Generic extensibility for the scheduler of an advanced planning system

Hutter, P.C.

Generic extensibility for the scheduler of an advanced planning system

Christian Hutter

Master thesis, April 2008

Thesis advisors:
prof.dr. L.C.M. Kallenberg (Universiteit Leiden)
dr. S.F. van Dijk (ORTEC BV)

Mathematisch Instituut, Universiteit Leiden
ORTEC BV

Contents

Preface

This master thesis is written to conclude my study Mathematics at the University of Leiden. The research is done at ORTEC, a software company that is one of the largest providers of advanced planning and optimization solutions.

At this place I would like to thank some people. First of all I would like to thank my supervisors dr. Steven van Dijk from ORTEC and prof.dr. L.C.M. Kallenberg from the University of Leiden for their support and discussions. I would like to thank ORTEC in general and the Scheduler Team at the ORTEC Software Development department especially for the opportunity to write this thesis. The latter is also thanked for their critical notes and attention paid to my research.

Last but not least a word of thanks to my family and girlfriend for their motivation and support.

Christian Hutter

Chapter 1

Introduction

This chapter will give a general introduction to this thesis. We will discuss some of the activities and products of ORTEC, focusing on the subject of this thesis. At the end of this chapter we will explain the goals and the outline of this thesis.

1.1 ORTEC

The activities of ORTEC are split into two companies: ORTEC Finance, dealing with *financial* problems, and ORTEC Logistics, dealing with *logistic* problems. This thesis is written at the latter. ORTEC Logistics can be described as a software company that is specialized in developing planning systems that help human planners to be more productive and to keep a better overview of all resources and tasks involved.

The logistic part of ORTEC has developed several planning systems:

- COMTEC: a system for all kinds of planning, some of the important products of this system are
  - ORTEC Transport & Distribution (OTD)
  - ORTEC Service Planning (OSP)
  - ORTEC Passenger Transport (OPT)
- SHORTREC: a system for transport and distribution, this is the predecessor of OTD
- HARMONY: a system for employee scheduling
- LOADDESIGNER: a system for load optimization

All these systems can be seen as decision supporting systems. The COMTEC framework is divided in several subsystems, like the GUI (Graphic User Interface), data management and the scheduler. This thesis will deal with the latter. Figure 1.1 shows an overview of what is discussed in this section.

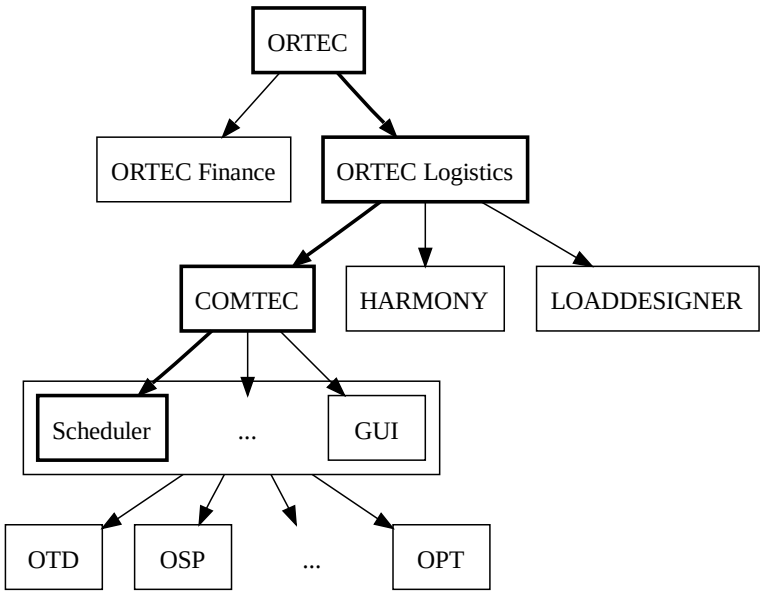

Figure 1.1: *Overview of section 1.1. This thesis deals with the scheduler subsystem of the COMTEC framework.*

1.2 Scheduling

As stated, the COMTEC system deals with all kinds of planning. This is typically an Operations Research (OR) related issue. An example of a classic problem in Operations Research is the traveling salesman problem (TSP). Suppose we are given a number of cities and the costs of traveling from any city to any other city. The problem can be defined as determining the round-trip with the lowest costs that visits each city exactly once and returns to the starting city.

An example of a problem in Operations Research related to the problems ORTEC Logistics deals with, is the Vehicle Routing Problem (VRP). This is a combinatorial optimization problem seeking to service a number of customers with a fleet of vehicles. One of the problems modeled in COMTEC is similar to the Vehicle Routing Problem with Pickup and Delivery (VRPPD): a number of goods need to be moved from certain pickup locations to other delivery locations. The goal is to find optimal routes for a fleet of vehicles to visit the pickup and drop-off locations. In the COMTEC framework a lot of extra restrictions are introduced like time-windows and capacity of the resources.

In Operations Research, *scheduling* can be described as

- decision making in manufacturing and service industries, and

- allocation of scare resources to tasks in time.

In general, scheduling concerns (optimal) assignment of *resources* (often called *machines* in scheduling problems), over time, to a set of *tasks* (often called *jobs*). The problems ORTEC handles, deal with all kinds of planning, among with transport, distribution and service planning.

1.3 The scheduler

A planning in an OR model consists of *n tasks* (like deliver a cargo or visit a client) with processing times, release dates, due dates and other properties, and *m resources* (like a truck or a plumber) with time dependent availability and properties which allow only certain subsets of tasks to be processed by certain resources. The tasks have to be executed by the resources. The basic concept of planning is assigning *actions* to resources to execute the tasks. A *schedule* is necessary to maintain a logically consistent course of actions for each resource while satisfying many constraints. An action has variables such as the time, the free capacity of the resource and the current address.

The *scheduler* is responsible for inserting and removing actions in the schedule, and for calculating the values for the variables of an action. The scheduler ensures the schedule is always in a *consistent* state. *Consistent* means that there is no contradiction. To hold this consistency, the scheduler ensures a number of restrictions is satisfied. These restrictions can either be *necessary*, e.g. if the deliver of a cargo is planned after its pickup, or *optional*, e.g. if a task is not planned too late. Optional restrictions can be turned off by the user.

1.3.1 The core

The scheduler is layered into several modules, of which some are considered *core* and others are *non-core*. The core is product-independent. The non-core modules provide functionality for specific products and work together with the core. For instance, *ORTEC Transport & Distribution* (OTD, see also section 1.1) is built using dedicated modules that use the core to provide functionality that is only relevant for this product. Figure 1.2 visualizes the several modules inside the scheduler. Structures to store data are typically defined in the core, as well as the representation of the schedule. There are several other core modules, like one where the restrictions are defined (there are also product-specific restrictions defined in the product-specific modules). On top of this core the products, like OTD, are built.

Ideally, all functionality in the core is as generic as possible and is specialized for a certain purpose by a product-specific module. For a lot of functionality this holds true, but certain aspects of the core are not extensible in other modules.

An example of a concept that is extensible nicely is *providers*. The scheduler has to communicate with other subsystems in the COMTEC framework. Therefore the scheduler *provides* the information of the schedule it is working

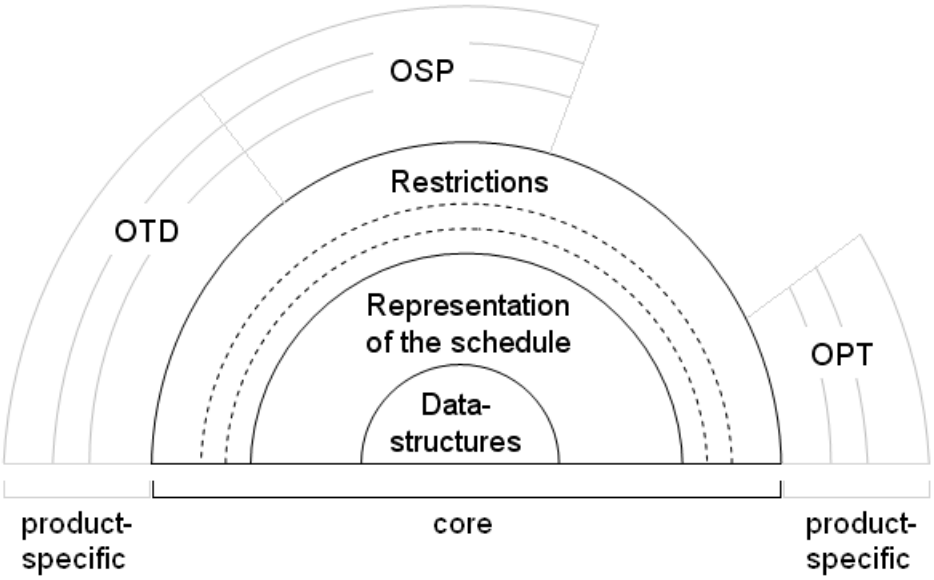

Figure 1.2: *Visualization of the core and non-core modules in the scheduler*

with. This information is stored in tables called *providers*. Most of them are defined in the core, like a provider for trip information which tells the cost of the trip among other information. It is however possible to add a provider from a non-core module. It is also possible to adjust a provider defined in the core. For instance, a product that retrieves the cost of a trip by another method can replace that part of the provider for trip information.

An example of an issue that is not extensible nicely is *actions*. Actions, which are treated more extensively later on, are the activities for one or more resources to execute the tasks. All actions have to be defined in the core. Moreover, it is not possible to adjust an action. One could think of the wish to alter the calculation behavior of an action. For instance, in service planning (OSP) the calculation of the finish time of some action differs from the other products like transport planning (OTD). For now this exception has to be made in the core. If it would be possible to make this exception in a non-core module one has to be careful; changing the calculation behavior could lead to endless loops.

1.4 Thesis goals

As mentioned in the previous section, the core of the scheduler of the COMTEC framework is not as generic and flexible as we want it to be. Several concepts in the core should be modeled otherwise, with as undesirable result we have to modify the core to add product-specific functionality. A scheduler of a planning

system should be generic, in such a way that it is possible to build several products with this scheduler. Goal of this thesis is to make it possible to set up an OR model for all kinds of planning sharing the same basis. Each kind should extend functionality in the basis to its own wishes and needs. Starting-point of this OR model in this thesis will be the scheduler of the COMTEC framework. We will focus on concepts for which extensibility is not trivial to model. Therefore, the main goal of this thesis is:

Generic extensibility for the scheduler of an advanced planning system.

Relating this main goal to the scheduler of the COMTEC framework leads to the following requirements:

- *It should be possible to extend functionality in the core without modifying the core*
- *It should be possible to alter functionality/behavior in the core without modifying the core*

1.5 Thesis outline

Before we are able to state our findings and improvements, there are some concepts that should be treated more extensively. This will be done in chapter 2. After this introduction of basic knowledge one can find our research and modeling to make the scheduler more extensible and more flexible.

In the chapters 3, 4 and 5 where we will present our findings, we will follow this structure:

- *Introduction:* we will introduce the subject of the chapter.
- *Our findings:* we will discuss our findings and improvements. We will also give a proof of concept.
- *Example:* we will give an example of how the presented findings and improvements can be used in general OR modeling. That is, we will try to extract it from the ORTEC context.
- *Results:* we will conclude and summarize the results.

Chapter 2

The scheduler

In section 1.3 we introduced the scheduler of the COMTEC framework. In this chapter we will treat some of the aspects of the scheduler more extensively. After explaining some features of the scheduler, we will discuss *design patterns* in section 2.2. After that we will treat some concepts which will be the subjects of the following chapters, namely *data elements* in section 2.3 and *actions* and the *propagation mechanism* in sections 2.4 and 2.5.

2.1 Features of the scheduler

Languages The COMTEC system is written in the programming languages *Delphi* and *C++*. The scheduler is programmed in *C++*.

Communication with the scheduler as well as storing the schedule takes place in the form of *XML*. XML is a way to represent structured data. This representation is readable for humans as well as computers. The use of XML implies a kind of tree structure of the schedule. This way, it leads to a topological sorting of the actions in the schedule.

Settings Inside the scheduler we have a lot of variables which are not specified in the source code (or filled with a default value) and other preferences which depend on the instance of the program. So, many aspects can be configured. These can either be user-specific or product-specific. For instance, an user would like to turn off some optional restrictions or to specify the default duration of loading a cargo. On the other hand, the instance of the product specifies which modules have to be used. All these values and preferences are stored in the *settings system*. A setting has a name and a value. This value is mostly a string or a Boolean (i.e. true or false). The settings system is a collection of key-value pairs stored in the database.

Tokens Inside the scheduler there is a mechanism to *tokenize*. That means a name, which is represented by a string, is identified with a unique number,

a so-called *token*. With these tokens one can easily use fast integer compares instead of string compares.

2.2 Design patterns

A lot of concepts modeled in the scheduler are based on *design patterns*. These provide solutions to recurring problems in software design. Elsewhere [Gamma et. al.] one can find an introduction to design patterns and an overview of a lot of them. Here we will describe shortly some of the patterns used in the scheduler.

2.2.1 Factory

The *factory method* pattern deals with the problem of creating objects without specifying the exact class of object that will be created. The factory encapsulates the creation of the object. The term *factory* is often used as a collective noun for methods to create objects.

We will give a C++ example. Suppose we have the class `Shape` with derived classes `Triangle` and `Circle`. The `ShapeFactory` will typically have a method

```
Shape* CreateShape( string );
```

After the classes `Triangle` and `Circle` have registered their methods to construct and their associated string (e.g. `"triangle"` and `"circle"`) with the factory, we can create for instance a `Triangle` with `CreateShape( "triangle" )`.

2.2.2 Flyweight

The *flyweight* pattern helps to minimize memory occupation by sharing as much data as possible with other similar objects and storing shared data only once.

2.2.3 Observer

The *observer* pattern defines a one-to-many dependency between objects such that when one object changes, all its dependents are notified. An observer has exactly one *subject*, i.e. the object the observer watches. A subject may have several observers watching it. In the implementation of this pattern in the scheduler, an observer can also be notified by other observers.

2.2.4 Propagator

An extended description of the *propagator* pattern can be found elsewhere [Feiler,Tichy], where it is described as a family of patterns for *consistently updating objects in a dependency network*. The scheduler is based on the pattern where the network is updated immediately after the change. So, all parts of the network remain up-to-date. As a result, the scheduler guarantees consistency.

2.3 Data elements

In a planning there are entities like *trips* (to group actions), *resources* (to model drivers, trucks, et cetera), *calendars* (to model availability), et cetera. These entities can refer to each other, for instance: a driver has a calendar storing its availability. Inside the scheduler we want to argue about these entities, e.g. we want to check if a driver doesn't work outside its calendar. For that reason the scheduler makes a cache of this information from the database. That is, we abstract the data from a table in the database to an object-oriented model in the scheduler. The entities we ask the database for are called *data elements*. Examples are *resource, trip, order* and *calendar*.

When a planner changes the information of a data element, e.g. he changes the calendar of a resource, the scheduler is informed and requests this data element from the database. The scheduler then processes the change and its consequences in the planning.

2.4 Actions

In general, a planning in an Operations Research model consists of tasks, which have to be executed by resources. To be able to do that we assign actions to the resources. In other words, a planning is basically a course of actions for the resources. Examples of actions in the scheduler are:

- *couple*: adding resources to a trip
- *decouple*: removing resources from a trip
- *stop*: stopping at an address
- *pickup*: loading of an order in a resource
- *deliver*: unloading of an order in a resource
- *travel*: driving from one address to another
- *wait*: waiting at an address, for instance till the order is available
- *drive through*: generic action to execute a 'non-transport' task on an address, for instance a task carried out by a service engineer.

Consider for example an order which has to be picked up and delivered. The trailer stands on another address than the driver and the truck. Planning this order could result in the following course of actions:

- couple the driver and the truck,
- travel to the trailer,
- couple the trailer,

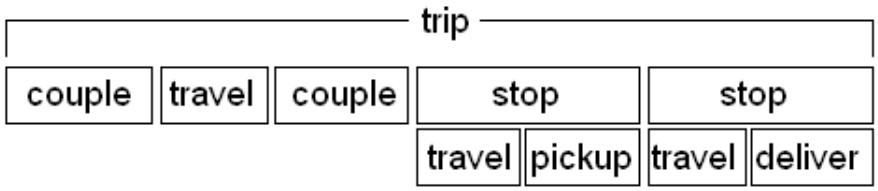

Figure 2.1: *Sequence of actions the way a trip is represented inside the scheduler. The time goes from left to right. An action above another action is the parent of the latter.*

- travel to the pickup-address,
- pickup the order,
- travel to the deliver-address,
- deliver the order.

The resources can either be planned for another order or can be decoupled on one or more addresses. Figure 2.1 shows the representation of this action sequence in the scheduler.

2.4.1 Dimensions

Actions can have properties or variables, which we call *dimensions*. Dimensions store calculated planning information of an action such as the *start time*, the *finish time* and the used *resources*. The value of a dimension can be calculated by the action or it can be retrieved from one of the preceding actions. For instance, the dimension storing the used resources is calculated by the *couple* and *decouple*. Other actions will retrieve this dimension from the last couple or decouple.

2.5 Propagation

As we explained, the schedule is a *consistent* course of actions. Actions are related to other actions, for instance when they use the same resources. So, the schedule can be seen as an implicit graph connecting related actions. Changes to one action can affect other actions. If in the above example the pickup happens later because the opening time of the pickup address changes, the deliver will also take place at a later time. The scheduler is responsible for calculating and propagating such changes, i.e. the scheduler is responsible to maintain the consistency of the course of actions. The propagation mechanism of the schedule is based on the *propagator* design pattern (see section 2.2.4).

This propagation mechanism makes use of *observers*, based on the *observer* pattern. In short, an observer observes other parameters (i.e. objects in the

schedule) that are related to the action. These parameters are mainly an action or a dimension. When these parameters change, the observer is notified and will recalculate the dimensions he is responsible for.

2.5.1 Observers

Each action has a number of observers. In other words, each action *owns* a number of observers. These observers are related to each other in a network. There are mainly three reasons for an observer to be present in such a network:

- Structural: these observers are related to the place in the structure of the plan. The subject is the parameter, i.e. action, in the plan. Examples are observers who watch the *parent* or the *previous brother* of the owner.
- Logical: these observers are a logical product of its predecessors. The subject is again the parameter in the plan. For instance, an observer that is equal to that predecessor which has the earliest start instant.
- Calculating dimensions: these observers are responsible for calculating and updating the value of one or more dimensions. The subject is the value of the dimension. For instance, an observer that computes the finish instant of an action.

Not all dimensions of an action are calculated by one of its observers. The value of some dimensions is obtained by querying the subject of a structural observer. This subject, i.e. another action, will retrieve the value from its observer which is responsible for this dimension or get redirected to another action. For instance, a *travel* action obtains the value of the dimension with the current resources from the preceding *couple* or *decouple* action.

The edges in the network define successor and predecessor relations defined as follows:

- Successor: *an observer A is a* successor *of observer B if and only if there exists a path from B to A and so, A is notified when B is notified*
- Predecessor: *an observer A is a* predecessor *of observer B if and only if B is a successor of A*

Observers cannot be both successors and predecessors of each other, that is, the network is acyclic. An edge in the network from observer A to observer B indicates B is a successor of A. All observers in a network are sorted topologically to ensure all dimensions are computed in the right order.

To make these concepts more clear, we show a part of the network of a *deliver* in figure 2.2. Each box represents an observer. In the figure there is one root-observer, called *CWatch*, with the owner (the action) as subject. In figure 2.1 we saw the relations between the actions *stop*, *travel* and *deliver*. So, the subject of the observer watching the parent, named *CSearchParent*, is the associated *stop*. The value of the address dimension is asked to this *stop*.

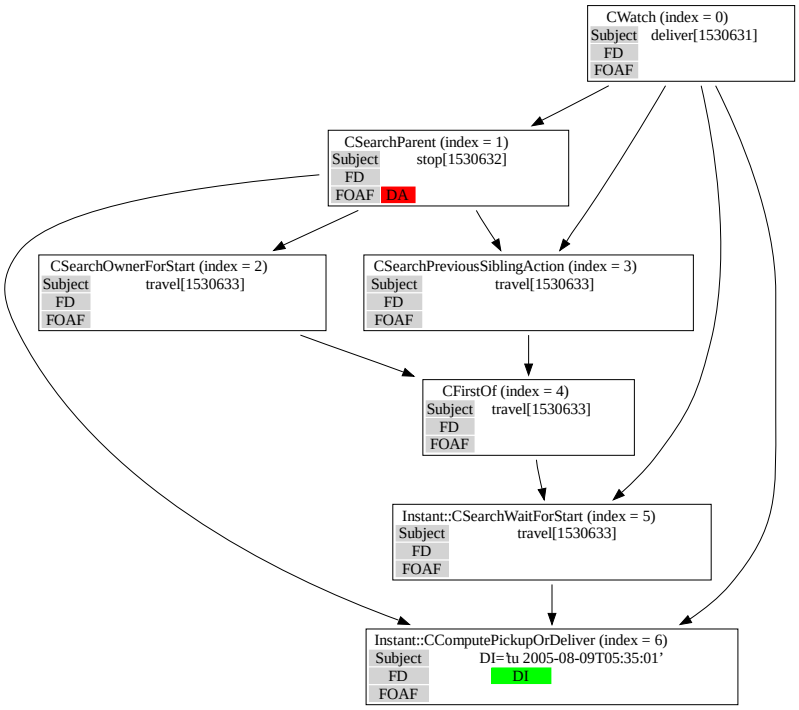

Figure 2.2: *Network of observers of a* deliver *for two dimensions: address (DA) and time (DI). Each node is an observer, showing its name, its subject, finish dimensions (FD, dimensions calculated by this observer) and finish owner ask finish (FOAF, for these dimensions the subject of the observer is asked for).*

The subject of *CSearchPreviousSiblingAction*, the observer watching the previous sibling action, i.e. the brother- or cousin-action, is the associated *travel*. The observer *CSearchOwnerForStart* searches the action with the start value of the dimension to calculate. In our case, the start value of the instant is obtained from the *travel*. The subject of the logical observer *CFirstOf* takes over the subject of the predecessor with the earliest start time. The observer *Instant::CSearchWaitForStart* determines if there is a *wait* action needed before the *deliver*, for instance if the deliver address is not open yet. Finally, the observer *Instant::CComputePickupOrDeliver* has enough information to compute the time of delivering.

As said, for every action there is a network of observers. Different instances of an action, e.g. two couples, have nearly the same network, namely the same observers and the same edges between them. Only the parameters the observers watch are different. This is efficiently modeled by the *flyweight* pattern. The

first instance of an action will create a prototype for the observers network of that action. This instance and all other instances that will follow only have to store the right parameters of the observers.

Every observer has to implement some methods, including the *DoSearch* method. This one will search for the subject of the observer. That is, for structural and logical observers this method will search for the associated parameter, i.e. action, in the plan or it will create an automatic action when this is needed, like a *travel* or a *wait* action. For observers who are responsible for some dimension this method will "search" for the correct value of the dimension, i.e. it will calculate this value.

Chapter 3

Extensibility for data elements

In this chapter we will describe the first project of this thesis, namely improving the extensibility for data elements. First, we will explain what a data element is. After that we will discuss our modeling of data elements.

3.1 What is a data element

In section 2.3 we shortly introduced data elements. Data elements contain all information the scheduler needs to know to generate a proper planning. In an OR plannings model one needs to store and argue about information. For instance, an user of the model specified for transport planning has *resources* (like trucks, trailers and drivers), *tasks* (like picking-up and delivering orders), *addresses* (like addresses of depots) et cetera which have all kinds of information. Both a *resource* and an *address* are not always available, they need a *calendar* to store their availability. These objects with information and references to other objects are called *data elements*. As we mentioned earlier, a data element is an object-oriented mapping of a relational table in the database.

Data elements are also known outside the scheduler. A data element and its information is stored in the database. When such an element is used in the planning, the scheduler asks it from the database. When the information of the data element changes or when it is no longer used in the planning, the scheduler is informed and updates or deletes it.

When we model a new OR problem, we may find it necessary to introduce new data entries (i.e. tables) in the database. We then also have to add a new data element to the scheduler. In the old situation, the core contained all data elements necessary for all products. In what follows we will discuss our proposals to improve the extensibility and flexibility for data elements.

3.2 Adding a data element

There are several places in the core of the scheduler where we use the information of a data element. For example, to *unplan* a data element (i.e. to remove the data element from the planning when it is not used anymore), which is typically done in the core, the scheduler needs to know information that differs per data element.

In the old situation, a lot of work had to be done to add a new data element. To all places the information of a data element is used, we had to add the new data element. Thus, the core of the scheduler had to be adapted. As a result, when we added a data element that was needed for one product, all other products also got this data element, since it was not possible to add it from a product-specific module.

Data elements should not per definition be part of the core. Most data elements will belong to general planning information (e.g. *resources*, *calendars*), but some are product-specific (e.g. *regulations* to model regulations[1] for transporting dangerous goods). We have modeled the framework around the data elements with the following requirements:

1. *Consolidation:* all information related to a certain data element should be kept together
2. *Registration:* it should be easy to add a new data element; a single registration should suffice
3. *Extension:* it should be possible to add a new data element from a product-specific (and thus non-core) module
4. *Encapsulation:* the core should not have knowledge about the data elements

These requirements are met with the introduction of the object *Data Elements Keeper*. All information of a data element, say for instance *resource*, is moved from all over the core to one new place. This information is only accessible by the *Data Elements Keeper* after the data element is registered with him. Anywhere we want information about some or all data elements we ask the *Data Elements Keeper* for it.

The *Data Elements Keeper* introduces an extra level of indirection. That is, there is an extra level between defining and using the information of a data element.

3.3 Data element interface

To concentrate all information about a data element we introduced an interface that all data elements have to satisfy. This interface has a number of methods

[1] An example of such a *regulation* is ADR, an European treaty that regulates the transportation of hazardous materials by road following the guide of the UN Model Regulation

```
<resource id="1">
  <fields homeaddressId="553" calendarId="5" resourcekindId="2" workareaId="1" />
  <capabilities>
    <capability id="6" forbidden="False" available="True" required="False" penalty="0" />
    <capability id="7" forbidden="False" available="True" required="False" penalty="0" />
  </capabilities>
  <rulesets>
    <ruleset id="9926" from="2005-01-01" till="2006-04-01" />
  </rulesets>
</resource>
```

Figure 3.1: *XML representation of resource with id 1. The* fields *tag contains attributes that refer to other data elements (like an address and a calendar). The tags under the* capabilities *and* rulesets *tags also refer to other data elements. So, in this example, resource 1 refers to address 553, calendar 5, resourcekind 2, workarea 1, capabilities 6 and 7 and ruleset 9926.*

and we required all elements that use the interface to implement every method. This implementation of a data element implicitly defines dependencies to other data elements. That is, the data element registers methods to access the attributes of the data element, which may refer to other data elements. In figure 3.1 we show the XML representation of a resource in the scheduler. To obtain the availability of a resource we need a method to access the attribute *calendarId* of a resource. As we will see in the next section, the existence of a method to access an attribute of a data element will define a dependency from that data element to the data element the attribute refers to. So, there is a dependency from *resource* to *calendar*.

Figure 3.2 shows a visualization of the communication from and to the *Data Elements Keeper*. The *Data Elements Keeper* stores all data related to data elements, including a graph with dependencies between data elements which will be the subject of the next section. The keeper offers a number of methods to provide information about data elements, among which a method to get all registered data elements, a method to get dependent elements given a set of data elements (will be discussed later on) and a method to check if an object is a data element. The *Data Elements Keeper* contains the interface (*ABCDataElement*) all data elements have to satisfy. Data elements are inherited from this abstract base class and have to implement the methods of the interface. These methods include a method to get the name and methods to register how to access the attributes of the data element. The latter is done with the methods *RegisterSingleDependency* (to register methods to access the attributes referring to a single data element like a calendar of a resource) and *RegisterMultipleDependency* (for attributes referring to multiple data elements like the rulesets of a resource). Every data element has to register itself to the *Data Elements Keeper*. For product-specific elements this registration as well as the definition of the data element (i.e. the implementation of the methods of the interface) will be done

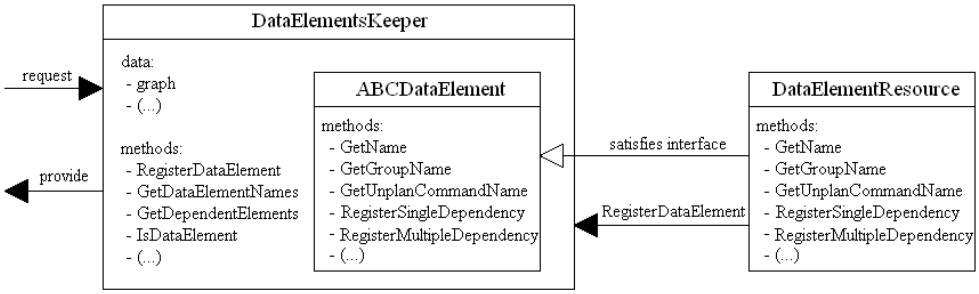

Figure 3.2: *Visualization of the objects* DataElementsKeeper*,* ABCDataElement *and* DataElementResource *and their data and methods. The object* ABCDataElement *(the prefix ABC states Abstract Base Class) is the interface every data element has to satisfy. The object* DataElementResource *specifies all methods inherited from the interface and registers itself to the* DataElementsKeeper*. From all over the scheduler one can request the* DataElementsKeeper *for information.*

in a product-specific module.

3.4 Dependencies between data elements

As mentioned, data elements can refer to each other. The *Data Elements Keeper* keeps track of the dependencies between data elements. The keeper manages these dependencies in a graph $G = (V, E)$ where V is the set of all data elements registered by the keeper and

$$(v_1, v_2) \in E \Longrightarrow \text{an attribute of } v_1 \text{ refers to } v_2 \tag{3.1}$$

where $v_1, v_2 \in V$. Note that in (3.1) the arrow goes only from left to right. That is, an edge in the graph implies that a data element refers to another data element, but the reverse does not necessarily hold. As mentioned in section 2.3, the scheduler contains an object model derived from the relational model stored in the database. This model is not exactly equal to how the scheduler will use this data, i.e. the relations between data elements used in the scheduler is a subset of the relations defined in the model. Therefore, dependencies in the graph are also based upon the way we search for information inside the scheduler. For example, when we want to detect the situation that a driver works outside its calendar, we need a way to obtain the calendar of a resource. For that purpose there is a method with as input a resource and as output the id of the calendar of the resource. The existence of such a method (which we call a *find method* since it finds information for the input) indicates a dependency from *resource* to *calendar*. The *Data Elements Keeper* detects dependencies between data elements by means of such find methods, that is, only relevant

Algorithm 1 *Get dependent data elements.*

1. For every element of the incoming set of data elements execute step 2 and 3
2. Expand the set of dependent elements with the current element
3. Get the set of adjacent vertices. For each vertex in this set, execute the following:
   (a) Get the id's of the already checked elements for this data element
   (b) Get the id's of the dependent elements using the find method associated with this edge.
   (c) For each id found in the previous step execute step 2 and 3 if the id is not empty and it is not in the already checked elements.

dependencies are taken in the graph. Thus, we can refine (3.1) as follows

$(v_1, v_2) \in E \Longleftrightarrow$ an attribute of v_1 refers to v_2 and we can access this attribute

where $v_1, v_2 \in V$. Note that we can access an attribute of a data element if and only if there exists an associated find method. This modeling of the dependencies is extensible easily for new data elements. The *Data Elements Keeper* inserts a node and some edges based on the registered find methods. Figure 3.3 shows the graph of the data elements and its dependencies for an instance of the scheduler modeling transports with dangerous goods. Note that the data elements *dgregulation* and *dgunnumber* are product-specific (the prefix *dg* states dangerous goods).

3.5 Detecting dependent data elements

In the scheduler we have to ensure the data elements are up to date. Therefore we are informed whenever a data element is changed. But, since data elements are dependent, other data elements could also have been changed, new data elements may need to be inserted or unused data elements could be removed. For instance, a changed *address* of a *resource* can refer to an *addresskind* that is not known in the scheduler; this *addresskind* has to be asked from the database. Thus, we need a method to detect all dependent data elements given a set of data elements. This method is implemented using a depth-first search algorithm, see algorithm 1.

We will give an example: let e be the current element in the algorithm, V the set of dependent elements and suppose we get an update of *resource* 1. The algorithm will act as follows:

- $V = \emptyset$

Figure 3.3: *Dependencies between the data elements inside the scheduler. The vertices in the graph are the data elements. The edges denote a dependency. For instance: a trip has one or more resources and a resource can have an address.*

- $e = resource\ 1, V = \{resource\ 1\}$. The adjacent vertices are *resource(), calendar(51), workarea(), address(3), resourcekind(2), configuration(), parameterset(), subcontractor(), costset(12), ruleset()* with the id's found in step 3b between brackets.

- $e = calendar\ 51, V = \{resource\ 1;\ calendar\ 51\}$.

- $e = address\ 3, V = \{resource\ 1;\ calendar\ 51;\ address\ 3\}$. The adjacent vertices are *calendar(52), addresskind(3), costset(12), ruleset()*.

- $e = calendar\ 52, V = \{resource\ 1;\ calendar\ 51, 52;\ address\ 3\}$.

- $e = addresskind\ 3, V = \{resource\ 1;\ calendar\ 51, 52;\ address\ 3;\ addresskind\ 3\}$. The adjacent vertices are *calendar(52), costset(), ruleset()*.

- $e = costset\ 12, V = \{resource\ 1;\ calendar\ 51, 52;\ address\ 3;\ addresskind\ 3;\ costset\ 12\}$.

- $e = resourcekind\ 2, V = \{resource\ 1;\ calendar\ 51, 52;\ address\ 3;\ addresskind\ 3;\ costset\ 12;\ resourcekind\ 2\}$. The adjacent vertices are *address(), calendar(), dgregulation(1), configuration()*.

- $e = dgregulation\ 2, V = \{resource\ 1;\ calendar\ 51, 52;\ address\ 3;\ addresskind\ 3;\ costset\ 12;\ resourcekind\ 2;\ dgregulation\ 2\}$

So, the dependent elements of *resource* 1 are *calendar* 51 and 52, *address* 3, *addresskind* 3, *costset* 12, *resourcekind* 2 and *dgregulation* 2.

As one can see from figure 3.3 the graph contains cycles. This suggests that algorithm 1 can lead to endless loops. However, we can argue that this algorithm terminates. First we remark that in step 3c we test if the element is already checked. So, an endless loop can only arise when every time we come in a certain vertex a data element with a not already checked id is found. Take for instance the cycle

$$address \rightarrow ruleset \rightarrow rulegroup \rightarrow address$$

For a certain *address* we will get at most one *ruleset*. For this *ruleset* we will get a number of *rulegroups*. For each *rulegroup* we will get a number of *addresses*. The *rulesets* of these *addresses* will contain (perhaps a subset of) the set of *rulegroups* already found. In theory the number of *rulegroups* can become very large, but in practice we will walk this cycle at most twice. Of course, one could think of some strange start instances of the algorithm leading to a lot of dependent elements. When this happens in practice, we will get at most all data elements that are present in the database and apparently that was the desired result.

3.6 Example

In this section, we will give an example of using the improved model for a general OR problem. Suppose we have a model with a scheduler with a generic core. Several products, like products for *transport* and *service planning*, are based on this core. In the model we need structures to store the data we are working with. These data structures contain information and can refer to other data structures. In the COMTEC framework this concept is modeled with *data elements*.

In the core of the model several data elements are defined. In the modules for the several products other product-specific data elements are defined. Suppose we have to model a new OR problem, for instance multi modal planning. Multi modal planning will introduce transporting goods with ferries. Implementing this new functionality could result in new standing data, i.e. a new data element. One could think of the desire to have a *contact* of an order. This contact specifies some properties of the order like the prices, the allowed companies of the ferries and the contamination with other orders. An order specifies its contact. So, we have to extend the data model with an insert of a new data element, namely *contact*, and an adjustment to an already existing data element, namely *order*.

The new functionality for the modeling of the multi modal planning should be placed in a new module based on the core. With the introduction of a manager for standard data and improvements to the scheduler presented in this chapter, it is possible to define the new data element in this new module. Moreover, from this module we can adjust the data element *order* with a new dependency to *contact*. Note that the core and other products based on this core are not adapted. From the new module we can manage all changes and inserts to the data elements model. In figure 3.4 we have made a visualization of this process.

3.7 Results

The *Data Elements Keeper* can be seen as a link between supply and demand. It is an extra level of indirection between definition and use of a data element. On one hand, data elements register its data/information with the keeper. On the other hand, anywhere in the code where we want to know something about data elements we ask the keeper for it. This demand can be of different kinds. One can request information of a specific data element, for instance the information how to unplan a *resource*. One also can request information of all data elements, for instance all names of registered data elements. One can also ask the keeper for all dependent elements, given a set of data elements.

The modeling of the *Data Elements Keeper* introduces a number of benefits:

- All information of a certain data element is stored at one place (i.e. requirement 1 from section 3.2, *consolidation*, is met).
- One does not have to worry about ordering or dependencies.

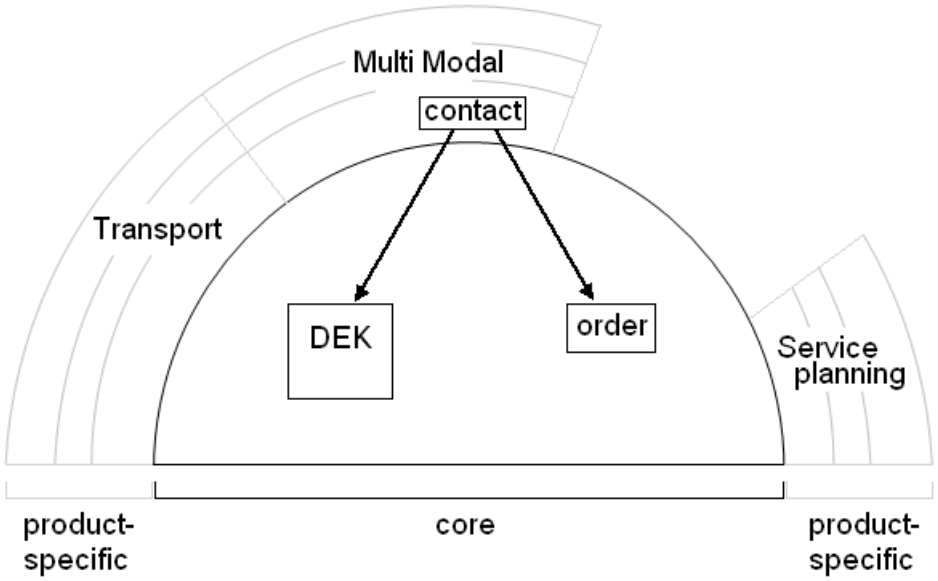

Figure 3.4: *Visualization of the core with several products built on it, like* transport, service planning *and* multi modal planning. *To ensure data elements (needed to store standing data) are flexible and extensible nicely, a manager of the data elements is modeled in the core (named* Data Elements Keeper *(DEK)). Generic data elements like* order *are defined in the core. A product-specific data element, like* contact *in the multi modal module, can be defined in the product-specific module. It registers itself to the DEK. It can adjust already defined data elements. In this case,* contact *adds a dependency from* order *to* contact.

- One cannot forget to specify some information for a data element since the interface requires the necessary information is given.
- One only has to register a data element with the *Data Elements Keeper*. The latter will take care of the rest (i.e. requirement 2 from section 3.2, *registration*, is met).
- Adding a data element, i.e. registering it, can be done from all over the code, from core as well as from non-core modules (i.e. requirement 3 from section 3.2, *extension*, is met).
- The core of the scheduler has no knowledge about data elements. When it needs to know something about them, it asks the *Data Elements Keeper* (i.e. requirement 4 from section 3.2, *encapsulation*, is met).

An OR plannings model needs to manage its data and information. In the COMTEC framework this is modeled with *data elements*. The scheduler needs to manage, order and use these data elements in an efficient way, such that it is extensible easily for new data elements. Since all requirements of section 3.2 are met, the *Data Elements Keeper* approaches this desire.

Chapter 4

Extensibility for actions

As stated in chapter 2 the basic idea of an OR plannings model is assigning *tasks* to *resources*, resulting in *actions* which have to be executed by the resources. In the same chapter we explained that one of the responsibilities of the *scheduler* is to keep the schedule consistent. A change to one action has to be provided to all its related actions. The propagation mechanism therefore makes use of *dependency networks* as described in section 2.5. The thesis goals relating to actions lead to the following requirements

- It should be possible to add a new action from a non-core module
- It should be possible to adjust the observers network of an action from a non-core module

Moreover, to make an OR plannings model generic and flexible, a desired feature is *scriptable actions*. That is, one should be able to configure the behavior of actions easily to some wishes and needs. These wishes and needs will differ per product.

To be able to meet the requirements as described above, we need observers to be flexible, leading to the following requirements for observers:

- It should be possible to define observers in a non-core module
- It should be possible to override already existing observers in a non-core module

We will start this chapter with the modeling of checking an action for some *type*. This will help us to be more flexible, i.e. to adjust or to add actions from non-core modules. In section 4.2 we will discuss adding actions. At last we will talk about adjusting actions in section 4.3.

4.1 Action-types

Actions can be of different *types*. This can be all sorts of type, like a wait action, a transport action, a resource action or an action that is automatically generated by the scheduler. On several places in the scheduler we want to, given an action, check if it is of some type. For example, while calculating the costs of an action we have to ignore the costs of wait actions. At this calculation we check if the type of the action is the wait action type.

There are several ways to model the check if an action is of some type, like a list of all actions per action-type or an $m \times n$-table for m actions and n action-types. We have chosen to model the check via a bipartite graph $G = (V_1 \cup V_2, E)$, where V_1 is the set of actions, V_2 is the set of action-types and

$$(v_1, v_2) \in E \iff \text{action } v_1 \text{ is of type } v_2$$

with $v_1 \in V_1$ and $v_2 \in V_2$. The implemented graph with all actions and its action-types for an OTD instance of the model has the following data:

$$|V_1| = 40, \quad |V_2| = 24 \quad \text{and} \quad |E| = 168 \tag{4.1}$$

In figure 4.1, one can see a sub graph of this graph showing only the actions *travel*, *couple*, *decouple*, *stop*, *drive through*, *pickup* and *deliver*. It shows for example that a *travel* is an action that is automatically generated by the scheduler and that *couple* and *decouple* are resource actions.

One of the advantages of the choice for a bipartite graph is its extensibility. One can easily add an action or action-type to the model by adding a node to V_1 or V_2 respectively. After that, one can set the dependencies to the existing nodes by adding the desired edges to the opposite set of nodes. This extensibility holds in particular for actions in non-core modules. The check mechanism, i.e. the graph, is modeled in the core, it is basic functionality. Nodes and edges can be added to the graph from all over the scheduler, from core modules as well as from non-core modules. A desired property is the following: when one adds an action-type from a certain non-core module, existing actions, especially actions from core modules, don't have to be touched. If we had modeled the mechanism with an $m \times n$-table, we should have explicitly told that there are no relations with the existing actions and the new action-type.

In graph theory one distinguishes between dense graphs, where $\#E \sim (\#V)^2$, and sparse graphs, where $\#E = \alpha \times \#V$ with $\alpha \ll \#V$. From (4.1) it follows that $\#V = \#(V_1 \cup V_2) = 64$, so in our case the following holds

$$\alpha = \frac{\#E}{\#V} = \frac{168}{64} = 2,625 \ll 64 = \#V$$

This result shows that the graph is sparse. This confirms our choice for a graph since dense graphs are very related to tables, in our case a $\#V_1 \times \#V_2$-table.

The actual check if an action is of some type now consists of looking up in the graph if there is an edge from the given $v_1 \in V_1$ and $v_2 \in V_2$. Besides this

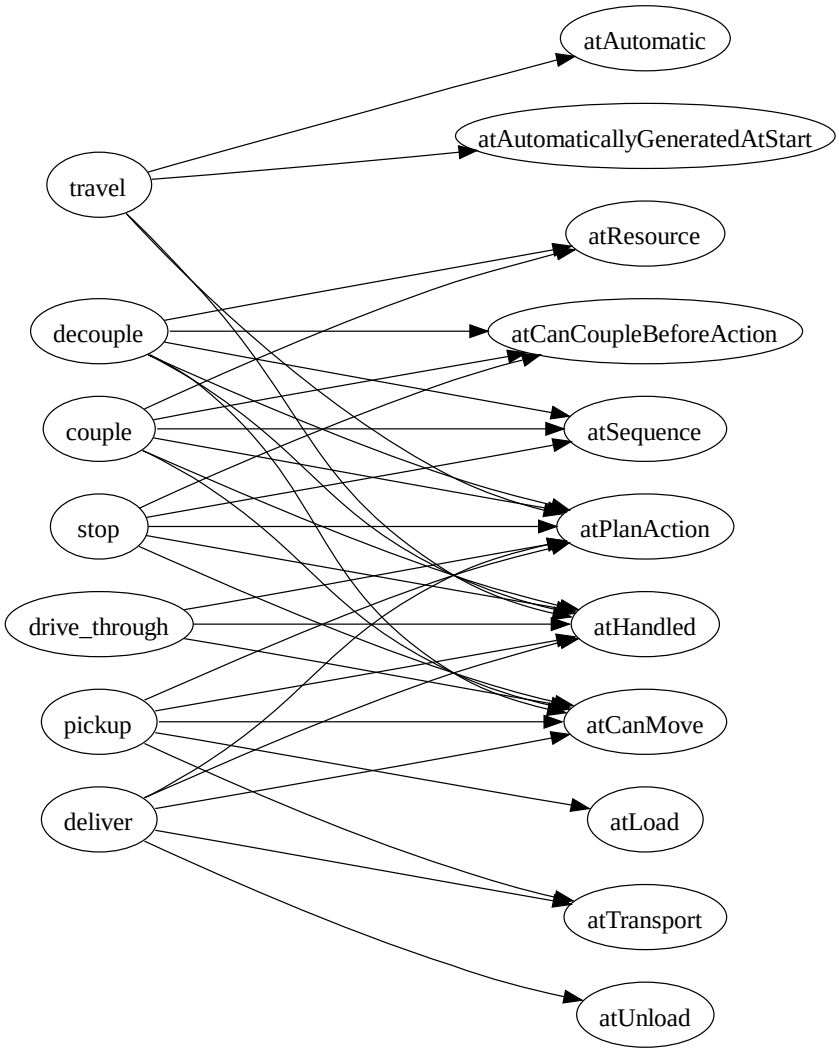

Figure 4.1: *Bipartite graph showing the relations between an action and its types. The left set of nodes represents the actions and the right set of nodes represents the action-types (prefixed with "at")*

Method	Input	Output
Register action	token of action, set of its action-types	-
Check action for	token of action, token of action-type	Boolean
Get actions for type	token of action-type	set of actions

Table 4.1: *Interface of the* Action checker.

check mechanism, the graph offers other functionality: given an action-type one can get all actions which are of this type. These are all incoming edges in the node of the given action-type. Suppose we would like to have all actions that model waiting time. By asking the bipartite graph given this type we get all wait actions that are defined in the scheduler, in core as well as in non-core modules.

The object containing this graph is called the *Action checker*. Its interface consists mainly of three methods, as described in table 4.1.

4.1.1 Performance

Extensibility is one issue you have to deal with when modeling an OR system and can be described as the way one can make changes to an already modeled system. In other words, extensibility deals with reusing solutions from existing OR models. Another important issue is *performance*, which deals with how fast and efficient the model of the system is or will be. The performance of the model defined above should be good since you only have to check for one edge in the graph given the two nodes. However, for one particular check the performance is not good enough. This check is executed so many times that querying the graph is too slow. For this check we use the following mechanism.

As mentioned, the scheduler has a mapping from a string to a *token*, i.e. a unique integer. In the mapping we have created a possibility to reserve a range of tokens. To do so, we call a method of the holder of this mapping with the name and the length of the range. Names that should be placed in this range have to specify the name of the range when registering. Ideally, the range is defined before names are registered with this range. However, we support if a name is registered with an undefined range by defining this range with a default length. Note that adding names to a token range can be done from outside the core. However, it could be possible that a name is added to a token range from outside the core while this name is already tokenized in the core. Then this token would already be tokenized outside the range. Since a name can only be once in the mapping, it can not be present both outside as well as inside the range. Therefore, we should remove the name from outside the range. However, this is not desirable since the scheduler uses a lot of static references to tokens. To prevent this from happening, we force the scheduler to fail if we try to tokenize a name in a token range that is already tokenized outside this range.

An action can be, via its name, represented by a token. For actions of

the type mentioned above we made a range of tokens, which means that the tokenized name is in this range. So, the check if an action is of this type consists of checking if its token is in this range, i.e. if the integer is in the interval. Checking if an integer is in some interval consists of two integer compares, which is very fast. Note that adding actions to this token range can be done from all over the scheduler, from core as well as non-core modules.

This alternative look up for the performance-critical check is faster than the described look up in the bipartite graph. However, it can't replace the latter, since that would require the sets of actions belonging to the action-types to be disjoint: we should define a token range for every action-type, but, since an action has per definition one token, it can only be in one token range. In our implementation of the token ranges it is not possible for two ranges to overlap. One of the reasons for this is that in the scheduler we use a lot of static references to a tokenized name, i.e. at some particular place we ask the token of a name once and every time we return here we use this token to refer. This is a performance critical implementation since we don't have to ask the token to the tokenize mechanism every time. Because of these static references we cannot change the token for a name once it is tokenized. That is, we cannot raise a token for a particular name. Therefore we don't support overlapping token ranges. Suppose we have two overlapping token ranges

$$R_1 = [a, b] \text{ and } R_2 = [c, d]$$

where $0 < a < c < b < d$. These ranges are defined and filled (partly) in the core. In a non-core module we want to add a token to R_1 which should not in R_2. This is not possible except if we permit tokens to change: c and d (and all other tokens in interval R_2) should be raised with one.

4.2 Adding an action

The action is a fundamental element of a scheduler. Modeling actions is a crucial part of finding a fitting solution for an OR plannings problem. All actions used in the scheduler of the COMTEC framework were defined and implemented in the core. A number of actions really belong to the core of the scheduler. However, a lot of actions, although basic, are specific for a product. Actions like *pickup* and *deliver* typically belongs to *transport*. In *service planning* we have other actions like executing tasks on an address.

An action plays various roles, i.e. it needs *types* to base logic on. Furthermore, an action calculates its own dimensions. The schedule is a consistent course of actions. One of the responsibilities of the scheduler is maintaining this course consistent. The main aid for this is observers, which observe other actions. A large part of defining an action is describing its calculation-network. This network will specify the calculation of the action's dimensions.

To add an action to the scheduler a number of steps should be executed, among with:

- defining the calculation-network of observers.
- making a new *propagator*, which contains the prototype (as described in section 2.5.1) of the network among other things.
- adding the action to several lists the action belongs to (this is already centralized by the *Action Checker* described in section 4.1)

4.2.1 Adding an action from a product-specific module

To add an action from a product-specific module we have to add all code about an action to the module where it should be. This will mainly consist of the dependency network of the observers. To be able to really add an action from outside the core we have developed a number of concepts to register the action and to have the desired behavior in the scheduler:

- *Register Action* method. With this method we register the action as parameter and we register the action kind. For this operation we only need to know the tokenized name of the action.
- *Propagator Keeper*: this object manages all propagators (can be compared with the *Data Elements Keeper* from the previous chapter). The *Propagator Keeper* mainly links the tokens of an action with the prototype of the network. When creating an action we look up the prototype for the network via this keeper. With this keeper we can register a propagator from all over the source code. Different actions can have the same propagator.
- *Action Checker* (described in section 4.1): one of the effects of having all actions in the core is that exceptions for non-core actions in some functionality are made in the core (since these actions are all known in the core). With the introduction of the *Action Checker* we can solve this as follows:
  - introduce a new action-type
  - mark all actions the exception is made for as of this type (note that you can do this also from outside the core)
  - at the place the exception is made we ask all actions of this type.

  For instance, when calculating the costs of the wait actions we have to do something special for *service planning wait* actions. At the place of this calculation (in the core) we used to check if the current wait action is this kind of wait action. Since this wait action belongs to the *service planning* module, it is moved with as result it is no longer known in the core. So, we introduced an action-type for *special wait* actions, we mark the *service planning wait* action as of this type (from the *service planning* module) and at the place of the calculation the adjustment for all actions of this type is made.

4.2.1.1 Proof of concept

To check if our modeling is correct, we made an existing action proof of concept. This action handles up- and down-times and is only implemented for *service planning* (OSP, see section 1.1 on page 7). In *service planning*, we model service-related tasks, executed by one resource, e.g. a plumber. That is, we model the day-planning of a resource. As proof of concept, the definition of this action handling up- and down-times, named *Part and unavailable*, is moved to the OSP module. First of all, we add in this module a method to *register* this action. This method does the following:

- it makes a propagator and registers it to the *Propagator Keeper*.
- as long it was not possible to adjust actions we need a callback mechanism from the core to the OSP module. Normally, functionality in a module (e.g. a *core* module) can be called by a module that is built on it (e.g. a *non-core* module), but not the other way around. We are moving *Part and unavailable* to the OSP module, but there are some other actions which use functionality belonging to this action. Since this functionality is moved and we are not able to adjust the concerned actions from the OSP module at this point (later on this will be possible as one can read in the following sections), we need, temporarily, to call functionality defined in the OSP module in the core.
- register sorting context to the *Engine finder*. This sorting context is used to retrieve the *context* of an action. This context will for instance be used to make clear the action's placement in the schedule. Every action has to specify how this context is searched, mostly the *parent* or *grandparent* is used.
- register the action-types to the *Action checker*.

At several places in the core we made exceptions for this action, see the example for the *Action checker* in section 4.2.1. We replaced this with the *Action checker* mechanism. That is, we made a new action-type, we marked *Part and Unavailable* as of this type (from the OSP module) and we make the exceptions for actions of this new type.

In our test application we add actions not defined in the core with default data.

4.2.2 Adding an action from the settings

In the previous section we described how to add an action from a non-core module. To be able to be more flexible and maintainable, we should not have the data of the actions in the source code, but outside of it. It is feasible to place this in the settings-system as XML, so we can model *scriptable actions*. The latter enables us to have different networks for an action. Moreover, scriptable actions are easy to adapt. If a product needs a network of an action, which differs

from the network defined in the core, only the settings have to be updated. The first step is to remove the definition of an action from the source code and add it to the settings-system. This will be the subject of the following subsections. To finish this model we discuss adjusting an existing action from the settings-system in section 4.3.

4.2.2.1 Network parser

The network of observers is no longer defined in the source code, but is read from the settings. The format of the network is XML. As mentioned in section 2.1, XML implies a kind of tree structure, which enables us to add the predecessors of an observer as children of the observer in the XML structure. Figure 4.2 shows the XML of the network of observers of a *deliver* for the dimensions *address* and *instant*, as it was drawn in figure 2.2 on page 18.

When creating the prototype of an action, this XML is read from the settings. It will be parsed to a dependency network by the *Network parser*. First of all, the XML is parsed to a Document Object Model (DOM), which has the same structure as the XML. The *Network parser* first handles the children of the *network*-tag. That is, for every tag the associated observer is created. This creation is deferred to the *Observers factory*, which will be the subject of the next section. This factory has the knowledge to create the observer after the *Network parser* has offered the names of the observer and its predecessors. The factory returns the created observer to the *Network parser*. The latter hooks this one up in the network of the action. After the network is built up, i.e. all observer-tags are converted to created observers, the *Network parser* will handle the *table*-tag. Under this tag, every dimension has to specify how its values should be calculated in this action. That is, the dimension designates an observer (called the *StartOwner* in figure 4.2), which subject should be asked to retrieve the start-value of the dimension. Similarly, if the dimension is not calculated by this action, it should designates an observer (called the *FinishOwner*) to retrieve the finish-value. Otherwise, the dimension should specify the observer (called the *FinishProducer*) who calculates its finish-value.

4.2.2.2 Observers factory

To be able to read the network from the settings instead of defining it in the code, we have to create the observers given a name. That is, the *Network parser* reads the name of the observer. With just this name we have to create the observer. We have modeled this with the *Observers factory*, based on the *factory* pattern (see section 2.2.1).

As mentioned in section 2.5.1 observers are required to implement some methods. One of these methods is the *RegisterPredecessors* method. When hooking the created observer up in the network, this method is called to register which observers are the predecessors of the just created one. The input of this method differs for several observers since the number of predecessors differs.

The *Network parser* offers the name of the observer to create to the factory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<action>
  <network>
    <CWatch name="Target" />
    <CSearchParent name="Parent">
      <argument name="Target" />
    </CSearchParent>
    <CSearchOwnerForStart name="OwnerForStart">
      <argument name="Parent" />
    </CSearchOwnerForStart>
    <CSearchPreviousSiblingAction name="PreviousSiblingAction">
      <argument name="Target" />
      <argument name="Parent" />
    </CSearchPreviousSiblingAction>
    <CFirstOf name="FirstOf">
      <argument name="OwnerForStart" />
      <argument name="PreviousSiblingAction" />
    </CFirstOf>
    <Instant_CSearchWaitForStart name="Wait">
      <argument name="Target" />
      <argument name="FirstOf" />
    </Instant_CSearchWaitForStart>
    <Instant_CComputePickupOrDeliver name="ComputeInstant">
      <argument name="Target" />
      <argument name="Parent" />
      <argument name="Wait" />
    </Instant_CComputePickupOrDeliver>
  </network>
  <table>
    <dimension name="Address" StartOwner="Parent" FinishOwner="Parent" />
    <dimension name="Instant" StartOwner="Parent" FinishProducer="Wait" />
  </table>
</action>
```

Figure 4.2: *Example of the XML of a network. Children of the* network*-tag are the observers. The* argument*-tags under an observer define its predecessors. As last, the table of dimensions is defined, that is, the information how to retrieve the start- and finish-value of the dimension is given.*

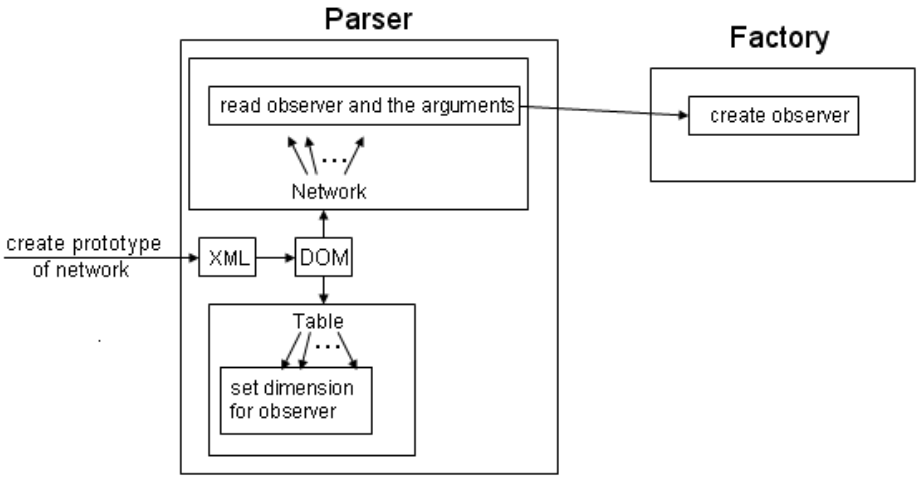

Figure 4.3: *Flow from XML to dependency network. When the prototype of the network of an action has to be created, the* Network parser *reads the XML of the network from the settings and parses it to a Document Object Model. The* Observers factory *is asked to create the observers.*

The latter tries to match this name and the number of predecessors (also offered by the parser) to one of the registered observers. The observers which are going to be created in a network that is read from the settings-system should *register* itself to the factory with its name and the following two methods, which are required to create the observer:

- the *constructor*: to really create the object, the constructor of the class of the observer should be called.
- the *RegisterPredecessors* method: to set the right dependencies in the dependency network the predecessors of the observer should be registered.

With these methods the factory is enabled to create the observer asked from the *Network parser*. In figure 4.3 one can see the cooperation between the *Network parser* and the *Observers factory*.

4.2.2.3 Registering the predecessors

As stated, every observer should be registered with the *Observers factory*. With this registration, the observer gives methods to create itself, including the *RegisterPredecessors* method. Every observer has this method with the preceding observers as arguments, but since the number of predecessors differs per observer, the number of arguments differs per observer. This is a problem if you want a general mapping from an observer (represented by its name) to the method to register the predecessors. This mapping is needed in the *Observers*

factory for the requests of the *Network parser* to create an observer given its name and the predecessors. So, the *Observers factory* has to deal with different interfaces of the *RegisterPredecessors* method.

We show the observer class `CFirstOf` as example. The interface of this class contains its constructor and the *RegisterPredecessors* method:

```
class CFirstOf :  public ABCSearchAndObserve
{
public:
  CFirstOf();
  RegisterPredecessors( CObserve& i_rContext, CObserve& i_rSecond );
  (...)
};
```

When we are going to parse the XML from the settings representing the structure of the network of observers, we have to create an object by calling its constructor and the *RegisterPredecessors* method of the object with the correct arguments. For instance, when we parse a tag named *CFirstOf*, we have to call the constructor of the class above after which we call the method `CFirstOf::RegisterPredecessors` with the observers described in the two children of the tag as arguments. To implement this efficiently and nicely we have to deal some problems. In what follows, we will discuss these problems and we will show some C++ implementations of the solutions.

Member function adapters First we will discuss *member function adapters*, see also page 520 of [Stroustrup]. A member function is, as the name implies, a member of an object. Therefore, the compiler has to know the object this method is called on. In this situation we have the *RegisterPredecessors* method of an observer which has to be called on a specific observer. However, we would like to design a general framework to call this method. Therefore we need delayed execution, i.e. we need to store the function call without the object. At compile time we don't know on which object this method is called. While parsing the network, i.e. at run time, we know on which observer, i.e. on which object, the method should be called. [Stroustrup] treats this problem. A solution is to make a templated class that contains a pointer to the method to call, i.e. the *RegisterPredecessors* method, as follows:

```
template<class TObject, typename TMethod>
class CRegisterPredecessors
{
public:
  CRegisterPredecessors( TMethod i_pfnMethod )
    : m_pfnMethod( i_pfnMethod )
  {
  }

  void operator()( TObject* i_pBaseClass, i_arguments )
  {
```

```
    (i_pBaseClass->*m_pfnMethod)( i_arguments );
  }

protected:
  TMethod  m_pfnMethod;
};
```

where *i_ arguments* is a vector of observers (the predecessors). With the following function we can access this method:

```
template<class TObject, typename TMethod>
CRegisterPredecessors* Get( TMethod i_pfnMethod )
{
  return new CRegisterPredecessors<TObject,TMethod>( i_pfnMethod );
}
```

To really call the *RegisterPredecessors* method (while parsing), we execute the following, where `pBaseClass` is the cast to TObserver:

```
CRegisterPredecessors* pfnRegisterPredecessors
          = Get<TObserver>( &TObserver::RegisterPredecessors );

(*pfnRegisterPredecessors)( pBaseClass, arguments );
```

Partial template specialization Secondly we discuss *partial template specialization*, see also page 26 of [Alexanderscu]. Partial template specialization enables to specialize a class template for subsets of that template's possible instantiations set. In our `CRegisterPredecessors` class we extend the template parameters with an integer, indicating the number of arguments of the *Register Predecessor* method. For every number of arguments we support we have implemented a specialization of this class, with the associated *i_arguments*.

Next, we will show the C++ implementation of the modeling of the `CRegisterPredecessors` classes. We would like to have one mapping from the name of the observer to its `CRegisterPredecessors` class. However, since the template arguments will differ per observer, this mapping has to consist of a name to a class without template arguments. Therefore we introduced the following class:

```
class ABCRegisterPredecessors
{
public:
  ABCRegisterPredecessors() {}
  virtual size_t GetNofArguments() = 0;
  virtual void operator()( CObserve* i_pBaseClass, i_arguments ) = 0;
};
```

The mapping now maps from the name of the observer to a pointer to this abstract base class. All `CRegisterPredecessors` classes will be derived from this

one. Since these will have some code in common (e.g. a member function pointer) we introduced a helper class from which all `CRegisterPredecessors` classes will be derived:

```
template<class TObserve, typename TMethod, int TNofArguments>
class ABCHelper :  public ABCRegisterPredecessors
{
public:
  ABCHelper( TMethod i_pfnMethod )
    :  m_pfnMethod( i_pfnMethod )
  {
  }

  size_t GetNofArguments()
  {
    return TNofArguments;
  }

  void operator()( CObserve* i_pBaseClass, i_arguments )
  {
    DoRegisterPredecessors( safe_cast<TObserve>, i_arguments );
  }

  virtual void DoRegisterPredecessors( TObserve* i_pBaseClass,
                                                    i_arguments ) = 0;

protected:
  TMethod  m_pfnMethod;
};
```

The `CRegisterPredecessors` class now only has to implement the `DoRegisterPredecessors` method. The class without any template specialization looks as follows:

```
template<class TObserve, typename TMethod, int TNofArguments>
class CRegisterPredecessors
    :  public ABCHelper<TObserve,TMethod,TNofArguments>
{
public:
  CRegisterPredecessors( TMethod i_pfnMethod )
    : ABCHelper<TObserve,TMethod,TNofArguments>( i_pfnMethod )
  {
  }

  void DoRegisterPredecessors( TObserve* i_arguments );
};
```

For every number of arguments we support, we have implemented a specialization of this class. We show the specialization for two arguments:

```
template<class TObserve, typename TMethod>
class CRegisterPredecessors<TObserve,TMethod,2>
    :  public ABCHelper<TObserve,TMethod,2>
{
public:
  CRegisterPredecessors( TMethod i_pfnMethod )
    : ABCHelper<TObserve,TMethod,2>( i_pfnMethod )
  {
  }

  void DoRegisterPredecessors( TObserve* i_arguments )
  {
    (i_pBaseClass->*m_pfnMethod)( i_arguments[0], i_arguments[1] );
  }
};
```

Note that the `CRegisterPredecessors` class without specialization has no implemented body for the `DoRegisterPredecessors` method. This way, we ensure we never use a number of arguments we do not support: if we use such a number, we will fall back on this class, since there is no specialization for this number. However, since there is no body for the `DoRegisterPredecessors` method, we will get a linker error while compiling the code.

Meta programming At last *meta programming* is the subject of our discussion. Meta programming is writing code that does (a part of) its job during compile time that otherwise is done during run time. We used this in combination with templates (see [Alexanderscu] and [Abrahams,Gurtovoy]). At compile time we have to know which of the implementations, i.e. specializations, of the `CRegisterPredecessors` class we are going to use for a specific observer. So, we should know the number of arguments the *RegisterPredecessors* method will accept. However, computing a number is typically done at run time, which is obviously too late.

The technique of template meta programming enables us to detect the number of arguments at compile time. For each number of arguments we support we defined a function f with a number of input parameters corresponding to the number of arguments. Here we show these f functions for 0, 1 and 2 arguments:

```
template < typename int TSize >
struct SSized
{
  char padding[ TSize ];
};

template< class C >
SSized<1> f( void ( C::*g )() );
```

```
template< class C >
SSized<2> f( void ( C::*g )( Engine::CObserve& ) );

template< class C >
SSized<3> f( void ( C::*g )( Engine::CObserve&, Engine::CObserve& ) );
```

Every *f* function returns an object (i.e. a struct) containing an array of char's. For each function this array will be of a different size, corresponding to the number of input parameters. To compute the desired integer, we defined the macro `GETNOFARGUMENTS(TObserver)` as follows:

```
( sizeof( f(&TObserver::RegisterPredecessors) ) - 1 )
```

where the argument `TObserver` is the observer class. We call *f()* with the *RegisterPredecessors* function of the observer as input. This will fit in exactly one of the *f* functions we have defined. With the *sizeof* operator we can compute the size of the array that would be returned. Note that we couldn't simply return an integer, because that would happen at run time. The *sizeof* operator is computed at compile time and returns the size of the argument passed to it (in C++, the size of a char is 1 byte). One of the advantages of this technique is that the *f* functions don't need a body because they are not really called. The compiler looks only what would be returned *if* they are called. The output of the *sizeof* operator is given as template argument to the classes described above.

Result With the implementation described above, it is very easy to register an observer to the *Observers factory*: one only has to specify the class. That is, one only has to add the following line:

```
ObserversFactory::RegisterObserver<CFirstOf>();
```

The implementation of the `ObserversFactory::RegisterObserver` method is as follows:

```
template< class TObserver >
void
ObserversFactory::RegisterObserver()
{
  // get the name of the observer as string, e.g.  "CFirstOf"
  string sName = ExtractName( typeid(TObserver) );

  // get a pointer to the CRegisterPredecessors class
  ABCRegisterPredecessors* pRegPred = Get< GETNOFARGUMENTS(TObserver),
                       TObserver>( &TObserver::RegisterPredecessors );

  // add to the mappings
  CObserversFactory::RegisterDataForObserver( sName,
                       CallConstructor<TObserver>, pRegPred );
```

```
}
```

The `Get` method is a method like the `Get` method described in the paragraph about Member function adapters. `CallConstructor` is a function pointer to the constructor of the observer.

4.2.2.4 Proof of concept

In section 4.2.1.1 we explained the case we used to test and proof the concepts of our model. Here we will discuss some issues we had to handle when moving the definition of *Part and unavailable* (P&U) to the settings-system. We removed the method which registered this action from the OSP module. We still need the registration to the callback mechanism as described in section 4.2.1.1, in the next section we will eliminate this "reversed dependency".

The method that takes care of registering the actions from the settings looks for active actions in the settings. Therefore we add P&U as active in the settings. Note that this is only done in the OSP-specific settings. In the default settings this action is inactive and therefore it is not present in the model by default. In our model we decoupled the connection between an action and its network. That is, an action has a *reference* to a network, it does not contain or own a network. This was needed since *part* and *unavailable* slightly differ from each other, but are the "same" action. That is, they have the same network, but for instance the action-types for *part* don't exactly match the action-types for *unavailable*. Beside the Boolean activity-setting we added the following settings for P&U:

- Action-types: a comma separated list of the types of the action. For instance, for *unavailable* the value of this setting is *atHandled, atAutomaticallyGeneratedAtStart, atAutomatic, atSkipWhenSaving, atPlanAction, atProvidesEmptyActionKindSpec, atNoCosts*.
- Network: a reference to the network of observers. For both *part* and *unavailable* the value of this setting is *NetworkPartAndUnavailable*.
- Sorting-context: the sorting-context of the action. For *unavailable* the value of this settings is *UseParent*.
- Token: the name of the action as used in the source code. For instance *unavailable*.

Beside these settings, the XML of the network, i.e. *NetworkPartAndUnavailable*, is added to the settings. This XML has the same structure as the XML in figure 4.2, but deals with more observers and more dimensions.

The scheduler does the following with the active actions: it registers the sorting-context to the *Engine Finder* and the action-types to the *Action Checker*. Also, a propagator is created (if not already present) and registered to the *Propagator Keeper*. This propagator cannot refer anymore to the definition of the network since this is no longer in the source code. Therefore we store the name

of the network at the propagator. When the propagator is going to create its prototype, the network is parsed from the settings as described earlier.

When defining the settings in the settings-system we cannot control if all values are filled in correctly. Moreover, we cannot control if the network in XML has valid data. In the situation of defining this in the source code, this risk is not present since the compiler complains if something is not correct. For that reason, we check the settings while parsing them (so, at run-time, not at compile-time). For instance, we check if all observers that should be created are real observers, i.e. registered with the *Observers factory*. If we detect an inconsistency in the XML or in another setting, we log an error and we throw an exception, i.e. we force the program to fail.

4.3 Adjusting an action

The functionality to add an action from outside the core can be used to make it possible to adjust observers in an already existing dependency network of an action. That is, with this functionality one can *add*, *remove* and *replace* observers. Note that replacing an observer consists of adding the new observer after removing the observer to be replaced. Adding an observer can be done as described in section 4.2.2.

With replacing and adding observers to a network, we are able to alter the behavior of an action. This possibility is needed to solve our problem with the callback mechanism as introduced in section 4.2.1.1.

Replacing an observer will mostly be *overriding* the observer, i.e. the behavior of an observer as defined in the core should be slightly adjusted for the specific product. However, most of the behavior will be the same. In other words, we override an observer with a *shim* around that observer. A shim is derived from the original observer and extends one of its methods. This extension will consist of:

- calling the method of the original observer (this will produce some data, i.e. an object or a value)
- apply some specific functionality to this retrieved data
- return the data

We will give a C++ example: the class `COriginalObserver` contains a method called *DoSearch*. This method returns a pointer to some class named `CParameter`. A shim around this observer will be a class derived from `COriginalObserver` (so the shim contains all functionality of the original observer).

```
class COriginalObserver
{
  CParameter*   DoSearch();
  ...
```

```
class CShim : public COriginalObserver
{
  CParameter*   DoSearch();
  ...
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<network>
  <ReplaceObservers>
    <COSPFinishInstant name="DimOwnFinishInstant">
      <original observer="Core_Instant_CAddDurationToFinishInstant" />
      <original observer="Core_CCloneInstantAndHandleFixation" override="False" />
      <argument name="StartInstant" />
      <argument name="Duration" />
      <argument name="Target" />
    </COSPFinishInstant>
  </ReplaceObservers>
</network>
```

Figure 4.4: *Example of the XML to adjust a dependency network: the observer responsible for the finish instant of the* travel *action is overridden by an OSP-specific observer, named* COSPFinishInstant. *The original observer is either Core::Instant::CAddDurationToFinishInstant or Core::CCloneInstantAndHanlderFixation (depending on an setting). The latter should not be overridden (made clearly by* override="False"*).*

```
};                                  };
```

The *DoSearch* method of the shim will have the following implementation:

```
CParameter*
CShim::DoSearch()
{
  CParameter* pResult = COriginalObserver::DoSearch();
  // adjust pResult
  return pResult;
}
```

To make it possible to adjust an action we have adapted the *Network parser*. The XML offered to this parser will slightly differ from the XML discussed in the previous sections. We should specify if we want to replace or add an observer. If we want to replace an observer by shimming it, we have to tell the original observer. It is possible that this original one is dependent on some setting, e.g. if some setting is true the original one differs from the observer that would be used when the setting was false. The scheduler detects which observer is used originally. It could even be possible that only one of the alternatives of the original observer should be overridden. If one of the original observers should not be overridden, we mark this in the XML. Figure 4.4 shows an example.

As said, a propagator contains a reference to the prototype of the network, in other words, the propagator manages the prototype. Therefore, we added a possibility for propagators to remove an observer from a network. For now we only support removing of an observer if it is replaced by a new one. All observers are present in the network because of some reason, for instance to calculate a

dimension. It is not trivial what should happen to this dimension when the observer is removed. Maybe it can be moved to another observer, but it is not guaranteed that this is possible. However, if we do so, the adjusted network will differ a lot from the original one. Purpose to make it possible to adjust actions is to add or change some functionality of an observer, not to change the whole calculation behavior. Therefore, only supporting to remove an observer if it is going to be replaced suffices and is not a real obstruction.

When an observer is removed from the network, references to this observer have to be adapted. The replacing observer will be added to the network as a new observer and will therefore get another index than the old one.

In section 2.5.1 we mentioned that observers in a network are sorted topologically. After adding or replacing an observer this topology still has to hold. Therefore, we ensure the changes to a network are registered before the prototype is created. This registration will inform the propagator of the changes to make. The propagator keeps this information and processes it when creating the prototype.

When the engine is calculating and a dimension of an action changes, other actions will be notified as a result of the observer mechanism. For instance, when the finish instant of a *travel* changes, its parent, e.g. a *stop*, will be notified as well as the next brother action, e.g. a *couple*. One of the problems encountered in our modeling was that overriding an observer led to endless loops. In these cases we had overridden the *DoSearch* method. As mentioned in section 2.5.1, the *DoSearch* method is responsible for setting the value of the calculated dimension. So, both the original observer as well as the shim set the dimension. When the calculated value of the shim differs from the original one, other actions, e.g. the parent, are notified. Most actions, like the travel, own an observer with the parent as subject. So, the travel will again calculate its dimensions when the shim calculated another value than the original observer. Normally nothing will be changed, but the shim will first call the *DoSearch* method of the original observer. The latter will calculate a different value than the last value calculated by the shim. Therefore, the other actions are notified. So, the finish-value of the *travel* will constantly be modified from the value calculated by the shim to the value calculated by the original one and vice versa.

To prevent this from happening, we never should set a dimension in a shim. We should only adjust the calculation. The original observer should be the only one who is responsible for setting the value of the dimension. With this restriction we ensure the scheduler will not loop infinitely. That is, adjusting the networks on this way will not lead to endless loops in the scheduler. Note that it is guaranteed this way that *if* the scheduler didn't loop infinitely without the possibility to adjust actions, *then* it will not loop infinitely with this possibility as presented above.

4.3.1 Proof of concept

In sections 4.2.1.1 and 4.2.2.4 we discussed our proof of concept. Here we will finish this. We moved all observer-functionality for *Part and unavailable* (P&U) to the OSP module. That is, we removed the observer *CDivideIntoPartsAndAddDuration* from the networks for *travel* and *drive through*. In the core these are replaced by *CAddDurationToFinishInstant* (for *travel*) and *CAddFlexibleDurationToFinishInstant* (for *drive through*). The first one has the following interface:

```
class CAddDurationToFinishInstant
{
  void              RegisterPredecessors( ... );
  CParameter*       DoSearch();
  virtual CInstant  CalculateFinishInstant( ... );
  ...
};
```

The *DoSearch* method is implemented as follows:

```
CParameter*
CAddDurationToFinishInstant::DoSearch()
{
  ...
  CInstant tFinish = CalculateFinishInstant( ... );
  SetDimension( tFinish );
  ...
}
```

From the OSP module we override the observers *CAddDurationToFinishInstant* and *CAddFlexibleDurationToFinishInstant* (which is a derived class of *CAddDurationToFinishInstant*) with a P&U specific shim. The first one, we override with `COSPFinishInstant<CAddDurationToFinishInstant>`. This shim looks like:

```
template < class T >
class COSPFinishInstant : public T
{
  CInstant  CalculateFinishInstant( ... );
};
```

The implementation of the overridden *CalculateFinishInstant* first calls the original *CalculateFinishInstant* method to obtain the original value. This value is handled with some P&U specific functionality.

```
template < class T >
CInstant
COSPFinishInstant::CalculateFinishInstant( ... )
{
```

```
  CInstant tFinish = T::CalculateFinishInstant( ... );
  // adjust tFinish
  return tFinish;
};
```

With this modeling, the observer is replaced with a shim that adjusts the calculation of the dimension the observer calculates the finish-value for. When the new observer is going to calculate, the *DoSearch* method of *CAddDurationToFinishInstant* is called which will call the *CalculateFinishInstant* method. Since the latter is a virtual method, the OSP variant of this method will be called. After the adjustment of the finish-value, the observer does the same as the original one. Note that the dimension is only set in the original observer. This ensures us we will not loop infinitely.

Beside this, we override all wait observers from the OSP module with a P&U specific shim. This one acts analogous to the just discussed shim.

At this point we are able to remove the call back mechanism, since all actions and other objects in the core don't need to use P&U-specific functionality anymore.

4.4 Example

In this section, we will give an example of using the improved model for a general OR problem, just like we did in section 3.6 for data elements. Suppose again we have a model with a scheduler with a generic core. Several products are based on this core. In the scheduler the activities to execute the tasks are defined. That is, in the core as well as in non-core modules the (calculation-)behavior of the activities is defined. Suppose we have to model a new OR problem, for instance planning transport of concrete. Implementing this new functionality could result in new kinds of activities. One could think of the activity *mix* (since concrete is composed of cement, water and other materials, mixing is essential for the production of concrete). Mixing concrete could also be done while traveling, so the generic activity *travel* should be adjusted.

The new functionality for the modeling of the transport of concrete should be placed in a new module based on the core. With the extensions and improvements to the scheduler presented in this chapter, it is possible to add the activity *mix* and adjust the activity *travel* in this new module. This has become possible since we introduced *scriptable actions* to the scheduler. In figure 4.5 we have made a visualization of this process.

4.5 Results

In this chapter we dealt with the modeling of a fundamental element of an OR plannings model. This model needs to manage the activities of the resources to complete the asked tasks. We focused on the possibility to model these activities generic and flexible. That means

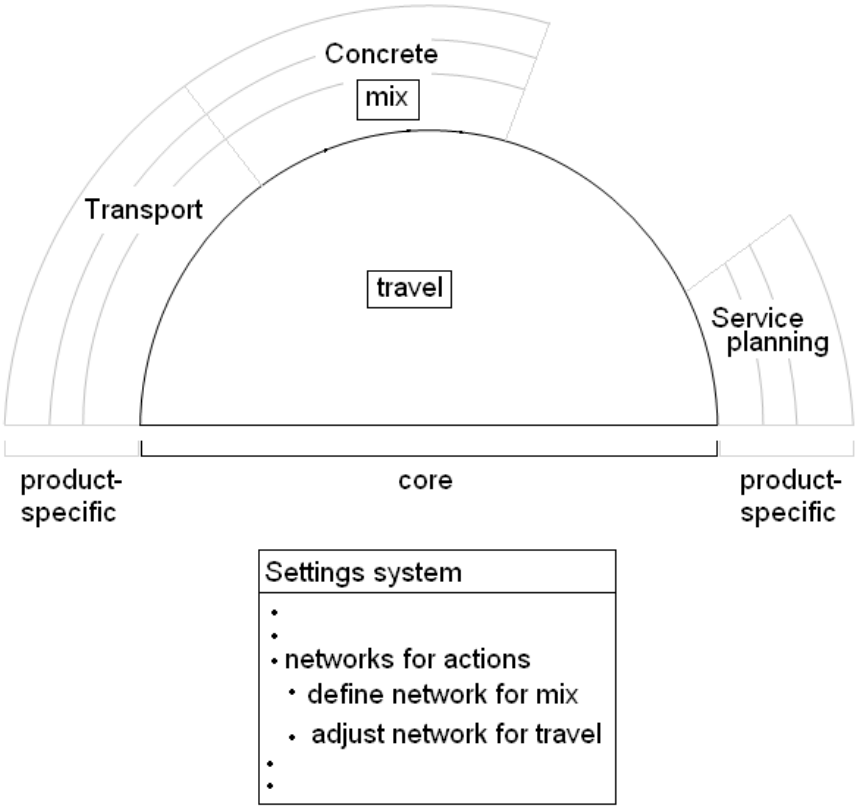

Figure 4.5: *Visualization of the core with several products built on it, like* transport, service planning *and* transporting of concrete. *The latter needs a new activity, namely* mix. *Therefore, this activity is defined in the product-specific module. The biggest part of the definition of an activity is its calculation behavior, as described in its* network. *To make sure activities are flexible and extensible nicely, these networks should be* scriptable. *One way to do so, is defining them in a settings system outside the source code. For the new functionality, the activity* travel *should be adjusted. But since the network of travel is scriptable we can make this adjustment in the settings system and there is no need to adjust the core.*

1. some activities are specific for some products. These are not defined in the core, but can be added to the scheduler from non-core modules.

2. some activities are generic, but different products could have to model these activities differently. These are defined in the core since they are generic. The implementation of them can be adjusted from non-core modules.

Translating this to the OR plannings model we work with, i.e. the scheduler of the COMTEC framework, it means that

1. actions can be added to the scheduler from non-core, i.e. product-specific, modules.

2. observers in an already existing network (i.e. defined in the core) can be replaced by shims, so we can adjust the calculation behavior of an observer. Moreover, new observers can be added to a network.

As already mentioned, one of the important responsibilities of an action is to calculate dimensions. Although the modeling of networks and observers is flexible and extensible at this moment, it is not possible to define dimensions from outside the core. Improving this will be the subject of the next chapter.

Chapter 5

Extensibility for dimensions

In the previous chapter we discussed the extensibility for a fundamental element of an OR plannings model, namely *actions*. One of the important responsibilities of an action is calculating *dimensions*. We handled how to adjust a dependency network (a large component of defining an action). However, to really be flexible it should be possible to define a dimension in a non-core module. This will be the subject of this chapter.

5.1 What is a dimension

In an OR plannings model there are time-dependent variables, i.e. variables that will change when the model is running. For instance, the loaded cargo of a trailer will change throughout the time. These variables belong to activities that are needed to model the tasks executed by resources. In the COMTEC framework these activities are modeled as *actions*. In section 2.4.1 we introduced *dimensions* as variables of actions. A dimension is a fundamental element of the scheduler.

Actions and dimensions are much related to each other. Dimensions are the variables of an action and one of the main responsibilities of an action is calculating its dimensions. To be more flexible and to obtain the extensibility for dimensions, we have to loosen this strong connection between actions and dimensions.

Every action has to specify how it deals with every dimension. This is needed since we want to trace the value of a dimension throughout the scheduler. For instance, during a trip we want to know at every action which resources are involved.

In the old situation, every action defined how it deals with every dimension. Moreover, to retrieve the value of a dimension, find methods have to be added to the *Engine Finder*. To improve the extensibility for dimensions we state the following requirements:

- it should be possible to add a dimension from outside the core, i.e. to define a dimension in a non-core module
- adding a non-core dimension to all networks should be generic

5.2 Dimensions Keeper

At several places in (the core of) the scheduler we want to know the values of dimensions of an action. To approach this demand and other analogous demands we developed a *Dimensions Keeper*. This keeper manages all dimensions. The interface of the keeper contains methods like:

- *RegisterDimension* with a new dimension as input. Dimensions are registered to the keeper with this method which is called in the module the dimension is defined in. This method also adds the dimension to all actions, that is, to all networks.
- *GetDimensionForDescription* with a description as input. A dimension has a describing name, e.g. *OrdersOnBoard*, and a code, e.g. *DOOB*.
- *GetStartValueOfDimension* with an action and a dimension as input. This method will return the start value of the dimension as used in the incoming action.

As we stated earlier, a dimension has to specify how to retrieve its start value, i.e. it has to designate an observer who is the *Start Owner* of the dimension. The subject of this observer is the action that is queried for the desired value. Analogous, a dimension has to specify how to retrieve its finish value. That is, it either has to designate an observer whose subject should be asked for the finish value, or it has to designate an observer who calculates this value. To decouple the strong relation between actions and dimensions we need a default assignment of a dimension to observers of an action. This default has to be defined by every action.

5.2.1 Logical predecessor

By default, for a new dimension we designate the observer watching the *logical predecessor* as *Start-* and *Finish Owner* of the dimension. That is, the start- as well as the finish-value of the dimension is by default retrieved from the action that is, in structure, the predecessor of the action we are adding the dimension to. To process this generically, we require every action to define its logical predecessor. If for a network one of the values of the dimension is retrieved otherwise, for instance because one of the observers calculates the finish-value, one has to specify this. Later on, we will show an example.

5.2.2 The value of a dimension

Retrieving a value of a dimension in general is not trivial since the data-structure is different for several dimensions. For instance, some dimensions store a string, others a map or an instant. The methods to retrieve these values, i.e. the "getters" of these dimensions, should be placed on one mapping. Moreover, the *GetStartValueOfDimension* method of the *Dimensions keeper* should be able to return all these kinds of data-structures. Since all data-structures can be converted to a string, we return all values as a string. A converted instant for example will look like "2007-12-02T17:45:00", i.e. a quarter to six at December the second in 2007.

5.2.3 Engine Finder

In section 4.2.1.1 we described the *Engine finder*. This mechanism will give the value of a dimension for a given action. Two methods have to be added to this mechanism for a new dimension. For a product-specific dimension, these methods can be defined in a non-core module and will only be used outside the core (otherwise the dimension should not be defined outside the core).

5.2.4 Delay

As well as the changes to action networks, adding a dimension to a network is delayed. That is, when a new dimension is registered, the networks are not yet created. Therefore, the new dimension is stored in the propagator of each action. When, later on, the prototype of an action is created, the new dimension is processed. After that, the network is sorted topologically to ensure the already existing as well as the new dimensions are computed in an efficient way.

5.2.5 Register a dimension

To meet the requirement that adding a dimension to all networks should be generic, we introduced a method to register a dimension. This method will add the dimension to all actions. To be more precisely, this method will add the dimension to all the propagators known by the *Propagator keeper*. These propagators will add the dimension to the networks as described above.

It is guaranteed that all propagators are known by the *Propagator keeper* when a dimension is registered. The registration of the propagators is partly programmed in the source code of the core and partly read from the settings. Reading the settings to register propagators goes as follows: in the core we detect and register all active actions in the settings. So, the registration of the propagators is done while loading the core. The registration of a new dimension is also read from the settings. However, these settings are read when the module with this dimension is loaded. Since product-specific modules are loaded after the core is loaded, it is guaranteed that all propagators are known whenever the first dimension is registered.

The method to register a new dimension also takes care of the exceptions on the default addition to a network. That is, the method reads some XML from the settings system. This XML defines the observers to obtain the start- and finish-values different from the default observers. An example of this XML will be given in the next section where we will discuss a proof of concept.

Beside this, the register method also registers the dimension to the *Dimensions keeper*.

5.3 Proof of concept

Analogous to the extensibility for actions, we will discuss one case we worked out in our modeling for adding dimensions from outside the core. One of the products built on the core of the scheduler models the transport of *containers*. For this instance of the scheduler, we need the dimension *Load state*, which is not used in the other products. *Load state* stores the quantity the container is filled with. Values of this dimension can be *empty*, *loaded*, *full* and *unchanged*. To prove the concept we described above, we moved the dimension *Load state* to the *Container* module.

In the core, this dimension was defined and it was added to all actions. First, we removed this dimension from all actions, since it will be added to all actions from the *Container* module. Moreover, we removed two observers calculating the finish-value of *Load state* (this was the only dimension these observers concerned). The definition of these observers is moved to the *Container* module. From this module we add these observers to the associated networks with the functionality described in section 4.2.2.

From the *Container* module we also added *Load state* to all actions. Figure 5.1 shows how the exceptions are made for actions not handling the default addition.

We also had to add the dimension and the methods to get the start- and finish-value to our test-application.

5.3.1 Compare old and new situation

Proving the concept also consists of comparing the old and the new situation. That is, we have to ensure the observer networks with our improvements are similar to the networks in the old situation when the *Container* module is loaded. If this module is not loaded the networks will definitely be changed since the dimension *Load state* and the two associated observers are not present in the new situation. If the module is loaded, the observer networks should be similar. To test this, we compared the visualization of these networks like the one in figure 2.2 on page 18. Remark that these networks are sorted topologically. Since a directed acyclic graph has one or more topological sorts, the topological sorting of an observer network is not unique. Therefore, the networks in the old and new situation could slightly differ from each other. However, we have to

```
<?xml version="1.0" encoding="UTF-8" ?>
<dimension code="DLS" name="LoadState">
  <CDefineStop FinishOwner="pLastChildOrLogicPredecessor" />
  <CDefineRelease StartOwner="pTarget" />
  <CDefinePickurOrDeliver FinishProducer="pDimOwnLoadState" />
  <CDefineDriveThrough StartOwner="pWait" FinishProducer="pDimOwnLoadState" />
  <CDefineDecoupling StartOwner="pOwnerStart" FinishProducer="pDimOwnLoadState" />
  <CDefineDecouple FinishProducer="pDimOwnLoadState" />
  <CDefineCoupling StartOwner="pStartLoadStateAndOnHold" FinishProducer="pDimOwnLoadState" />
  <CDefineCouple FinishOwner="pLastChildOrPredecessor" />
  <CDefineAllOtherTravelDetails StartOwner="pParent" FinishOwner="pParent" />
  <CDefineAcquire FinishProducer="pDimOwnLoadState" />
</dimension>
```

Figure 5.1: *XML used to specify the observers for the start- and finish-value of* Load state *for the networks that will differ from the default extension for this dimension. For instance, the network of the* stop *action will retrieve the start-value of this dimension by the default way, namely the observer named* pLogicPredecessor. *The finish-value will be retrieved by the observer named* pLastChildOrLogicPredecessor.

ensure these networks are *isomorphic*[1].

Ensuring this isomorphism is enough: the visualization shows the observers, the predecessors of each observer and the dimensions of each observer. In other words, this visualization tells all information we considered with the creating of networks. For all actions, an isomorphism from the network in the old situation to the network in the new situation is found.

5.4 Example

In this section, we will give an example of using the improved model for a general OR problem, just like we did in section 3.6 for data elements and in section 4.4 for actions. Suppose again we have a model with a scheduler with a generic core. Several products are based on this core. In the scheduler the variables of an action are defined. That is, in the core as well as in non-core modules the dimensions, i.e. variables that are time-dependent, are defined. Suppose we have to model a new OR problem, for instance transporting liquids. Implementing this new functionality could result in new dimensions. A reasonable new dimension could be the *temperature* of the liquid.

The new functionality for the modeling of transporting liquids should be done in a new module based on the core. With the introduction of scriptable dimensions and improvements to the scheduler presented in this chapter, it is possible to add this dimension in this new module. That is, other programs based on the core are not adapted with this new dimension. In figure 5.2 we have made a visualization of this process.

[1] Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called isomorphic if there is a bijection f between V_1 and V_2 with the property that $x \in V_1$ and $y \in V_1$ are adjacent if and only if $f(x)$ and $f(y)$ are adjacent in G_2.

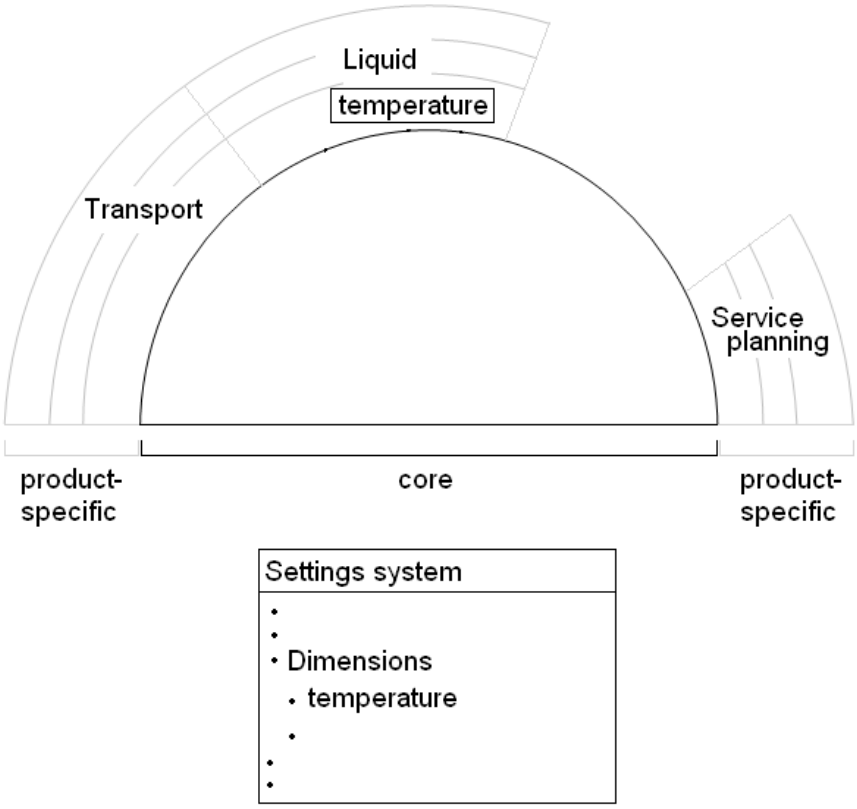

Figure 5.2: *Visualization of the core with several products built on it, like* transport, liquid *and* service planning. *The new dimension* temperature *is added from the new module. This new dimension should be added to all activities defined in the scheduler. Each activity has to provide the default way to add a dimension. So,* temperature *has only to specify the non-default additions. The latter is done using the settings system.*

5.5 Results

As stated in section 5.1 one way to improve the extensibility of a dimension is to decouple actions and dimensions. To achieve this, an action has to define a default behavior of a dimension. This way, actions and dimensions don't have to be aware of each other: the behavior of a dimension doesn't have to be defined in the action. When a dimension is added to the constructed framework of actions, it will act as the default behavior defined by every action. When the dimension needs a special behavior for some action, the dimension has to make this exception. Mostly such an exception will be coupled with a new observer containing the calculation of the finish value of the dimension.

Combining the results of the previous chapter (as stated in section 4.5) and the results of this chapter, leads to a flexible modeling of *actions*, which is extensible nicely. That is, defined actions can be adapted to obtain the functionality as desired in a specific product.

Chapter 6

Conclusions

In this chapter we will conclude this thesis by discussing our findings and further developments.

6.1 Results

In this thesis we discussed setting up an OR plannings model. We focused on the desire to get a flexible and generic model that is extensible easily. That is, the model should consist of a generic *core*. This core contains plannings functionality and modeling of plannings issues. It should be possible to use the model to build products for all kinds of planning. Such a product should be built with several modules on top of the core.

In this thesis we handled some fundamental issues one encounters setting up a plannings model. First of all, in chapter 3, we discussed the modeling of managing the data inside the model. For instance, a *resource* contains data as an *address* and its availability. The latter is stored in a *calendar*, which can also be used for opening times of a depot. So, data can refer to other data. We introduced an extra level between defining these kinds of data and using it, leading to flexibility.

Another issue we handled models the basic idea of planning, namely the *activities*, i.e. *actions*, of the resources to complete the planning-tasks. In chapters 4 and 5 we discussed that an OR plannings model needs the possibility to adjust the behavior of these *actions*. Moreover, product-specific *actions* should not be part of the core. To really be flexible, it should be possible to adapt all properties of an *action*. Therefore we decoupled the definition of the *variables* of an *action*, i.e. *dimensions*, from the definition of the action.

With the improvements presented in this thesis, modeling new OR problems with the COMTEC framework has become easier. That is, building new products on the core of the scheduler has been improved: we are able to specify most of the functionality in a new, product-specific, module. Moreover, already existing products in COMTEC could be decoupled, leading to a smaller, more

generic core.

Our improvements greatly enhance the generality and flexibility of ORTEC's advanced planning systems:

- new products can freely add data
- new activities and associated attributes can be defined when convenient

6.2 Future work

Here we will describe some further developments.

6.2.1 Data elements

In section 3.5 we discussed an algorithm to detect all dependent data elements given a set of data elements. The order this graph is passed is random. That is, we start in the graph with the first element of the given set. One place we use this detection of dependent elements is at the removal of an element. When a data element is not used anymore in the planning it can be erased. Since this element can refer to other data elements, it could be possible to erase more elements. However, a data element cannot be erased if another element refers to it. To break through this "loop", we keep passing the graph as long there are candidates to erase. If in a round no elements are erased, we stop the algorithm.

This is not the most efficient way. It would be better to repeat the following:

- handle a node without incoming vertices in the graph
- remove the node and the associated vertices from the graph

However, since the graph contains cycles (as one can see in figure 3.3 on page 26), it is not trivial to develop a generic algorithm.

6.2.2 Networks

We introduced possibilities to adjust networks by adding or replacing observers and by adding dimensions. However, it is still not possible to add dependencies to an existing network (i.e. to add edges to a network of observers like the one in figure 2.2 on page 18), to ensure an observer gets notified when another recalculates its subject. The desire to do this can proceed from replacing an observer. However, adding an extra dependency could be unsafe: it is not trivial to ensure the scheduler will not loop infinitely. Moreover, the number of predecessors is specified for each observer and used in the modeling of the *Network parser* and the *Observers factory*.

Bibliography

[Eckel] Bruce Eckel, *Thinking in C++, Volume 1: Introduction to Standard C++*, Pearson Professional Education, 2000

[Gamma et. al.] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman Inc., Boston, 1995

[Feiler,Tichy] P.H. Feiler, W.F. Tichy, *Propagator: A Family of Patterns*, Proc. Technology of Object-Oriented Languages and System (TOOLS 23), IEEE CS Press, July 1997

[Stroustrup] B. Stroustrup, *The C++ programming language*, Addison Wesley Longman Inc., Boston, 1997

[Alexanderscu] A. Alexanderscu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, Boston, 2001

[Abrahams,Gurtovoy] D. Abrahams, S. Gurtovoy, *C++ Template Metaprogramming*, Addison Welsy, Boston, 2005