints on the schedule should be satisfied by the calculations of the engine. Since it is not always possible to satisfy every constraint of the schedule, the restrictions were created to warn the planner in case a constraint was not satisfied after the time-calculations of the engine. The restrictions are also responsible for adjusting the tolerance in case the planner decided to accept an additional violation. The restrictions are designed such that they operate independently of the engine, i.e. the engine performs the time-calculations and the restrictions check afterwards for additional violations. After the engine and the restrictions were implemented, the beautify was designed to reduce the amount of waiting time in the schedule. The beautify proposes new start instants for a trip and asks the engine to update the schedule for this new start instant. Afterwards it asks the restrictions to check for additional violations and decides to accept or reject the new start instant. Then the next start instant is proposed and the process is repeated until an optimal solution is found. Hence not only the engine, but also the beautify performs time-calculations, with the difference that the beautify only calculates the optimal start instant of an schedule, while the engine updates the schedule for every step in the binary search. The relations between the beautify, the engine and the restrictions are shown in Figure 23.

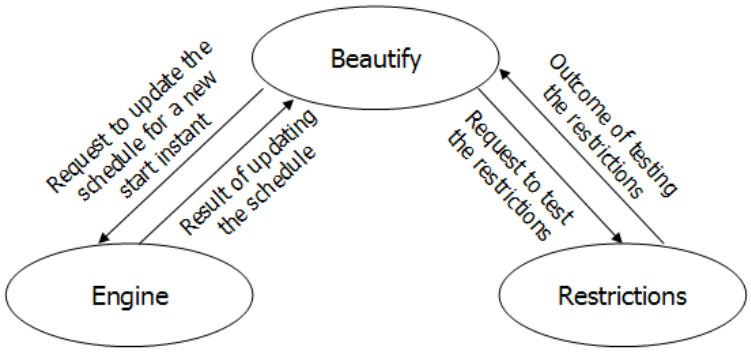

Figure 23: Diagram with the relations between the beautify, the engine and the restrictions in the original implementation.

As shown in Figure 23, there is no communication between the restrictions and the engine. However, using calendars of addresses, tasks and resources when wait actions are generated reveals that the engine and the restrictions cannot be treated as two independently operating objects. They are connected, since the engine uses these calendars to reduce the violations of the corresponding restrictions. The two shortcomings of this mechanism are discussed in section 8.2;

- The connection between the engine and the restrictions only exists for some restrictions.

- The method that is used to determine the violation of a restriction is more complicated than the method that is used to determine the duration of a wait action for this restriction.

To determine the duration of a wait action, the engine searches for the next open period of a calendar after the start instant of the wait action. The finish instant of the wait action will be (at least) the start instant of this open period. However, instead of using the calendar to determine the duration of the wait action, the engine should use the set I_r given by

$$I_r = \{\ t \mid f_A^r(t) = 1\ \},$$

where r is the restriction corresponding to the calendar. Hence the set I_r contains all the start instants of action A for which restriction r is not additionally violated. When this set is used, restriction r is only additionally violated if there is no later start instant without an additional violation. Using this set will prevent excluding the optimal solution from the search domain of the binary search.

Since the restrictions determine $f_A^r(t)$, they contain information that can be used to obtain the set of allowed start instants of action A. Therefore, a framework will be implemented that enables the engine to communicate with the time-concerning restrictions. When the engine generates a wait action before action A can start, it will be able to ask every time-concerning restriction to return the set I_r for that action. Hence the role of the restrictions will be changed from only checking the actions in a schedule after the engine has updated the schedule to both advising the engine during its time-calculations and checking the actions afterwards. The new relations between the beautify, the engine and the restrictions are shown in Figure 24.

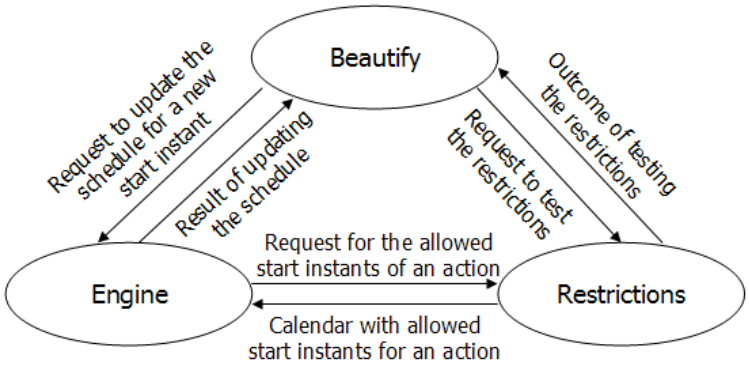

Figure 24: Diagram with the relations between the beautify, the engine and the restrictions in the new implementation.

Changing the role of the restrictions will reduce the number of violations in a schedule. If an action had a small additional violation in the original implementation, the planner was able to tolerate it. In the new implementation the engine automatically generates a wait action to delay the start of the action until there is no additional violation (if such a start instant exists). Therefore, the possibility of the planner to tolerate additional violations in the schedule is limited due to this new implementation. The framework is implemented in

such a way that it is possible to return to the original behavior for generating wait actions. Details of the implementation are given in the next section.

8.4 Details of the framework

When the engine asks the restrictions for advice on the allowed start instants of action A, every restriction returns a set I_r with allowed start instants. If action A would start at an instant that is included in every set I_r, then no restriction would be additionally violated. Therefore, the set I of allowed start instants of action A is given by

$$I = \bigcap_{r \in \mathcal{R}} I_r, \tag{6}$$

where $\mathcal{R}$ is the set of all time-concerning restrictions that are used in the calculations of the engine. To be able to intersect all these sets I_r, every I_r is considered to be a calendar. As explained in section 3.1, calendars consist of a weekly repeated pattern, possibly together with additional open periods and extra closed periods. They contain information on the opening times of an address, task or resource. By construction, it is not possible for the engine or the restrictions to create or modify such a calendar. Therefore, new types of calendars will be introduced that can be created or modified by the restrictions.

8.4.1 Calendars

For some restrictions, the set of allowed start intervals for an action consists of just one interval. Examples are restrictions that check if the start or finish instant of an action is contained in some time window. The calendars that are mentioned in section 3.1, i.e. calendars that can contain versions, extra open intervals and extra closed intervals, are more complex than necessary for those restrictions. The same is true for restrictions that return a set of allowed start intervals for an action. In this section, some new types of calendars will be introduced that can be created by the restrictions.

Before these new calendars are introduced, first consider the definition of a calendar again. In the original situation, only one calendar existed. Therefore, it was clear that a calendar was an object that could contain versions, extra open and extra closed intervals. Now that there are several types of calendars, the notion of a calendar is changed to an object that contains information on time-intervals. By design, a calendar cannot return a list with all its open intervals, but it returns the first interval that ends after a given instant t. Given this new notion of a calendar, several new types of calendars are implemented. Table 5 shows all these new calendars together with the already existing calendar `Weekly repeated`, and every calendar in this table except the calendar `Weekly repeated` can be used by the restrictions to create a calendar with allowed start instants for an action.

Restrictions will create a calendar `One interval` or a calendar `Set of open intervals` if the set of allowed start instants of an action consists of one interval resp. a set of intervals. A calendar `Adapter` will be created if the set of allowed start instants is obtained by modifying an already existing calendar. A restriction that uses the calendar `Adapter` is the restriction `Address opening any overlap`. If the duration of the action is nonzero and there are no

Calendar type	Description
`One interval`	Contains only one interval $[a, b]$ with $b \in [a, \infty]$.
`Set of open intervals`	Contains a set of open intervals.
`Weekly repeated`	Contains weekly repeated open intervals, together with extra open and extra closed intervals.
`Adapter`	Contains another calendar and a specification on how to adapt the intervals in this calendar.
`Days only`	Contains another calendar and extends every open interval in this calendar to whole days.
`Intersection of a set`	Contains a set of calendars and returns only intervals that are included in every calendar of this set.
`Intersection of a pair`	Contains two calendars and returns only intervals that are included in both calendars.

Table 5: Different types of calendars.

tolerated values, the calendar containing the allowed start instants contains all the open intervals of the calendar of the address shifted to the left by the duration of the action. An example of two such calendars is given in Figure 25.

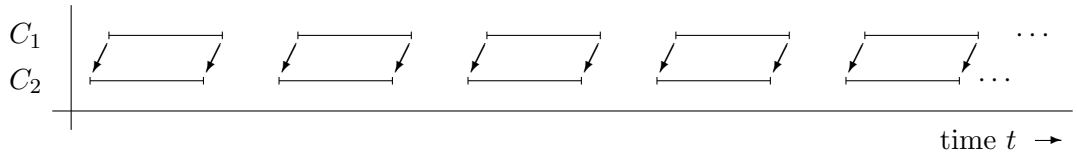

Figure 25: Graph of two calendars, where the bars indicate open periods. C_1 is a calendar `Weekly repeated` and C_2 is a calendar `Adapter` that shifts every open interval of C_1 slightly to the left.

The input for a calendar `Adapter` consists of a base calendar that will be adapted together with a description of the modification. Since for some restrictions it is possible to have different tolerances for the last interval of a calendar, it is possible to specify two modifications in the calendar `Adapter`. The first description is compulsory and every interval in the calendar will be adapted according to this description. The modification of the start and finish instant of an interval are allowed to differ. The second description is optional and will only be applied to the last interval of the calendar. When this description is given, it overrules the first described modification for this interval.

When the engine determines the finish instant of a wait action before action A, it obtains a set of calendars with the allowed start instants of the action A from the restrictions. Given the start instant t of the wait action, the engine has to determine the earliest allowed start instant of action A that is larger than t from these calendars. This start instant of the action will be the finish instant of the wait action. To obtain this earliest allowed start instant of action A, the engine should take the intersection of the set of calendars. In the next section, two different intersection algorithms are described.

8.4.2 Intersection algorithms

Every time-concerning restriction is now able to advise the engine on the moment at which an action is allowed to start. The engine should delay the start instant of an action A until no restriction is additionally violated for that action. This implies that the finish instant of the wait action before action A must be contained within the calendar of allowed start instants of action A for every restriction. This start instant can be found using the calendar `Intersection of a set`. Some details of the intersection algorithm that is used by this calendar can be found in Appendix B.

In some situations, the set I in formula (6) is empty after start instant t of the wait action. If the calendar `Intersection of a set` is used, then no allowed start instant t can be determined for action A and no wait action will be generated. This situation is not always desirable, as illustrated in Example 8.4.

Example 8.4. Suppose that for a deliver action the three restrictions `Address opening any overlap` (r_1), `Task time window` (r_2), and `Early arrival duration` (r_3) are used in the time-calculations of the engine. Furthermore, suppose that

(i) the calendar of the address of the deliver action is open every day from 8 AM until 6 PM,
(ii) the preceding travel action ends at 4:45 PM of day Y,
(iii) the task of the deliver action must start *and* finish between 5 PM and 6 PM of that same day Y,
(iv) the duration of the deliver action is 45 minutes,
(v) there should be at least 45 minutes between the arrival at the address and the start of the deliver action.

The calendars with allowed start intervals on the day of arrival are shown in Figure 26, where C_i corresponds to the calendar of restriction r_i for $i = 1, 2, 3$.

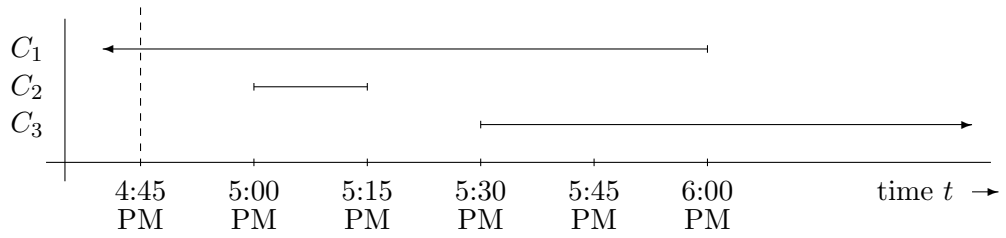

Figure 26: The calendars with allowed start instants for the three restrictions in Example 8.4. The dashed line indicates the finish instant of the action preceding the deliver action.

The finish instant of the wait action should be included in the intersection of the three calendars, but this intersection is empty. Therefore, no wait action is generated for the deliver action and both the restriction `Task time window` and the restriction `Early arrival duration` are additionally violated.
If this situation occurs during a normal planning command, then it is preferred to generate a wait action until the time window of the task opens, i.e. until 5:00 AM, and to let the planner decide if he tolerates the violation of the restriction `Early arrival duration`. This

restriction is often only used as a precaution, to minimize the effects of traffic jams on a schedule, since the extra wait action can be used to compensate (some of) the time that is lost due to a traffic jam.

In Example 8.4 it is suggested to ignore certain calendars if the intersection of the whole set of calendars is empty after a given time t. Then a wait action is generated to prevent an additional violation for the restrictions that are used in the intersection, while the planner has to decide if (s)he accepts the additional violations of the restrictions that correspond to the ignored calendars. Naturally, the question arises which calendars should be ignored in this intersection. There is no general answer to this question, since the calendars that should be ignored belong to those restrictions that are least important to a planner, which differs for every planner. Therefore, a planner will be enabled to choose which restrictions should be used by the engine in the time-calculations and to sort them on their importance. This results in the ordered set of $X \in \mathbb{N}_{\geq 0}$ calendars $\mathcal{C} = \{C_1, C_2, \ldots, C_X\}$, where C_1 is the calendar with the highest priority. The implementation is such that if $X = 0$, the behavior of the system equals the original behavior.

To obtain a start instant of an action from the ordered set $\mathcal{C}$, a new intersection operator $\cap_t^*(C_i, C_j)$ is defined, where a calendar will be ignored if the regular intersection would be empty. Let the function `IsEmpty` be given by

$$\texttt{IsEmpty}(C,t) = \begin{cases} \texttt{true}, & \text{if } C \text{ does not contain an open period ending after } t, \\ \texttt{false}, & \text{otherwise,} \end{cases} \tag{7}$$

and define the function $\texttt{IsNotEmpty}(C,t) = \neg\texttt{IsEmpty}(C,t)$. The intersection operator $\cap_t^*(C_i, C_j)$ is defined as

$$\cap_t^*(C_i, C_j) = \begin{cases} C_i \cap C_j, & \text{if } \texttt{IsNotEmpty}(C_i \cap C_j, t), \\ C_i, & \text{if } \texttt{IsEmpty}(C_i \cap C_j, t) \text{ and } \texttt{IsNotEmpty}(C_i, t). \\ C_j, & \text{if } \texttt{IsEmpty}(C_i, t) \text{ and } \texttt{IsNotEmpty}(C_j, t). \\ \emptyset, & \text{if } \texttt{IsEmpty}(C_i, t) \text{ and } \texttt{IsEmpty}(C_j, t). \end{cases} \tag{8}$$

The calendar `Intersection of a pair` is used to intersect two calendars. It is also possible to use the more general calendar `Intersection of a set`, but for performance reasons the calendar `Intersection of a pair` is implemented. The results are the same for both types of intersection calendars. From formula (8) it follows that operator $\cap_t^*(C_i, C_j)$ equals the normal intersection operator if the intersection of C_i and C_j is non-empty after time t, while calendar C_j is ignored if it make the intersection empty. If calendar C_i is already empty after time t, then calendar C_i cannot be used in the time-calculations and C_j is used instead. Therefore, operator $\cap_t^*(C_i, C_j)$ is not symmetric in C_i and C_j.

To obtain the finish instant of a wait action, the engine intersects the calendars in the ordered set $\mathcal{C}$ using operator $\cap_t^*$ as described in Algorithm 1.
Let the calendar that is obtained from Algorithm 1 be denoted by $\texttt{Intersection}_t^*(C_1, \ldots, C_X)$. Then

$$\texttt{Intersection}_t^*(C_1, \ldots, C_X) = \cap_t^*\Big(\cap_t^* \big(\ldots \big(\cap_t^* \big(\cap_t^* (C_1, C_2), C_3\big), \ldots, \big), C_{X-1}\big), C_X \Big). \tag{9}$$

Algorithm 1 The implementation of the intersection of a set of calendars using the intersection operator $\cap_t^*(C_i, C_j)$.

Given an instant t.
Given the set $\mathcal{C} = \{C_1, C_2, \ldots, C_X\}$ with $X > 0$.

Let C_{int} denote an intersection calendar.
Let $\emptyset$ denote the empty calendar.

Define $C_{int} = \emptyset$.
for $(i = 1,\ i \leq X,\ ++i)$ **do**
    $C_{int} = \cap_t^*(C_{int}, C_i)$.
end for

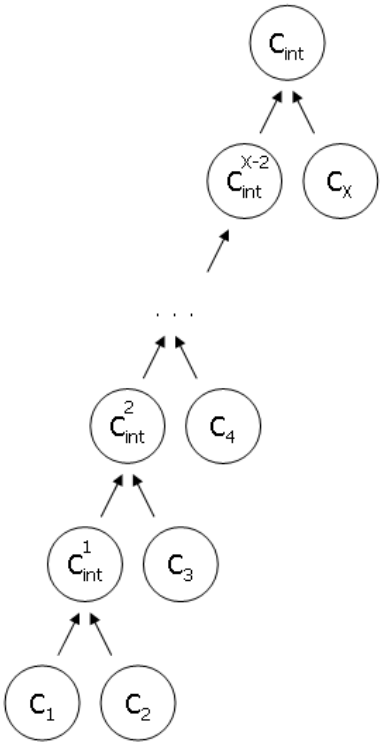

Figure: For every two calendars on the same level, an intersection calendar is obtained using the operator $\cap_t^*$. The result is calendar C_{int} in the level above the two intersected calendars. The calendar at the top is the final intersection calendar.

Some properties of this calendar are listed below. They are proved in Appendix D.

- If the set $\mathcal{C} = \{C_1, \ldots, C_X\}$ contains at least one non-empty calendar after time t, then $\texttt{Intersection}_t^*(C_1, \ldots, C_X) \neq \emptyset$.
- If calendar C_i in the set $\mathcal{C} = \{C_1, \ldots, C_X\}$ is ignored in calendar $\texttt{Intersection}_t^*(C_1, \ldots, C_X)$, then $\texttt{Intersection}_t^*(\mathcal{C}) = \texttt{Intersection}_t^*(\mathcal{C} \setminus C_i)$.
- Given a set of calendars $\mathcal{C} = \{C_1, \ldots, C_X\}$. If instant $t' > t$ is such that $t' \in C_i\ \forall 1 \leq i \leq X$. Then $t' \in \texttt{Intersection}_t^*(\mathcal{C})$.

A calendar C_i is only ignored in the calendar $\texttt{Intersection}_t^*$ if the intersection with the calendar that is obtained from all the higher prioritized calendars is empty. Therefore, the start instant t_A for action A that will be obtained from $\texttt{Intersection}_t^*$ is not included in any open period of $C_i = \{\ t \mid f_A^{r_i}(t) = 1\ \}$. Hence the time-concerning restriction r_i is additionally violated for an action if and only if

(i) r_i is additionally violated for any later start instant as well, or

(ii) any later start instant without an additional violation of r_i will result in an additional violation of a higher prioritized restriction.

Theorem 8.2. *If a start instant of action A is chosen in the set* $\texttt{Intersection}_t^*$*, and if Assumption 4.2 (Start instants of actions) is valid, then the optimal solution cannot be eliminated from the search domain of the binary search.*

Proof. By contradiction.
Let $\mathcal{C} = \{C_1, \ldots, C_X\}$ be the set of calendars with allowed start instants for action A.
Suppose that the optimal solution of the binary search is start instant $\hat{t}$. Let start instant t of the trip be chosen during a step in the binary search such that $t < \hat{t}$ and t is rejected.
(1) If t is rejected because the finish instant of the trip is delayed, then by Assumption 4.2 the finish of the trip will also be delayed for the later start instant $\hat{t}$. But $\hat{t}$ is feasible. $\lightning$
(2) If t is rejected because restriction r_i is additionally violated for action A with start instant t_A, then calendar C_i corresponding to restriction r_i is ignored in the calendar $\texttt{Intersection}_t^*(\mathcal{C})$.
Let the start instant of action A for start instant $\hat{t}$ of the trip be denoted by $\hat{t}_A$. By Assumption 4.2, $t_A \leq \hat{t}_A$. Since $\hat{t}$ is a feasible solution, no restriction is additionally violated for action A with start instant $\hat{t}_A$. Hence $\hat{t}_A \in C_i \ \forall 1 \leq i \leq X$, and by Lemma D.4 every calendar C_i is used in the calendar $\texttt{Intersection}_t^*(\mathcal{C})$. But C_i was ignored. $\lightning$
Therefore, instant $t < \hat{t}$ cannot be rejected, hence the optimal solution cannot be eliminated from the search domain. □

Example 8.5 shows the impact of the order of the restrictions on the time-calculations of the engine.

Example 8.5. Consider again the restrictions in Example 8.4. Let time t denote the finish instant of the travel action that precedes the deliver action, i.e. t = 4:45 PM. If the ordered set of calendars with allowed start instant is given by $\mathcal{C} = \{C_1, C_2, C_3\}$, the calendar $\texttt{Intersection}_t^*$ is given by $\texttt{Intersection}_t^* = C_1 \cap C_2 = C_2$, since $C_2 \cap C_3 = \emptyset$ while C_2 is non-empty after time t. Hence C_3 is ignored and the start of the deliver action is delayed until 5:00 PM, which results in an additional violation of restriction r_3.
If the priorities of the restrictions are changed such that $\mathcal{C} = \{C_1, C_3, C_2\}$, then the calendar $\texttt{Intersection}_t^* = C_1 \cap C_3$ and C_2 is ignored. Therefore, restriction r_2 will be additionally violated. Indeed, since the start instant of the deliver action is delayed until 5:30 PM and the duration of the deliver action is 45 minutes, the finish instant of the deliver action is not included in the time window of the task.

Hence given the restrictions ordered on their importance, the engine will obtain an ordered set of calendars with allowed start instants for an action. If possible, the start instant of that action is obtained from this set such that no restriction is additionally violated. If the normal intersection of all these calendars is empty, then every calendar that makes the intersection with all not ignored higher prioritized calendars empty is ignored in the calendar $\texttt{Intersection}_t^*$.

Suppose that a planning command is executed and a restriction is additionally violated. A warning is generated and the planner decides to tolerate this violation. If the planner then decides to execute the same planning command again, no restriction is additionally violated if the original mechanism is used. The same property is true for the new framework. Since this uses the function $f_A^r(t)$ to determine for every restriction the set of allowed start instants, the calendar of allowed start instants depends on the tolerated values. Therefore, it is not

obvious that no restriction is additionally violated if the same command is executed again. It is proven in Theorem 8.3.

Theorem 8.3. *Suppose that the new framework for the generation of wait actions is used in the time-calculations of the engine. Let a planning command be executed. If there are restrictions additionally violated for an action A, and the planner tolerates every additional violation, then executing the same planning command again will not result in an additional violation for any time-concerning restriction.*

Proof. Suppose that the ordered set of restrictions $r_1, r_2, \ldots, r_X$ with $X \in \mathbb{N}_{>0}$ and r_1 the restriction with the highest priority are used in the time-calculations of the engine. Let $\mathcal{C} = (C_1, \ldots, C_X)$ denote the set of calendars with allowed start instants for action A the first time the command is executed, where calendar C_j corresponds to restriction r_j.
Suppose that the first time the planning command is executed the restrictions $r_{i_1}, \ldots r_{i_J}$ were additionally violated for action A with start instant t_A. Since every additional violation is tolerated by the planner, the corresponding calendars with allowed start instants of action A will be changed for the second execution of the planning command. Denote those new calendars by $C'_{i_1}, \ldots C'_{i_J}$, and denote the new set of all calendars by $\mathcal{C}'$.

Claim 1: $t_A \in \texttt{Intersection}^*_t(\mathcal{C}')$.
Proof: (1) $t_A \in C_i \ \forall i \in \{1, 2, \ldots, X\} \setminus \{i_1, \ldots, i_J\}$, since the corresponding restrictions were not additionally violated for instant t_A.
(2) Every additional violation during the first execution is tolerated, hence the start instant t_A is allowed for every restriction that was additionally violated.
Therefore, $t_A \in C'_{i_j} \ \forall 1 \leq j \leq J$.
(1),(2) $\Rightarrow t_A$ is included in every calendar of the set $\mathcal{C}'$. By Lemma D.4,

$$\texttt{Intersection}^*_t(\mathcal{C}') = C_1 \cap C_2 \cap \ldots \cap C_{i_1-1} \cap C'_{i_1} \cap C_{i_1+1} \cap \ldots \cap C_{X-1} \cap C_X, \quad (10)$$

and $t_A \in \texttt{Intersection}^*_t(\mathcal{C}')$.

Claim 2: $\nexists\ t' \geq t$ such that $t' < t_A$ and $t' \in \texttt{Intersection}^*_t(\mathcal{C}')$.
Proof: Suppose that such an instant $t' \in \texttt{Intersection}^*_t(\mathcal{C}')$ exists. By formula (10), no calendar is ignored in $\texttt{Intersection}^*_t(\mathcal{C}')$, hence t' is included in every calendar of $\mathcal{C}'$, especially $t' \in C_i \ \forall i \in \{1, 2, \ldots, X\} \setminus \{i_1, \ldots, i_J\}$. By Lemma D.4,
$t' \in \texttt{Intersection}^*_t(\mathcal{C} \setminus \{C_{i_1}, \ldots, C_{i_J}\})$. Since the calendars $C_{i_1}, \ldots, C_{i_J}$ were ignored during the first execution of the command, and by Lemma D.3,
$\texttt{Intersection}^*_t(\mathcal{C} \setminus \{C_{i_1}, \ldots, C_{i_J}\}) = \texttt{Intersection}^*_t(\mathcal{C})$.
Hence $t' \in \texttt{Intersection}^*_t(\mathcal{C})$ and $t \leq t' < t_A$. But t_A was the earliest instant in this calendar. $\lightning$

Hence start instant t_A is the smallest start instant in the calendar $\texttt{Intersection}^*_t(\mathcal{C}')$, and the start instant of action A is not changed the second time the planning command is executed. Therefore, the schedule is unchanged after the second execution of the command and no restriction is additionally violated. □

In theory, every restriction should be able to create a calendar with all the allowed start instants for an action. In practice however, there are restrictions for which this is not possible. This will be discussed in the next section.

8.5 Drawback of the mechanism

In the framework described in the previous sections, the engine asks the restrictions for their calendar with allowed start instants for an action. A drawback of this mechanism is that, although for many restrictions such a calendar can be created, it is not possible to create such a calendar for every time-concerning restriction. In Appendix C an overview is given of the restrictions that cannot advise the engine in the time-calculations. The two main reasons for not being able to create a calendar with allowed start instants are given below.

- The restriction is not tested on a single action but on a whole trip or schedule. These restrictions are called schedule restrictions. An example of such a restriction is `Maximum driving time`. Rules are used to limit the driving time that is scheduled within a trip and the start instant of the first external action of the trip is used to determine the rules that are valid for that trip. If the amount of driving time in a trip is unchanged for every start instant of the trip, it would have been possible to obtain a calendar with allowed start instants. However, the amount of driving time that is scheduled can depend on the start instant of the trip[1]. Given a start instant of the trip, it is not possible to predict the amount of driving time that is scheduled. Therefore, every action in the trip should be updated before the total driving time in the trip can be calculated. For performance reasons, it is not possible to update the schedule for every theoretical start instant of the trip to decide if it is allowed. Therefore, it is not possible to create a calendar with allowed start instants for such a schedule restriction.

- Determining the allowed start instants of an action A_1 for a restriction R requires information of an action A_2 with a later start instant than action A_1. In chronological terms, action A_1 is positioned left of action A_2. The time-calculation mechanism of the engine is such that actions are always updated 'from left to right'. Hence when a new start instant for action A_1 is determined, it is possible that action A_2 is not yet updated. If the information of action A_2 is used to obtain a calendar with allowed start instants for action A_1, then this calendar might be incorrect when action A_2 is updated afterwards. As a result, it is possible that restriction R is suddenly additionally violated for the chosen start instant of action A_1. Therefore, action A_2 should be updated for every possible start instant of A_1 to determine if that start instant is allowed. This implies that to determine the start instant of an action, the start instants of other actions should be changed and these actions should be updated. This cannot be embedded in the concept of updating actions 'from left to right'. Therefore, no calendar can be created for these restrictions.
  Examples of these restrictions are some of the restrictions that concern driver legislation. They are tested on a travel and would have to use information of the succeeding action to determine the calendar with allowed start instants for the travel action.

Hence there are calendars for which it is not possible to create a calendar with allowed start instants for an action. However, for the majority of the time-concerning restrictions it is possible to create such a calendar.

[1] There are situations where the amount of driving time that is scheduled for a travel from location A to B is less than the actual amount of driving time that is required to travel from A to B. This depends on the start instant of the travel and conditions on the action succeeding the travel.

A side effect of the framework is that it enables showing the different components of a wait action. This will be explained in the next section.

8.6 Wait reports

For every travel action in the schedule, the planner can ask for a report that specifies the different components of that travel, i.e. driving, pausing or resting. If the travel action starts at time t_F and ends at time t_T with duration 12 hours, the report contains a list of intervals with the description of the action that is planned, for example

$$\begin{array}{lll} [t_F, t_1] & : \text{drive}, & [t_3, t_4] \quad : \text{rest}, \\ [t_1, t_2] & : \text{pause}, & [t_4, t_T] \quad : \text{drive}, \\ [t_2, t_3] & : \text{drive}, & \end{array}$$

where $t_F < t_1 < t_2 < t_3 < t_4 < t_T$.

For a wait action it was not possible to create such a report. With the new framework, this can be made possible, since the creation of a wait action uses the intersection of a set of calendars using operator $\cap_t^*$ to determine the finish instant of the wait action. If a wait action is created before action A, then there is at least one calendar with allowed start instants of action A that (1) is not ignored in the calendar $\texttt{Intersection}_t^*$ and (2) is not open at the finish instant of the action that precedes action A. Let t_F denote the start instant of the wait action t_F and let t_T its finish instant. Furthermore, let R be the highest prioritized calendar for which (1) and (2) hold. Then the wait action will contain the period $[t_F, t_1]$, where t_1 is the start of the first open interval that is contained in R and ends after time t_F. If $t_1 \neq t_T$, then t_1 is not contained within the calendar $\texttt{Intersection}_t^*$. Hence there is at least one other calendar that is used in $\texttt{Intersection}_t^*$ for which (1) holds and that is not open at t_1. Therefore, a second interval $[t_1, t_2]$ is contained within the wait action. This principle can be repeated until the finish of such an interval equals t_T. Note that it is possible that more than one restriction is additionally violated in an interval of the wait action, but only the highest prioritized restriction is used to determine the finish instant of that interval. The report of the wait action will have the same format as the report of a travel, and will consist of a list containing all the intervals within a wait action together with a description of the highest prioritized restriction corresponding to the calendar that is not open in that interval. An example of a wait report is shown in Figure 27, where a wait action is used that consists of three intervals.

Name	DurationInSeconds	FinishInstant	StartInstant	reason
group				
detail	40800	2008-10-02T00:00:00	2008-10-01T12:40:00	CRestrictionTaskTimeWindow
detail	21600	2008-10-02T06:00:00	2008-10-02T00:00:00	CRestrictionAddressOpeningTimesAnyOverlap
detail	18000	2008-10-02T11:00:00	2008-10-02T06:00:00	CRestrictionAllowedNrOfResourcesAtAddress

Figure 27: Example of a wait report.

Hence wait reports inform the planner why an action has to wait, which makes them a very pleasant side effect of the framework.

8.7 Results

The new framework for generating wait actions is implemented because there exist restrictions for which Assumption 4.1 (Behavior of restrictions) is not valid, i.e. the function $f_A^r(t)$ need not be monotonically decreasing for every restriction r. This implies that in theory the binary search method cannot be applied in the beautify, since it is possible to eliminate the optimal solution from the search domain. Therefore, the set of possible start instants I_r^{pos} of action A according to restriction r is constrained such that it is the largest set for which the function $f_A^r(t)|_{I_r^{pos}}$ is monotonically decreasing. If the normal intersection operator is used to obtain the set I of allowed start instants of action A, and the start instant of action A is delayed until the next instant in I, then by Theorem 8.1 the optimal solution cannot be eliminated from the search domain of the binary search. Since no wait action will be generated if the set I is empty, an alternative intersection method is preferred that ignores certain calendars that make the regular intersection empty. For this method, the set I of allowed start instants is given by $I = \texttt{Intersection}_t^*(\mathcal{C})$, where $\mathcal{C}$ is the set of calendars that is obtained from the restrictions, and $\texttt{Intersection}_t^*$ is given by Formula (9). Theorem 8.2 states that also with this intersection mechanism it is no longer possible to eliminate the optimal solution from the search domain of the binary search.

Since it is not possible to obtain a calendar with allowed start instants for every time-concerning restriction, the number of calendars in the set $\mathcal{C}$ can be smaller than the number of restrictions that is tested in the beautify. If the graph $f_A^r(t)$ for such a restriction is not monotonically decreasing, then it is still possible to eliminate the optimal solution from the search domain. However, if the restrictions for which no calendar can be created are not tested in the beautify, and if Assumption 4.2 (Start instants of actions) is regarded to be valid, this new framework prevents the elimination of the optimal solution from the search domain of the binary search.

It is known that planners sometimes overrule the solution of the beautify because they have seen a later start instant for a trip that is also feasible. This suggests that the optimal solution was excluded from the search domain of the binary search. Since planners often do not report this wrong behavior of the beautify, there is no data available of the frequency of these events. There are also no examples available of schedules where the solution of the beautify is not the optimal solution. Therefore, it cannot be concluded that the new framework solves these issues, but it is expected that they will occur less often.

As mentioned before, the use of the new mechanism for generating wait actions will not be limited to the time-calculations of the beautify, but the framework will be used in every time-calculation of the engine. Therefore, the effect of the framework is not limited to the results of the beautify. With the original mechanism, the planner had hardly any influence on the time-calculations of the engine. Due to the new framework, the planner is enabled to specify a list of time-concerning restrictions that should be used in the time-calculations of the engine, and (s)he can even prioritize these restrictions. If a planner does want to be informed of an additional violation of a restriction, but does not want the engine to automatically generate a wait action in case of an additional violation, the system can be configured such that this restriction is excluded from the time-calculations of the engine and is only

tested afterwards. Hence, the influence of the planner on the time-calculations of the engine is increase due to the new framework for generating wait actions.

Due to the new framework, the finish instants of the wait actions can be changed compared to the original mechanism. But the effect of the framework is not only noticeable for the actions in the schedule. Since the number of calculations needed for the generation of wait actions is increased, the amount of time needed to execute a command is increased by the framework. For the beautify command, where the actions are often updated, the required time can increase up to 70%, depending on the configuration of the restrictions. Almost 82% of the additional calculation time is used to create the calendars of allowed start instants for the restrictions, while 13% is used to calculate the calendar $\texttt{Intersection}_t^*$. This is shown in the left diagram of Figure 28, while in the right diagram the time needed to create the calendars is further specified.

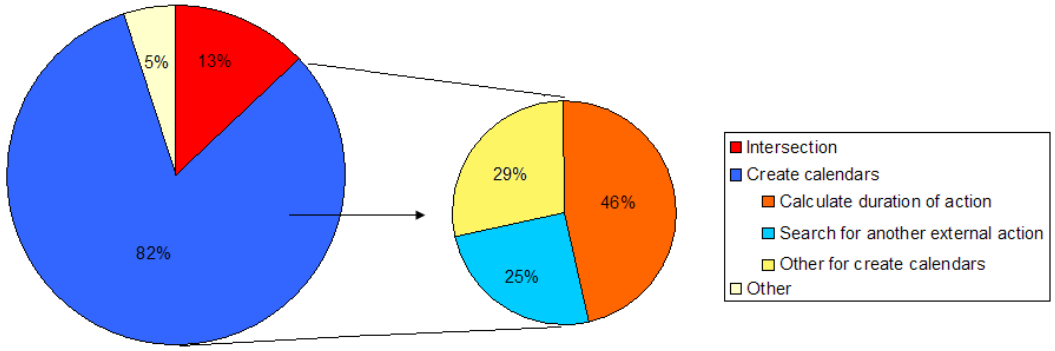

Figure 28: Distribution of the additional time that is needed to generate wait actions with the new framework.

If there are constraints on the finish instant of an action, then for some restrictions the duration of the action is needed to be able to create the calendar with allowed start instants. Of the total amount of time that is needed to create the calendars with allowed start instants, 46% is used to calculate the duration of actions. Another 25% is used to search for a previous action or for the first external action in the trip or couple action. The remaining 29% is used to obtain other information that is needed to create the calendars. Hence the duration of a command that requires time-calculations of the engine will be extended due to the new framework.

The framework does not only repair a deficiency of the beautify, it is also seen as a conceptual improvement of the product. The restrictions are not only used for testing after the time-calculations are performed, but they can also advise the engine during the time-calculations. Apart from the improvement for the beautify, the conceptual improvement itself is important enough to accept the additional calculation time and use the framework in the product.

Part IV
Conclusions and Future work

9 Summary and conclusions

The algorithm that is used by ORTEC bv to minimize the amount of waiting time in a schedule has been discussed in this thesis. This section contains an overview of the analysis of this algorithm, which is called the beautify, and the improvements that have been achieved.

The beautify is a heuristic for solving the minimization problem for the amount of waiting time in a schedule. It reduces this minimization problem to N smaller maximization problems, where N denotes the number of trips in the schedule. The objective of the maximization problem is to delay the start instant of a trip in the schedule until any further delay either results in a delay of the finish instant of the trip, or results in an additional violation of a restriction for some action.

A trip is only allowed to be optimized once, and the schedule that is obtained after the N maximization problems are solved, depends on the order in which the trips are optimized. Choosing the optimal order to optimize every trip is very complex and this problem is not considered in this thesis. The beautify uses a simple algorithm based on the dependency relations between the trips to determine the optimization order. Although this order might not be optimal, and the set of maximization problems is not equivalent to the minimization problem, the beautify provides good results for most schedules.

The method of binary search is used to solve the N maximization problems. In every step of the binary search, a start instant t of the trip is chosen and every action in the trip is updated. The restrictions are tested if the finish instant of the trip that is optimized is not delayed. Although some steps in the algorithm for testing the restrictions in the beautify are redundant, the performance is very good and the effect of the suggested improvements is negligible.

The binary search can only be applied in the beautify if a delay of an action cannot result in an earlier start instant of a succeeding action in the schedule, and if the violation of a restriction for an action A is monotonically increasing as a function of the start instant of A. For both assumptions there are situations in which they are invalid. Therefore, it is possible to eliminate the optimal solution from the search domain of the binary search.
To prevent the elimination of the optimal solution due to the second assumption, a new framework is implemented to constrain the set of start instants of an action. The resulting set of start instants is the largest set for which the second assumption is valid. Due to the framework, the role of the restrictions is extended with the possibility to advise the engine during its time-calculations on the start instants for which there is no additional violation. The new framework enables the planner to specify the constraints that should be respected by the engine during its time-calculations for the actions. Wait actions are automatically generated to prevent additional violations of the corresponding restrictions. The framework also enables to planner to request a wait report that specifies the reasons for the generation

of a wait action. As a result, the product benefits greatly from the introduction of the new framework, and the impact on the performance is considered acceptable.

10 Future work

This thesis addresses the problem of minimizing the amount of waiting time in a schedule. The already existing mechanism to reduce this waiting time, called the beautify, is analyzed. Based on this analysis, the existing algorithm has been improved instead of developing a novel algorithm. The amount of time that is needed to execute the beautify is considerable due to the many calculations in every step of a binary search. For reasons of performance, it is interesting to search for a novel algorithm that optimizes the schedule without the need for a binary search.

The new framework that has been implemented for the generation of wait actions by the engine increases the amount of time that is needed to apply the beautify. Further research should be done to constrain this increase. Two possibilities are given below.

- Stop updating the remaining actions and do not test the restrictions if during a step in the binary search a calendar with allowed start instants is ignored in the calendar $\texttt{Intersection}_t^*$. The corresponding restriction will be additionally violated for every start instant that is chosen in this intersection calendar. Therefore, the chosen start instant of the trip will be rejected.
- Stop updating the remaining actions if the finish instant of an action that is part of the optimized trip exceeds the maximum finish instant of the trip, since it is already known that the finish instant of the trip that is optimized will be delayed.

The amount of time that is needed to apply the beautify can be further decreased if the total number of steps in the binary search is minimized. Further research on this subject should include the following areas;

- Choose a better right boundary T_r of the search domain at the start of a binary search. One option is to determine the sum of the durations of the pickup, deliver and (de)coupling actions and the total driving time in the trip that is optimized. Subtracting this sum from the finish instant of this trip results in a better T_r. Another option is to ignore the constraints for driver legislation and use the method of calculating backwards (see section 4.4.1) to find the latest possible start instant of the trip.
- Use information from the schedule to choose the start instant of the trip in the next step of the binary search.
  - If a restriction is additionally violated, for example if an action ends one hour too late, then choose the next start instant of the trip based on this violation instead of in the middle of the search domain. This requires insight in the complex relation between the start instant of the trip and the start instant of an action.
  - Use the amount of waiting time in the trip to choose the next start instant to try. An approach could be to delay the start instant of the trip with the duration of the first wait action in the trip.

A Details of the implementation of the restriction tester

When a restriction R is tested on an action A, the function `TestTolerateAndGenerateWarning`(A) is applied for restriction R. The implementation of this function can differ for every restriction, but many restrictions use the same implementation, which will be given in this Appendix. For every restriction, a function `GetViolated()` is implemented that returns the value of the violation of that restriction on a particular action. The return type of this function can differ for every restriction and is therefore denoted by `TTolerance`. The original implementation of the function to test the restrictions is given in Algorithm 2.

Algorithm 2 The implementation of the function `TestTolerateAndGenerateWarning`(A) for action A.

```
TTolerance violation = GetViolated(A).          ▷ Get the violation of the
                                                ▷ restriction for action A.
TTolerance tolerance = GetTolerance(A).         ▷ Get the tolerated violation of
                                                ▷ the restriction for action A.
if violation > 0 then
    if violation > tolerance then
        Generate a command to adjust the tolerated value.
        Generate a command to warn the planner.
    else if violation < tolerance then
        Generate a command to adjust the tolerated value.
    end if
else if tolerance > 0 then
    Generate a command to remove the tolerated value.
end if
```

To determine if the restriction is additionally violated, not the return value of `TestTolerateAndGenerateWarning`(A) but the existence of the command to warn the planner is used. To decrease the time needed to test the restrictions in the beautify, a new function `CheckIsViolated`(A) is implemented for every restriction. The implementation of this function is given in Algorithm 3.

Algorithm 3 The implementation of the function `CheckIsViolated`(A) for action A.

```
TTolerance violation = GetViolated(A).
TTolerance tolerance = GetTolerance(A).
if violation > tolerance then
    return true.
else
    return false.
end if
```

B Details of the calendar `Intersection of a set`

Given a set of calendars and a start instant t, the calendar `Intersection of a set` will return the first open interval that ends after time t and that is included in every calendar of the set. This section describes the algorithm that is used to obtain this intersection. Let the function `GetFirstOpenPeriod(` C`,` t `)` return the first open interval of calendar C ending after time t, and let instant T denote the maximum plan date[1] for the schedule. Algorithm 4 describes the intersection of the set of calendars.

Algorithm 4 The implementation of the intersection of a set of calendars. The result is an interval.

Let $I = [I_{left}, I_{right}]$ denote an interval.
Let $I_\emptyset$ denote the empty interval.
Given instant t and the set of $X \in \mathbb{N}_{>0}$ calendars $\mathcal{C} = \{C_1, C_2, \ldots, C_X\}$.

Define the instants $t_{Search} = t$, $t_{PrevFrom} = 0$, $t_{From} = 0$.
Define $I = I_\emptyset$.
repeat ▷ Find the left boundary of the intersection interval.
  $t_{PrevFrom} = t_{From}$.
  for ($i = 1$, $i \leq X$, $++i$) **do**
    $I =$ `GetFirstOpenPeriod(` C_i`,` t_{Search} `)`.
    if $I = I_\emptyset$ **then**
      return $I_\emptyset$.
    end if

    $t_{From} = \max(I_{left}, t_{From})$.
    $t_{Search} = \max(I_{left}, t_{Search})$.
  end for
until ($t_{PrevFrom} = t_{From}$ or $t_{Search} > T$.) ▷ If $t_{PrevFrom} = t_{From}$, then t_{From} equals ▷ the start of first interval that ends after ▷ time t and is included in every calendar.

if $t_{Search} > T$ **then** ▷ The intersection is empty.
  return $I_\emptyset$.
end if

Define $t_{Till} = \infty$. ▷ Find the right boundary of the intersection interval.
for ($i = 1$, $i \leq X$, $++i$) **do**
  $I =$ `GetFirstOpenPeriod(` C_i`,` t_{Search} `)`.
  $t_{Till} = \min(t_{Till}, I_{right})$.
end for
return $[t_{From}, t_{Till}]$.

[1] Maximum plan date: fixed instant somewhere around the year 2030. This date is used to avoid infinite loopings in case of two disjoint calendars without an upper bound. This can occur for two calendars of the type `Repeated` if one calendar is open every morning while a second calendar is open every afternoon and there is no end date specified for these calendars.

C Properties of the time-concerning restrictions

As mentioned in section 8.5, it is not for every time-concerning restriction possible to obtain a calendar with allowed start instants for an action. Table 6 shows two important properties for a time-concerning restriction. The column 'Monotonic' indicates if the function $f_A^r(t)$ is monotonous for the corresponding restriction r, while the column 'Create a calendar' indicates if it is possible to create a calendar with allowed start instants for an action.

Name of the restriction	Monotonic	Create a calendar
`Address opening any overlap`	No	Yes
`Address opening completely within`	No	Yes
`Allowed dropyards`	No	Yes
`Allowed nr of resources at address`	No	Yes
`Appointment`	No	Yes
`Assistant availability`	No	Yes
`Coupled resource not available`	No	Yes
`Day plan start date`	No	Yes
`Driver legislation`	No	No
`Driving duration`	Yes	No
`Early arrival duration`	Yes	Yes
`Ends in day planning`	Yes	Yes
`Fixed action`	Yes	Yes
`Lunch available`	No	No
`Lunch duration`	Yes	No
`Max detour`	Yes	Yes
`Maximum nr of stops in trip`	No	Yes
`Minimal part duration`	No	No
`Order Actions`	Yes	Yes
`Project time window`	No	Yes
`Resource availability at stop`	No	Yes
`Resource calendar`	No	Yes
`Resource work time`	No	No
`Resources idle`	Yes	Yes
`Slave planned inside master`	No	Yes
`Task calendar`	No	Yes
`Task finish time window`	No	Yes
`Task planned according to description`	No	No
`Task predecessors`	Yes	Yes
`Task time window`	No	Yes
`Trip start and end time`	No	Partly[1]
`Wish link instant`	Yes	Yes

Table 6: Overview of two important properties of the time-concerning restrictions.

[1] No calendar is created with allowed end instant of the trip, since generating a wait action if a trip ends too early is only a cover up, while generating a wait action if the trip ends too late will only increase the violation.

D Properties of the calendar $\texttt{Intersection}_t^*$

In this section, several properties of the calendar $\texttt{Intersection}_t^*$ that is defined in section 8.4.2 are stated.

Lemma D.1. *Let $\mathcal{C} = \{C_1, \ldots, C_X\}$ be a set of calendars such that $\textit{Intersection}_t^*(C_1, \ldots, C_{X-1}) \neq \emptyset$ after time t. Then*

1. *$\textit{Intersection}_t^*(C_1, \ldots, C_X) \neq \emptyset$ after time t ,*
2. *$\textit{Intersection}_t^*(C_1, \ldots, C_X) \subseteq \textit{Intersection}_t^*(C_1, \ldots, C_{X-1})$ after time t.*

Proof.

$$\texttt{Intersection}_t^*(C_1, \ldots, C_X) = \cap_t^*(\texttt{Intersection}_t^*(C_1, \ldots, C_{X-1}), C_X).$$
$$= \begin{cases} \texttt{Intersection}_t^*(C_1, \ldots, C_{X-1}) \cap C_X, & \text{if non-empty after } t, \\ \texttt{Intersection}_t^*(C_1, \ldots, C_{X-1}), & \text{otherwise.} \end{cases}$$

Hence $\texttt{Intersection}_t^*(C_1, \ldots, C_X) \neq \emptyset$ after time t, and
$\texttt{Intersection}_t^*(C_1, \ldots, C_X) \subseteq \texttt{Intersection}_t^*(C_1, \ldots, C_{X-1})$ after time t. □

Lemma D.2. *Let $\mathcal{C} = \{C_1, \ldots, C_X\}$ contain at least one non-empty calendar after time t. Then $\textit{Intersection}_t^*(C_1, \ldots, C_X) \neq \emptyset$.*

Proof. Let $C_i \in \mathcal{C}$ be the first calendar that is non-empty after time t. Then
$\texttt{Intersection}_t^*(C_1, \ldots, C_i) = C_i \neq \emptyset$. It follows directly from property 1 in Lemma D.1 that
$\texttt{Intersection}_t^*(C_1, \ldots, C_j) \neq \emptyset$ for every $i \leq j \leq X$. □

Lemma D.3. *Let $\mathcal{C} = \{C_1, \ldots, C_X\}$ such that calendar C_i is ignored in calendar $\textit{Intersection}_t^*(C_1, \ldots, C_X)$. Then $\textit{Intersection}_t^*(\mathcal{C}) = \textit{Intersection}_t^*(\mathcal{C} \setminus C_i)$.*

Proof. Let calendar C_i be ignored in $\texttt{Intersection}_t^*(C_1, \ldots, C_X)$. Then

$$\texttt{Intersection}_t^*(C_1, \ldots, C_i) = \cap_t^*(\texttt{Intersection}_t^*(C_1, \ldots, C_{i-1}), C_i)$$
$$= \begin{cases} \texttt{Intersection}_t^*(C_1, \ldots, C_{i-1}) \cap C_i, & \text{if non-empty after } t, \\ \emptyset, & \text{otherwise,} \end{cases}$$

because calendar C_i is ignored. In both situations,
$\texttt{Intersection}_t^*(C_1, \ldots, C_i) = \texttt{Intersection}_t^*(C_1, \ldots, C_{i-1})$, hence

$$\begin{aligned} \texttt{Intersection}_t^*(C_1, \ldots, C_X) &= \texttt{Intersection}_t^*(C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_X) \\ &= \texttt{Intersection}_t^*(\mathcal{C} \setminus C_i). \end{aligned}$$

□

Lemma D.4. *Let $t' > t$ such that $t' \in C_i \; \forall 1 \leq i \leq X$. Then $t' \in \mathit{Intersection}^*_t(C_1, \ldots, C_X)$ and $\mathit{Intersection}^*_t(C_1, \ldots C_X) = C_1 \cap C_2 \cap \ldots \cap C_{X-1} \cap C_X$.*

Proof. By induction.
$t' \in C_1, C_2 \Rightarrow t' \in (C_1 \cap C_2)$, hence $\cap^*_t(C_1, C_2) = C_1 \cap C_2$. Therefore, the theorem is correct for $X = 1, 2$.
Suppose that $t' \in \texttt{Intersection}^*_t(C_1, \ldots, C_{i-1})$ for arbitrary $1 < i \leq X - 1$, and that $\texttt{Intersection}^*_t(C_1, \ldots C_{i-1}) = C_1 \cap C_2 \cap \ldots \cap C_{i-2} \cap C_{i-1}$. Then

$$\begin{aligned} \texttt{Intersection}^*_t(C_1, \ldots, C_i) &= \cap^*_t(\texttt{Intersection}^*_t(C_1, \ldots, C_{i-1}), C_i), \\ &= \texttt{Intersection}^*_t(C_1, \ldots, C_{i-1}) \cap C_i, \\ &= C_1 \cap C_2 \cap \ldots \cap C_{i-1} \cap C_i, \end{aligned}$$

Hence $t' \in \texttt{Intersection}^*_t(C_1, \ldots, C_i)$ for every $1 \leq i \leq X$ and $\texttt{Intersection}^*_t(C_1, \ldots, C_i) = C_1 \cap C_2 \cap \ldots \cap C_{i-1} \cap C_i$. □

References

[1] Regulation (EC) No 561/2006 of the European Parliament and of the Council of 15 March 2006.

[2] Dantzig, G.B., Ramser, J.H. (1959), The truck dispatching problem, *Management Science*, Vol. 6, p80-91.

[3] Bodin, L., Golden, B., Assad, A. Ball, M. (1983), Routing and scheduling of vehicles and crew, the state of the art, *Computer and Operations Research*, Vol. 10, No. 2, p63-211.

[4] Laporte, G., Gendreau, M., Potvin, J-Y., Semet, F. (2000), Classical and modern heuristics for the vehicle routing problem, *International Transactions in Operation Research*, Vol. 7, p285-300.

[5] Baker, M.H., Ayechew, M.A. (2003), A genetic algorithm for the vehicle routing problem, *Computers and Operations Research*, Vol. 30, No. 5, p787-800.

[6] Savelsbergh, M.W.P., Sol, M. (1998), Drive: dynamic routing of independent vehicles, *Operations Research*, Vol. 46, No. 4, p474-490.

[7] Goel, A., Guhn, H. (2006), Drivers' working hours in vehicle routing and scheduling, *Proceedings of the 9th International IEEE Conference on Intelligent Transportation Systems*, Toronto, p1280-1285, ISBN: 1-4244-0094-5.

[8] Bartodziej, P., Derigs, U., Malcherek, D., Vogel, U. (2009), Models and algorithms for solving combined vehicle and crew scheduling problems with rest constraints: an application to road feeder service planning in air cargo transportation, *OR Spectrum*, Vol. 31, No. 2, p405-429.

[9] Daniels, R.L., Hoopes, B.J., Mazzola, J.B. (1997), An analysis of heuristics for the parallel machine flexible-resource scheduling problem, *Annals of Operations Research*, Vol. 70, No.1, p439-472.

[10] Nguyen, Hung T., Kreinovich, V. (1997), *Applications of continuous mathematics to computer science* , Dordrecht: Kluwen Academic Publishers, Ch. 1.

Index