Master Thesis

Implementation of Various Techniques for the Optimisation of Business-Performance Frameworks

Jan Kischkel
Matrikelnummer: 20216175
Praktische Informatik

BT Group
Research and Venturing
Intelligent Systems Research Centre
Dr. Martin Spott

Braunschweig, June 10, 2007

Jan Kischkel
Contents

Eidesstattliche Erklärung II

1 Introduction 1
   1.1 Motivation ............................................. 1
   1.2 Overview .............................................. 2

2 Real Time Business Intelligence 3
   2.1 Business Performance Frameworks ...................... 3
      2.1.1 Determination of the Different Layers ............ 4
   2.2 Functions .............................................. 5
      2.2.1 Modelling and Updating Business Performance Frameworks 5
      2.2.2 What-if-Analysis .................................... 6
      2.2.3 Real Time Monitoring .............................. 6
      2.2.4 Forecast and Control .............................. 6
      2.2.5 Target Optimisation ............................... 6
   2.3 Architecture .......................................... 7
      2.3.1 Communication Between the Layers ................. 9

3 Requirement Analysis 10
   3.1 Previous State of the Target Optimisation ............ 10
      3.1.1 The Implementation ................................ 10
      3.1.2 The Problems ....................................... 10
   3.2 Desired Functionality .................................. 11

4 Implementation - First Steps 12
   4.1 The Formula Editor .................................... 12
      4.1.1 The Graphical Interface ......................... 13
      4.1.2 Input of the Properties .......................... 14
      4.1.3 Event Handling .................................... 15
      4.1.4 Extending The Class ................................ 15
1 Introduction

1.1 Motivation

The modern business world is constantly confronted with fundamental changes of the requirements of their environment. “Globalisation, Internationalisation and Liberalisation of the global market pose a growing challenge to companies” [1]. With the decreasing product life cycle and the fast circulation of knowledge, it is hard to get into a leading position and even harder to stay there. This fierce competition demands a continuous analysis of the customer requirements, as well as the own economic efficiency. If something needs adapting, it is mandatory that the decisions are based on the most current and significant data regarding the business-performance. To achieve this broad basis of information, constant monitoring with high data quality is required. Based on this, the productivity of the business-processes needs to be optimised to withstand the current and upcoming challenges.

“In short, companies only have bright future prospects when crucial factors can be identified and controlled” [1]. These factors include financial matters, staff, customer satisfaction and the external influences of the business. The technologies of today’s computer science help tremendously to achieve these targets in an economic way. Distributed systems combined with database management systems are affordable for almost every bigger company nowadays, so the remaining question is how to use this equipment to get a real advantage in the market.

The basis of this thesis is one answer to this question: The RTBI (Real-Time Business Intelligence) system. This solution offers facilities for the modelling of all layers of real-life processes into performance frameworks, that can be used to visualise and analyse the business, or optimise it for a certain objective. This thesis focuses on the latter aspect and presents different solutions for the definition of objectives and the non-trivial task of determining their ideal framework-settings. The more sophisticated solutions will feature the iOpt toolkit, that has been integrated into RTBI for its powerful optimisation algorithms.
1.2 Overview

Following the first chapter is a brief introduction to the RTBI system, that gives an overview of the functions and the architecture. It is just an outline, as an in-depth discussion of every aspect would go beyond the scope of the thesis.

The third chapter will review the current state of the implementation for the target optimisation and explain why the reengineering had to be done.

The fourth chapter introduces an editor for the definitions of objectives. After the discussion of the main decisions of the implementation process, an initial solution is shown.

The fifth chapter is all about the iOpt toolkit and explains how problems and algorithms can be modelled with it.

The integration of the toolkit into RTBI is the main subject of the sixth chapter. This includes the conversion of the business-performance frameworks into the data-representation of iOpt and the selection of the appropriate algorithm. By the end of the chapter a first solution that uses iOpt for the optimisation will be presented.

The seventh chapter reviews the properties of optimisation problems and presents concepts for their solutions, with the main focus on metaheuristics.

The eighth chapter discusses improvements to the former solution, and shows how the quality of the results and the usability can be vastly improved.

The analysis of the different solutions is the topic of the ninth chapter. For the testing process, a business-performance framework of a server farm will be introduced and explained. The results from the different solutions will be compared.
2 Real Time Business Intelligence

In today’s highly competitive and dynamic environment, successful decisions require knowledge about the performance of your business at any time. It is important to react quickly upon missed targets, or even better, use the collected information to predict upcoming events and act pro-actively. Since business processes consist of different layers, targets are usually determined for every level. But how can the dependencies between strategic high-level objectives like customer satisfaction and operational targets like the number of employees be modelled? And how can the ideal settings for all elements of a process be found, while the targets are constantly influenced by customer demands and competitor strategies?

The aim of the RTBI system is to answer these questions by translating the processes into collections of entities, that represent every level of the business. Furthermore the interactions between these entities can be modelled manually or learned automatically to create business performance frameworks. Once the framework is defined, it can be used for real-time monitoring, testing of scenarios, predictions about the future and target optimisation. The following sections provide an overview for the concepts behind business performance frameworks and the underlying technical fundament.

2.1 Business Performance Frameworks

A business performance framework is an abstract view on a business process, from the viewpoint of performance measures. Figure 2.1 shows an example that is based on a call centre, that handles customer calls concerning fault repairs and provision processes. The colour indicates the measures layer and the number on the left displays its current value. The arrows display how the measures influence each other.
2.1.1 Determination of the Different Layers

Essentially all performance measures consist of an identifying name, an assigned value range and a target value. Distinctions are drawn to represent the different aspects of business processes. The RTBI platform divides them into four categories, that are explained in more detail in the following section.

Operational Levers

The operational levers are the lowest level of each framework and are the only levers that can be changed directly. In Figure 2.1 they enclose the different types of staff and the activities affecting them. In general every technical, human or social resource of the operating company is conceivable. All values of the above layers result from the setting of the operational levers and the external influences, so finding the best setting with respect to the aspired objectives is an important aspect.

Figure 2.1: An example business performance framework
External Influences

In Figure 2.1, the job rate and the call rate are the external influences. Their values can only be influenced with significant effort and an insecure outcome, e.g., by conducting advertising or marketing campaigns.

Key Performance Indicators

KPIs and the links model how the measures influence each other. For example, in figure 2.1, even a large number of technicians does not necessarily entail a low clear time and a high customer satisfaction, if the staff is incompetent or unmotivated. On the other hand, expanding their knowledge will obviously result in higher costs. The different ways of modelling the relationships are described in 2.2.1.

Strategic Objectives

The highest level consists of the strategic measures, that reflect the performance of the business. Two usual aspects of this layer are shown in figure 2.1: the customer satisfaction and the costs arisen. These measures usually evaluate the work of the management, so there will be a lot of effort in optimising them. But since they depend on so many influences this is a complex challenge, especially in large scale businesses.

2.2 Functions

2.2.1 Modelling and Updating Business Performance Frameworks

The basis for all other functions is the modelling of the performance frameworks. All defined performance measures are generic and can be used in different contexts to avoid redundancy in similar processes. In a simple example like in figure 2.1, the relationships are straightforward and could be modelled by simply using equations, but with rising complexity this approach becomes infeasible. To tackle this problem, the dependencies can also be learned from historic real-life-, or simulated data, which is used in conjunction with modern techniques of artificial intelligence and statistics (Neural Networks, Regression Trees, Linear Regression etc.). To keep up with the changes of the business environment, these learning
procedures can be conducted on a regular basis, e.g. when new results of a customer survey are available.

### 2.2.2 What-if-Analysis

To analyse how particular sets of values propagate throughout the framework, the what-if-analysis allows the user to set the values for the operational levers and external influences manually. The resulting values are computed based on the defined relationships and are displayed so that the impact on the higher levels of the business can be evaluated.

### 2.2.3 Real Time Monitoring

As shown in figure 2.2 freely configurable dashboards can be used to display the performance of the business compared to its targets. So far traffic lights, speedometers, conventional graphs and scorecards are supported to monitor the current value of any measure of the business framework. If available, the input data can be picked up and processed in real-time. For measures that have a lower sampling rate, like customer satisfaction, the current values can alternatively be derived from the learned relationships or even predicted from the external influences as elucidated in 2.2.4.

### 2.2.4 Forecast and Control

The business performance results from the setting of the operational levers and the external influences. Having a good ratio of both is the key to success. Since the operational levers, like the number of technicians, can not be changed instantly, it is desirable to foresee the upcoming values for the external influences and change the levers pro-actively. The RTBI-system provides this function by exploiting model predictive control.

### 2.2.5 Target Optimisation

Opposed to the bottom-up approach of the what-if-analysis, is the target optimisation. The task here is to find an optimal set of values for the operational levers, that achieves user defined strategic objectives, assuming knowledge about the external influences is given. This is not a trivial problem because the measures can not be optimised one by one, as that would disregard the global relationships. If, for example, every expense factor in figure 2.1...
would be minimised, the customer satisfaction would drop to zero. To deal with this global nonlinear optimisation problem, criteria have to be defined and an appropriate method has to be chosen. In the example above the usual aim would be to minimise costs as low as possible under a certain target and maximise the customer satisfaction at the same time.

Figure 2.2: Monitoring the current real life values in a dashboard

2.3 Architecture

The RTBI system is based on a client-server architecture that is divided into five layers (see figure 2.3) and resembles the popular MVC pattern.

- **Resource layer** - Contains different types of physical storage for the data access layer.
- **Data access layer** - Provides an interface between the resource layer and the business logic. The Data Access Objects (DAOs) hide the different ways the data is stored from the layer above.
• **Business logic layer** - Combines the data with their assigned business logic and methods by using stateless EJBs, that can be accessed by the control layer.

• **Control layer** - Provides servlets that redirect all requests from the presentation-, to the business logic layer and send the responses back.

• **Presentation layer** - Includes all GUIs for the user.

As a major difference to most servlet based distributed systems, the presentation layer does not use a web browser for the user interaction. Instead, every client executes a local application, which is based on the Java Swing toolkit and accesses the control layer via a class called `ServletCon`. This class forwards all requests to the appropriate servlets, that communicate to the server based section of the system. It is possible to attach objects to the request, to specify further options. A common practice is the addition of a `Properties` object (from the `java.util` package) to combine the parameters of a query.

The division of the layers is based on the PAC pattern, a distributed version of the MVC pattern, that separates the client view from the business logic and manages the communication between them. The presentation layer contains the view and the controller while the
abstraction layer is similar to the model in the MVC pattern and consists of the business and data access layer. The gateway between the two is the control layer, that implements the façade pattern to simplify the interface for the client side.

2.3.1 Communication Between the Layers

As stated above, all communication between client and server runs through the ServletCon class and its overloaded `getObject()` method, that always returns an instance of the type `Object`. From the developers point of view this is an important restriction for two reasons. First, all possible return types have to implement the `Serializable` interface or they can not be transferred. Second, the return types have to be known on the client and server side, what demands the use of shared libraries (in the simplest case the Java system library). Another aspect that arises from this approach is the special attention that has to be paid to the type safety, to avoid the occurring of `ClassCastException`
3 Requirement Analysis

3.1 Previous State of the Target Optimisation

Before the deployment of the new techniques, the target optimisation was already imple-mented in an interim way. This solution was solely intended for small demonstration frame-works that operate with a discretised solution space. So for the following chapters, the definition of performance measures in 2.1.1 is extended by a step size, to discretise the value range.

The following sections describe the way the previous implementation worked and what in-conveniences accompanied it. After this description, the requirements for the new solution are defined.

3.1.1 The Implementation

The target optimisation was done in the server-sided BfOptmisationSessionBean, a state-less EJB, and basically consisted of two steps. First, a database table was created and filled with all possible value combinations for all performance measures of the framework. Second, the table was sorted for all measures, ascending if their value should be maximised and vice versa. When the user executed a target optimisation, the session bean would perform a look-up for the row that has the lowest cost and update the framework with the appropriate values. If the table contained no combination of values that fulfilled all target values, a message got displayed and the framework remained unchanged.

3.1.2 The Problems

While the actual look-up was very fast, the solution suffered from multiple problems. The most significant one was the time-consuming creation of the table, especially when the framework got larger and/or the step sizes were small. Another aspect that resulted from it was
the static nature of the table, that required a repetition of the creation process, every time
the step size or the value range of a performance measure was altered. Furthermore the
static criterion cost presupposed that every framework contains that measure and prevented
the use of another objective, or multiple criteria.

3.2 Desired Functionality

The first aim of the reengineering is to solve the above mentioned problems to decouple
the solution from static resources, like a database table or a certain performance measure.
This entails the possibility of declaring objective functions, that can contain any number
of strategic objectives from the framework. To concatenate them, basic mathematical op-
erators, functions and additional constants have to be provided in an intuitive GUI. Each
objective function needs to be assigned to only one framework and it has to be possible to
save, load and edit them. Additionally, the user has to declare if the result of a function
should be minimised or maximised. Since iteration of every possible combination is not a
feasible approach, a dedicated optimisation algorithm has to be integrated into RTBI. This
requires the preceding extraction of the framework properties and their conversion into a
format the algorithm can work with, as the data representation is not suited for this task.
This transformation has to be done every time the optimisation is executed to enable rapid
alterations of the properties.

The optimisation has to be performed server-sided, including the update of the framework
values. That means that no data gets transferred, except for the client request and the
server notification if a valid solution is found. Furthermore it is desirable to encapsulate the
actual optimisation from the BfOptimisationSessionBean, for easier exchange, extension or
to allow parallel available implementations.
4 Implementation - First Steps

As discussed in the previous section, the former solution generally optimised frameworks by minimising the static criterion ‘cost’. So before any new ways of optimisation can be implemented, it is mandatory to know how objectives are defined. Therefore, providing the user a facility to catenate arbitrary strategic measures into objective functions is a necessary basis for the further development. But what kind of interface offers powerful possibilities but is user-friendly at same time? The following chapter gives an answer to this question and presents an implementation.

4.1 The Formula Editor

It is clear that an interface that is based on a widely used input scheme is easier to grasp than a custom built one. The handling of a pocket calculator for example is familiar with most users due to its widespread usage. The RTBI software already contains an editor that is based on this paradigm. It is intended to define the process of a KPI, by creating formulas that describe the correlation between the input measures and the output. All basic mathematical operators are supported, as well as functions, like sum or logarithms and three different types of brackets. This fits the requirements described in the last chapter and so extending and adapting the class would be a possible approach. Is it reasonable to create a dependency to the functionality of another class? In this case the answer is yes, because of the following aspects:

- More time for the development of the actual optimisation processes
- Reduction of redundant source code
- Easier changes of the general look and feel
- The editor is stable and faces no foreseeable changes

Essentially, the quality of new editor benefits from the solid foundation, while the expenditure of time for development and maintenance decreases.
4.1.1 The Graphical Interface

The editor, as seen in Figure 4.1, extends the JPanel-class and consists of various Swing components. The different types of elements that can be added are below the panel for naming and loading on the top (this concept is explained in 4.1.3; the drop down menu for selecting a table was disabled in that version). The panel on the bottom, displays all elements of the current formula and allows the user to move or delete them. Both of these panels are defined in the class MathViewPanel, that operates independently and gets accessed as an instance variable.

Figure 4.1: The formula editor in the adapted version
4.1.2 Input of the Properties

The names of the already defined formulas get retrieved in the class, by sending a request to the ServletCon class. The drop down menu “Variables” on the other hand is initialised by passing a Collection of Strings as a parameter to the constructor. In the parent class it contains the names of the inputs of the KPI. For the adapted version however, it has to hold the names of the operational levers. So the first necessary step is setting up a function to generate this Collection. This may seem like an easy task but unfortunately the data structure is not suitable. The reason for this lies in the concept of keeping the measures generic. That entails that the DAOs do not have an attribute field to reliably assign them to a layer. To add this function, several steps are required - steps that are similar for every extension of the function range. The following section explains them exemplarily.

Retrieving the Operational Levers

The first question is to implement it on the server or client side. In this case, the method is useful in various contexts, so it is reasonable to do the latter. Enabling this request to the client presupposes:

- adding the command to the appropriate servlet
- extending the interface of the EJB
- implementing the function

The processing of the command words works by comparing the input String with the possible commands. By adding a new branch to this if-statement, the work is done on the client-side, as the servlet class provides an instance of the framework’s EJB-interface. Of course, this requires the prior extension and the actual implementation.

Operational levers form the lowest level of the frameworks, hereafter referred to as “root-nodes”. They can be extracted from a list of all nodes by comparing their names with the inputs of every other node. Every node value-object contains a Collection of its inputs, nested iterating solves the task.

Now the client process can request the names of the root-nodes by sending the command "getrootnodes" to the ServletCon-object.
4.1.3 Event Handling

The event-handling is realised with the `ActionListener`-interface and comprises five events:

- **Open-button is pushed**: The button is renamed to “Create” and a drop down menu with the saved formulas replaces the text field
- **An item from this drop down menu is selected**: The formula is loaded into the editor, overwriting the prior settings
- **Create-button is pushed**: The button is renamed to “Open” and a text field replaces the drop down menu
- **OK-button is pushed**: The current formula gets persisted in the database
- **Cancel-button is pushed**: The dialogue is closed

These functions will be supported in the new version of the editor, so the `actionPerformed`-method does not have to be changed. The events for loading and saving formulas however, call dedicated methods that have to be adapted.

4.1.4 Extending The Class

To adapt the editor to the new task, a new class has to be created, that extends the current class. Now the key methods can be overridden. In this case, this comprises the methods for loading a formula, retrieving all formulas of the framework and adding a new formula to the database. To store the data, the easiest way would be to use the existing table for all formulas, but this raises the question how to distinguish the different types. One answer is the addition of a new column for this purpose.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>the name of the formula</td>
<td>varchar</td>
</tr>
<tr>
<td>table</td>
<td>unused legacy column</td>
<td>varchar</td>
</tr>
<tr>
<td>bfname</td>
<td>the name of the associated framework</td>
<td>varchar</td>
</tr>
<tr>
<td>formula</td>
<td>the formula as displayed</td>
<td>varchar</td>
</tr>
<tr>
<td>xml</td>
<td>the formula as a xml-string</td>
<td>varchar</td>
</tr>
<tr>
<td>type</td>
<td>the type of the formula</td>
<td>varchar</td>
</tr>
</tbody>
</table>

Table 4.1: The adapted "RtbiFormula" table

The disadvantage of this solution is the fact that the parent class can not remain unchanged any more. However, considering the clearer data structure and the time saved, this is
acceptable. Extending the table entails the adaptation of the associated EJBs and their value objects. Furthermore, the responsible servlet has to process the additional parameter of the request’s Properties object. By writing and executing a sql-script, the existing data sets can be converted to the new structure. It is important to specify the updates in a script, as there are different instances of the database, that have all to be changed, once the development is finished. To regain the former functionality of the parent class, the methods for storage and retrieval simply have to add the formula type as a parameter.

At this stage, the preparation is done and the editor can be implemented. To retrieve all formulas of a framework, a new command has to be added in the same way as described in 4.1.2.

The implementation in the EJB works by simply adding the following sql-command to the annotation of an abstract method, with the name of framework (bfname) as a parameter:

```plaintext
1 SELECT DISTINCT o.name FROM RtbiFormulaBean AS o
2 WHERE o.type = 'opt'
3 AND o.bfname = ?1;
```

If the user selects a formula out of the drop down menu, the methods of the parent class for the retrieval can be used. The same applies for the storage, since the distinction is made only by the given parameter value. To detect formulas that are no longer valid because of missing root-nodes, an additional validation is required. It is implemented in the framework’s session EJB and works by checking if every name of the used root-nodes is still in the framework. Clearly, it would be preferable to decouple the names from the nodes, by using identification numbers instead of names, but this would require a broad refactoring of the existing data structure.

### 4.1.5 Assigning Formulas to Frameworks

For the assignment of the formulas, a new column is needed in the table of the frameworks. The procedure is the same as above and the associated method gets called when the user confirms the settings by pressing the OK-Button.

The last aspect left is the distinction if a formula should minimise or maximise its result. Again, the solution is the extension of the framework’s table by another column, that describes the type. When the user assigns a formula, a small dialogue containing two radio buttons with the options “minimise” and “maximise” appears, to obtain the input.
4.1.6 Integrating the Editor into the GUI

To provide the new function to the user, the editor has to be integrated into the existing graphical interface. Therefore, an icon is added to the toolbar and linked to the ActionListener. To prohibit the invocation when the framework is empty or has no root-nodes, two verification tests have to be passed. If one of them fails, an appropriate message tells user about the problem and recommends steps for the solution.

4.2 Virtual Views for Optimisation

For testing purposes of the editor, the previous solution for the optimisation can be adapted by adding a virtual column to a view on the generated table described in 3.1.1. “A virtual column is a column in a query whose value is calculated from the value(s) of other column(s).” [2] Essentially, the defined formula is used as the sorting criterion of the view. This can be achieved by piecing together the following SQL command inside of the EJB BfOptimisationSessionBean with a StringBuffer:

```
CREATE OR REPLACE VIEW <framework name>_opt_view AS
  SELECT DISTINCT <allnodes>, <optimisation formula> AS opt_value
  FROM <framework name>_opt
  ORDER BY opt_value [ASC | DESC]
```

When the user executes a target optimisation, the top row of the view is selected and the values are set to the corresponding nodes. This already allows the analysis of the impact from different optimisation settings on the framework.

The solution is feasible, if the configuration of the framework does not get changed and the optimisation table can be generated in a timely fashion. However, the dependency on the time-consuming creation phase, inherits the problems discussed in 3.1.2 to this solution. Therefore, every modification still forces a rebuild of the table. The only real advantage is the elimination of the hard coded criterion. A non-static approach is preferable and one way to achieve this goal is presented in the next chapter.
5 The iOpt Toolkit

5.1 Abstract

Defining and implementing advanced solutions for optimisation problems is a highly complex challenge and requires expert knowledge and experience. The iOpt toolkit strives to remove these barriers, to provide modern technologies to a broader audience. The main intention is to support developers with a framework of ready-to-use components for problem modelling, search algorithms, scheduling and visualisation. It is entirely written in Java, utilises techniques of the of Artificial Intelligence based on heuristic search and employs the paradigm of declarative programming. All components can be configured for the existing requirements or customised with the methods of object orientated programming. The available constituents are shown in figure 5.1.

![Figure 5.1: The components of iOpt [8]](image)

For this project, the packages for problem modelling and heuristic searching are used for the
target optimisation. The following chapter describes the relevant concepts in this context in a general way and from a developers point of view.

## 5.2 Invariant Library

“The Invariant library provides built-in data types such as integer, real, boolean, string, object and also set versions of these types.” [8] Furthermore, it offers arithmetic and logical operators, to model the constraints and properties of a problem in a way that is close to the equivalent mathematical formulation. Hence, a problem model is basically a framework of concatenated and nested invariants that can be used for iOpt’s other functions. In this framework, an invariant acts as a lightweight constraint, that propagates value changes of its inputs. This makes them especially suitable for heuristic searching, as they allow a rapid evaluation of different solutions.

### Developers View

Invariants can roughly be divided into three different types in terms of restriction. If the value range should only be limited by the boundaries of the associated data type, it is called an expression. Their value can solely be changed by iOpt during execution, in contrast to the type variable, that remains constant until the user changes it. An expression with a determined value range within the bounds of the data type is named domain variable. Table 5.2 gives an overview of the different types, that are defined in individual classes for every supported data type. The order corresponds to their type hierarchy, with RealExp as the superclass.

<table>
<thead>
<tr>
<th>Type</th>
<th>Class (type real)</th>
<th>Function</th>
<th>Changed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>RealExp</td>
<td>represents data type</td>
<td>iOpt</td>
</tr>
<tr>
<td>Variable</td>
<td>RealVar</td>
<td>constant</td>
<td>user only</td>
</tr>
<tr>
<td>Domain Variable</td>
<td>RealDomainVar</td>
<td>defined value range</td>
<td>iOpt</td>
</tr>
</tbody>
</table>

Table 5.1: The different types of restriction

All expressions are available from one static class and can not get instantiated directly. Every individual type and every combination of different input values and return types is, however, defined in its own class. This minimises the source code in every class and simplifies the extension of the library. As some mathematical functions (power and logarithms) of the
formula editor weren’t implemented, they had to get integrated during this project. An example of an invariant that calculates the value of a real-expression to the power of a double is shown in the class `com.bt.iOpt.inv.RealExpPowRealRealExp` of iOpt.

### 5.3 Problem Modelling

The framework for problem modelling is dedicated to the definition of problems within iOpt and the management of the solutions. The representation follows the concept of constraint satisfaction problems (CSP) or constraint optimisation problems (COP), where the user specifies problems as a triplet \((X, D, C)\) for a CSP [3], or as a quadruple \((X, D, C, F)\) for a COP.

- **X**: is a set of decision variables \(X_1, \ldots X_n\)
- **D**: a collection of sets \(D_1, \ldots D_n\), where \(D_i\) defines the value domain for the \(i\)th decision variable
- **C**: a set of constraints defined over the decision variables to be satisfied
- **F**: an objective function defined over the decision variables providing fitness value for any variables’ allocation.

During the search process for best solution of a CSP, iOpt has to find a set of values for the decision variables, that satisfies all constraints. The COP additionally evaluates the solutions with the fitness function to further optimise the results, even when all conditions are already fulfilled. These aims are defined in the objective function of the problem, that has to be minimised or maximised.

### Developers View

Since the problem modelling is fully based on the invariant library, all necessary components are part of it. Decision variables are modelled with domain variables, containing one or multiple intervals. A constraint is represented as a numerical invariant expression, that returns a boolean invariant expression, corresponding to the compliance of the constraint. The objective function results from the sum of several invariant terms, with each term being a numerical invariant expression, returning another invariant expression of the type `real`.

New problem models are defined by extending the class `Problem` and defining the invariant framework. The direction for optimisation is passed to the constructor. Initialisation and
alteration of the framework’s settings has to be done inside of a transaction to have the possibility of a rollback if an error occurs. Therefore, these are all methods from the class `Problem` required for the definition:

- `public final void beginModelChanges()`
- `public final void addDecisionVariable(Variable v)`
- `public final void addConstraint(BooleanExp b)`
- `public final void addObjectiveFunction(RealExp r)`
- `public final void endModelChanges()`

This simplicity is a good example of the way iOpt uses declarative programming to reduce the expenditure of time for the developer. The next chapter will deal with the representation of solutions and shows how the RTBI-frameworks can be translated into iOpt problem models.

### 5.4 Algorithm Modelling

The heuristic search framework is designed for the development of advanced optimisation techniques, known a meta-heuristics. Due to its generic architecture, it supports a wide range of techniques. These include methods from manipulating a single solution like a local search to methods based on a population of solutions, for example genetic algorithms and evolutionary methods. The internal representation is based on the the assumption, that many heuristic algorithms have similar procedures and can be fragmented into constituent parts. Due to this partitioning, different methods are not independent from each other, as they can share a set of components.

#### 5.4.1 The Concept

Three concepts are the basis of the heuristic search framework: `heuristic search`, `heuristic solution` and `heuristic problem`. Heuristic search is used for defining search algorithms and heuristic solutions represent the optimisation problem within the search process. The connection of the optimisation problem and the search framework is done by the heuristic problem component, that acts as an interface between them.

Each search component has an assigned role and a requirement that has to be satisfied to ensure the correct behaviour. For example, an instance of a `SingleSolutionHeuristicSearch`
has the role of solving a heuristic problem with a single solution algorithm. The require-
ment is an instance of a Single Solution Method. Accordingly, a Single Solution Method has a
heuristic solution, as the input modifies it with its own method and returns the new heuristic
solution. Due to these dependencies, every valid heuristic search instance is a tree of search
components, which satisfies the requirements of all its leaves. Figure 5.4.1 shows the class
hierarchy of the necessary components for building the core of a local search method.

![Class hierarchy of a local search component](image)

**Figure 5.2: Class hierarchy of a local search component**

**Developers View**

The actual implementation is tied very closely to the concept. To define a search algorithm,
the first step is the creation of the required components and, if applicable, their configuration.
In the second step, the parts are assembled to the heuristic search, by nesting them into a tree
as shown in 5.4.1. Therefore, every component inherits methods to add other components
from an abstract super class. The execution of the search requires the problem model as a
parameter and runs for a defined period of time. When the process is finished, the heuristic
solution, or multiple solutions are available for examination from the search instance.
Further details about the actual implementation and appropriate methods for RTBI are
shown in the next chapter.
6 Integrating iOpt into RTBI

6.1 Motivation

The techniques for optimisation provided by iOpt enable the flexibility demanded in 3.2. Therefore, an integration of the toolkit into the current system is inevitable, since the interrelated architecture prohibits an extraction of specific parts. But how can the two data structures be connected in a way that satisfies current and future requirements? And is it possible that both sides, RTBI and iOpt, gain benefits from the development? This chapter shows how these targets can be achieved, with a minimal impact on both systems.

6.2 Integration

Since iOpt does not require a dedicated process to operate, it can be integrated by simply adding its jar file to the RTBI class path. With some minor changes in the scripts for building and execution, the library becomes available at runtime. But the usage of the problem modelling facility requires the data of the framework to be available in iOpt’s own format, the invariants (see 5.2).

6.2.1 Conversion of the Data Types

As the two data structures are very different, the task is not trivial. On the one hand there are the huge value objects of RTBI, filled with a lot of information that is not needed in this context, like actual instances of every input node. On the other hand there are the invariants, only representing one piece of information each. Hence, the first step for the conversion is the determination of the necessary data from the value objects and the selection of their invariant counterparts. So what is needed to represent the framework and its nodes in this context?
Apart from an identifier, the main characteristics contain the value range, the step size and the current value. To specify the aspired direction of the optimisation, knowledge about the target value of the node and if the value should get minimised or maximised is required. Finally, the layer of the node in the framework is necessary for the correct handling. In RTBI, this information is found in two different classes: `BeVO` to represent the node inside of the framework and `BeProcessVO`, to describe the process that can be assigned to a node. The distribution of the information over the two classes is shown in Table 6.1.

<table>
<thead>
<tr>
<th>Information</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td><code>BeVO</code></td>
</tr>
<tr>
<td>Process</td>
<td><code>BeVO</code></td>
</tr>
<tr>
<td>Value Range</td>
<td><code>BeProcessVO</code></td>
</tr>
<tr>
<td>Step Size</td>
<td><code>BeProcessVO</code></td>
</tr>
<tr>
<td>Direction</td>
<td><code>BeProcessVO</code></td>
</tr>
<tr>
<td>Current Value</td>
<td><code>BeProcessVO</code></td>
</tr>
<tr>
<td>Layer</td>
<td>(BeProcessVO)</td>
</tr>
</tbody>
</table>

Table 6.1: Distribution of the information

While this partitioning makes sense to keep the components generic, it is rather circuitous for the target optimisation. The main problems are twofold: First, two object instances filled with unneeded ballast for every node use more memory than necessary during processing. Second, even if hardware resources are not an object, the current representation doesn’t reflect the structure of the whole framework concisely.

The problems can be solved by introducing new value objects for the nodes, that only contain the required data and a class that assigns the nodes to their layer. A negative aspect of this approach is the need for a function to convert `BeVO`s and `BeProcessVO`s into the new representation, that has to be updated if the data structure of these classes get changed. But since changes on these classes are unlikely, the gain outweighs this dependency.

The following classes got created:

- **ThinBeVO**: This class, a regular JavaBean, contains the information shared by all nodes: name, value, value range, step size, target value and the direction (minimise or maximise). The purpose of this class is to represent root nodes and constants, and to act as the superclass for all specialised forms of nodes. During the development of this thesis, the only implemented type of process was using formulas. In the future advanced processes like neural networks or regression trees will be available as well (see 2.2.1).\footnote{Note that the information about the layer is only reliable, if the node is not a root-node, see 4.1.2.}
To add them, other specialised subclasses with the additional data can be added to the existing value objects.

- **FormulaBeVO**: The first subclass of the ThinBeVO extends the parent class with a `String` containing the formula.

- **FrameworkVO**: This class holds the framework’s name, the formula and direction for the optimisation. To represent the different layers, an array of ThinBeVO objects for each type contains the inclosed nodes. This means that no data for the assignment has to be saved inside of the value objects, as it is solely done in this class. Furthermore, it entails that the distinction of the different process types is done outside of this class, with the `instanceof`-operator.

In addition to the usual getters and setters, help methods provide an easy access to the nodes. Getting or setting the value of a specific node with them, relieve the user of searching through all nodes.

The class diagram of the value objects is shown in figure 6.1

![Class diagram of the new value objects](image)

**Figure 6.1: Class diagram of the new value objects**

To transfer the data from the old into the new representation, a method inside of the BfOptimisationSessionBean iterates over all nodes and sorts them into four Collections, depending on their type. After that, the FrameworkVO is created and filled with the data from the BfVO and their associated BeProcessVO. The last step is the retrieval and setting of the associated optimisation formula and the direction from the RtbiFormulaSessionBean, with the functions added in 4.1.5.

With this conversion, the handling of the data got simplified with a data structure, that could be used in various contexts. A useful side-effect is the possibility to manually specify
and test frameworks outside of the RTBI environment - a task that was very difficult before, caused by the large amount of required data.

### 6.2.2 BeVOs into Invariants

Converting the new value objects into their equivalent invariant representation, requires the selection of their adequate counterparts. Table 5.2 shows the different types of restrictions that invariants with a numerical value can have. The translation of the layers into these types is apparent, as shown in the table below.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Type</th>
<th>Class (type real)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operational Lever</td>
<td>Domain Variable</td>
<td>RealDomainVar</td>
</tr>
<tr>
<td>Strategic Objective</td>
<td>Expression</td>
<td>RealExp</td>
</tr>
<tr>
<td>KPIs</td>
<td>Expression</td>
<td>RealExp</td>
</tr>
<tr>
<td>Constant</td>
<td>Variable</td>
<td>RealVar</td>
</tr>
</tbody>
</table>

Table 6.2: RTBIs layers and their matching types of iOpt

This realisation entails that all nodes except the root nodes have no fixed value range in their associated invariant. This is acceptable, because it is the responsibility of the framework modeller to avoid that values resulting from the decision variables can exceed their bounds.

A task that is more difficult is the translation of the formula-Strings into invariants. In RTBI, a formula is represented in two ways: as displayed or with a XML-representation. An example for a simple term:

\[(3.14 + x^2)\] \hspace{8cm} (6.1)

This is the according XML-representation:

```xml
<TERM>
  <CONSTANT>3.14</CONSTANT>
  <OPERATOR>+</OPERATOR>
  <FUNCTION>
    <TYPE>POWER</TYPE>
    <VARIABLE>x</VARIABLE>
  </FUNCTION>
  <OPERATOR>,</OPERATOR>
  <CONSTANT>2</CONSTANT>
</TERM>
```
Defining this term with invariants requires some preparation. First, an expression for the variable ‘x’ is needed, which could be another term, a constant or a domain variable. Second, the elements of the term have to get defined separately and then nested into each other, as invariants usually only hold two arguments. Listing [6.1] shows the necessary code, with ‘x’ realised as a domain variable.

Listing 6.1: Defining a term with invariants

```java
/*
 * Defining an expression 'x' with initial value 6, a value range from 1 to 1679616 and a step size of 0.1
 */
String range = "[1->1679616]";
RealExp x = new RealDomainVar(6, range, 0.1);
/*
 * Defining and nesting the invariants
 */
RealExp power = Inv.pow(x, 2);
RealExp term = Inv.plus(3.14, power);
```

So how can an arbitrary number of nodes and processes be translated automatically? Obviously, processes that only rely on the inputs of root-nodes and constants have to get processed first and the required expressions can be generated easily according to table 6.2.2. Converting the formulas of the processes, requires parsing their Strings and putting all of their elements into one expression, that can be used as an input for other nodes. This has to be done with respect to the basic rules of mathematics, like the different operator priorities and that the resolution starts with the deep nested terms. There are two ways to realise the latter aspect: with iterating, or by using recursion. Recursion has the advantage that the algorithm is similar to one that solves only a single term, that entails a simpler implementation. Therefore, the processing starts with the first term found and solves it until another term is found. Then the method is called again with the inner term as argument and returns the according expression that gets handled like any other expression. Of course, there are more things to consider but this the high-level approach:

1. Remove the term indicators (brackets or tags)
2. Check the type of the next element, if it is a term, pass the inner term to 1.
3. Add the next element to a list
4. If the tag still has element go to 2.
5. Convert all function-, multiply-, and divide-elements of the list
6. Convert all addition-, and subtraction elements of the list

7. Put everything into one invariant and return it

A generic implementation of this algorithm, enables its application for different representations of the elements or rules of translation. This requires a decoupling of the definitions and the rules for converting from the algorithm. The realised architecture is based on the usage of XML-tags and is shown in figure 6.2.

![Class diagram of the classes for conversion](image)

**Figure 6.2: Class diagram of the classes for conversion**

One advantage of this approach is that the algorithm can support every tag representation, when an appropriate implementation of the `DefinitionInterface` is available. But even the mapping of the tags to their invariants can be changed, with an adapted implementation of the `ConvertInterface`. The caller of the method has to pass one instance of both to the constructor of the converter, which makes him the only class with knowledge about the chosen context. This makes the converter versatile and enables users and developers of iOpt to use it outside of this project.

The `ConvertInterface` contains the `getExpression`-method, that is overloaded to allow all combinations of expressions and constants. The arguments given inside of a wrapper that is necessary for the processing.
The `DefinitionInterface` implementation contains all supported tags, regular expressions for the tags, operators and functions as private constants. The regular expressions for the tags are formed like this:

`<[tag-name]>.*` 

They are used to identify the next tag, by comparing the remaining `String` with its `matches` (`String regex`) method to them. Literally, the comparisons to the expression asks: “Does the given `String` match this specific tag, followed by none or any number of other characters?”.

The converter has three public methods:

1. `RealExp stringToRealExp(String s)`: Main method for the conversion
2. `void setExpressions(RealExp[] inVars)`: Sets the available expressions
3. `RealExp stringToRealExp(String s, RealExp[] inVars)`: Combines 1. and 2.

An internal array holds the available expressions to replace their identifiers in a formula with them. The main method starts with the creation of an `ArrayList`, that gets filled with all elements of the term during the processing. Then the iteration begins and the given `String` is compared to the regular expressions. When a term is found, the process has to find the end tag, call the method again with this substring and add the result to the list. If an inner term contains other terms, the right ending is found by the `getTagIndex`-method, that increments a variable with the value 1 for every opening tag, and decrements it for a closing tag, until the value becomes zero. Operators are extracted by using the `substring`-method of the `String` class, with the length of the opening tag as the start index, and the first occurrence of the closing tag as the end index. Constants are handled in the same way, but they have to get parsed with the class `Double`. If a variable is found it gets replaced by its `RealExp`, unless the internal array does not contain it and an `ExpressionNotFoundException` is thrown. A special case is the handling of function elements, as they consist of other elements and have different numbers of arguments. Their sequence is built like this:

```
<FUNCTION>
  <TYPE>MIN2 | MAX2 | POWER | EXP | LN | LOG</TYPE>
  <CONSTANT>(argument)</CONSTANT>
  | <VARIABLE>(argument)</VARIABLE>
  | <TERM>(term)</TERM>
  [<OPERATOR>,</OPERATOR>
  & <CONSTANT>(argument)</CONSTANT>
  | <VARIABLE>(argument)</VARIABLE>
  | <TERM>(term)</TERM>]
</FUNCTION>
```
A definition of the type is followed by one element, or by two elements separated by a comma operator. The private method `proceedFunction` is responsible for this conversion and starts by cutting out the operator. When the arguments contain no other functions, the arguments can be separated into an array with the `split`-method of the class `String`. Otherwise, the comma operator is determined by iterating over all elements. To avoid a wrong identification caused by nesting, the process skips all formulas and terms during the search by moving the start index of the comparison behind them. Afterwards, the arguments of the array are put inside a `List` - either directly for a constant or by calling the `stringToRealExp`-method for sub terms. Then the arguments are passed to the `getExp(Object, Object, Operator)`-method that identifies their type, calls the `ConvertInterface` and returns the `RealExp`. Functions with only one argument are put in an expression that multiplies them with an identity element\(^1\).

Back to the `stringToRealExp`-method, the next step is to actually convert the collected arguments. This is done in the method `proceed()`, that searches the list for operators, starting with the highest priority and converts the two belonging arguments. The process is repeated until the lowest priority for operators is reached and the list contains only one expression. This example shows the procedure for the elements of the term \((3 + x * y - 2.2)\):

1. \{`Const(3)`, `Op(+)`, `Exp(x)` , `Op(*)`, `Exp(y)` , `Op(-)` , `Const(2.2)`\}  
   \[\rightarrow e_1 = x * y\]

2. \{`Const(3)`, `Op(+)`, \texttt{RealExp(e1)} , `Op(-)` , `Const(2.2)`\}  
   \[\rightarrow e_2 = 3 + e_1\]

3. \{\texttt{RealExp(e2)} , `Op(-)` , `Const(2.2)`\}  
   \[\rightarrow e_3 = e_2 - 2.2\]

4. \{\texttt{RealExp(e3)}\}

### 6.2.3 Defining the Problem Model

When all nodes are collected in a `List` inside of a subclass of iOpt’s `Problem`-class, the problem model can be defined, as described in 5.3. All expressions of the root-nodes are added as decision variables and then the objective function can be generated. Therefore, the function defined with the editor of 4.1 gets converted to invariants and is added with the `addObjective(RealExp)`-method. This means that no constants, KPIs or strategic

\(^1\)An identity element is a constant with the value 1
objectives are added directly - they can only influence the model via the objective. Essentially the problem model in this context consists of a set of decision variables plus an objective function, that has a value depending on a subset of all nodes of the framework.

### 6.2.4 Access to Optimisation

The next question is how the optimisation techniques should be accessed. Putting them directly into the optimisation EJB is not good practice, as this would result in a large and incomprehensible class. Encapsulating the methods into a dedicated class is a better approach because it separates the preparation from the execution and enables independent maintenance of the two parts. To gain further encapsulation, the optimisation should get requested by calling a single method that has a FrameworkVO as a parameter and returns it optimised. The usage of an interface matches these requirements and even exceeds them because it hides the actual implementation and allows the exchange of them. If, for example, another method should be available parallel to iOpt, one decision statement in the EJB about the used implementation would be sufficient to integrate it.

#### Exception Handling

Errors during optimisation can occur for multiple reasons and it is the duty of the implementation to detect and interpret them. Two different exceptions are used to apprise the calling class about the false behaviour:

- **FrameworkNotValidException**: When the FrameworkVO can not get solved by the optimisation because of missing or false nodes. This is usually thrown when a process references a node that is not in the framework.

- **FormulaNotValidException**: This is thrown when the FrameworkVO contains a FormulaBeVO with a misformed formula.

For the scope of this thesis these exceptions are enough, but with the addition of the new process-types, the range of exceptions should be extended accordingly.
7 Optimisation Algorithms

7.1 Combinatoric Optimisation

In general, optimisation is used for the determination of the best solution for an optimisation problem. "A combinatoric optimisation problem is characterised by a finite set $X$ of solutions and the target function $f$, that assigns a real number to every feasible solution." [6]. An optimal solution for a minimisation problem is a valid solution $s^* \in X$, that satisfies:

$$f(s^*) = \min_{s \in X} f(s)$$ (7.1)

A common example is the calculation of the coordinates of global or local minima or maxima from one-dimensional functions by finding the zeros of its derivation. Problems of the real world however, usually depend on a larger number of input parameters and are classified as NP-hard. This means that an exact algorithm like the examination of every possible combination of input values is not a reasonable approach, because the determination of the optimal solution can not be finished in an acceptable period of time. Therefore, the aim of existing heuristic optimisation techniques is to find a good solution that could be suboptimal but requires less effort. Heuristics are distinguished into two categories: heuristics with a dedicated algorithm that is developed for only one problem are called specific and are opposed to the general heuristics, that can be applied in various contexts. The latter only defines how the heuristic should be created and leaves the selection of a particular method for the solution to the developer. Due to this high-level concept, they are called metaheuristics. Obviously, they usually can not compete with the customised solutions, but they are useful when the development of a specific algorithm is impractical.

Common metaheuristics are: Hill climbing, genetic algorithms, simulated annealing and tabu search. Since evaluating and comparing all of them would exceed the scope of this thesis, this chapter will only discuss simulated annealing in detail, as it provides good results for the existing framework.

---

1NP = nondeterministic polynomial - a classification from computer sciences’ theory of complexity, cp. [4]
7.2 Heuristic Optimisation

The general concept of heuristic optimisation is the iterative improvement of an initial solution. However, because of the vast search space of most optimisation problems, an examination of possible alterations can only be done locally. To improve the result, the solution is translated into another state by performing an elemental transformation, like exchanging two locations on the route of a travelling salesman problem [6]. These modifications are also called moves and all solutions that can be reached from the current state with only one move make up the neighbourhood of the solution.

7.2.1 The General-Descent-Method

Based on this model, a simple approach called General-Descent-Method (GeDe) solves problems by comparing the results from all neighbourhood-solutions to an initial, random solution. If a better result is found, the old solution is replaced and the new neighbourhood gets examined. This process is repeated until no further improvement can be found. The disadvantage of this approach is the fact, that only local minima or maxima can be found and that those could differ very much from the global ones. Essentially, the quality of the result depends on the selection of the starting point and is thereby random as well. Since there are no methods to determine a favourable starting point, the GeDe has to be extended with the option to move out of a local minima or maxima again, with the target of finding a better solution.

7.2.2 Simulated Annealing

Simulated Annealing (SA) is an improved version the GeDe, that uses the same simple core, but provides results that are mostly independent from the selected starting point.

The origin of the technique lies in statistical mechanics, a branch of thermodynamics that deals with microscopic properties of physical systems. This is important because the algorithm is based on the behaviour of atoms in a piece of metal, that cools down again after it got heated to a high temperature. The interesting phenomenon is that after the original fence structure of the atoms are destroyed by the heating, the atoms order themselves again into an improved state of minimal energy, when they anneal slowly. This state of minimal energy is also called the solid state and results in a better quality of the metal, cp. [9].
The SA-algorithm connects the elements of this process to the properties of optimisation problems as follows:

- The *variables* of the problem are analogue to the atoms.
- A *valid solution* matches a solid state of the material.
- The *target function* defines the temperature.
- The *search* for the optimal solution equals the cooling process.

Like the GeDe-method, the algorithm begins with a random solution $s$ and an initially high temperature $T_0$. By performing a local modification, a neighbourhood solution $s^*$ is generated and compared to the current solution by calculating their difference.

$$\Delta f = f(s^*) - f(s)$$  \hspace{1cm} (7.2)

To decide if the new solution should replace the current one, a random number from the interval $[0, 1)$ is compared with the value of a probability mass function. This function has to have the property of being non-zero even if $s^* > s$ is satisfied, to allow leaving a local minimum again. On the other hand, if $T$ comes close to zero and $s^* > s$, the probability has to be very close to zero as well. This behaviour represents the physical process described, as a high temperature encourages the exploration of broad regions - similar the movement of the atoms, that increases parallel to the temperature - while a low temperature navigates the algorithm towards the local minimum.

The function can differ between implementations, but since they all share the characteristics above, they are similar to the original version, that gets 1 if $s^* < s$. The function below calculates its value otherwise:

$$P(\Delta f, T) = e^{-\Delta f / T}$$  \hspace{1cm} (7.3)

Hence, improving moves are always performed, while the execution of impairing moves depends on the temperature. This procedure can be repeated several times for every temperature, depending on the settings made. Afterwards, $T$ is lowered and the whole process is repeated until the temperature is close to zero or until another stop criterion, like a time or a move limit, is reached.
The whole procedure is shown below:

1. **Initialisation:**
   - IterationCounter $k = 0$
   - MaxIterations $k_{\text{max}}$
   - EvaluationCounter $l = 0$
   - MaxEvaluationsPerStep $l_{\text{max}}$
   - Temperature $T$
   - MinTemperature $T_{\text{min}}$
   - StartingPoint $s$
   - CurrentValue $v_1 = f(s)$

2. **Selecting a New State:**
   - create a neighbourhood solution: $s^*$
   - calculate its value: $v_2 = f(s^*)$
   - compare the values: $\Delta f = v_2 - v_1$
   - assign $s^*$ to $s$ if:
     - $\Delta f < 0$
     - or
     - $e^{-\Delta f/T} \geq \text{random}([0, 1])$
   - go to 2. if: $l < l_{\text{max}}$
   - increment counter $k = k + 1$
   - reset evaluation counter: $l = 0$

3. **Lower the Temperature**
   - assign new temperature $T = T_{\text{new}}$ with $T_{\text{new}} < T$

4. **Stop Criterion**
   - go to 2. if:
     - $k < k_{\text{max}}$
     - $T \geq T_{\text{min}}$

Note that if the result should be maximised instead, the sign of $\Delta f$ has to be made negative in the decision function.
To adjust SA to an actual problem, values for initial and final temperature have to be chosen. Furthermore, a function that generates the next temperature step has to be defined. These attributes are summarised in the so-called annealing schedule, that should be customised for every problem. Another important aspect is the handling of the previous solutions. Since the process, as described above, forgets the best solution found so far whenever a new solution is selected, it is reasonable to extend it, so that the best solution is saved during the search. Altogether, SA is an algorithm that provides very good results with an easy implementation, if the parameters are set correctly. Therefore, it is suitable for the first realisation of optimisation based on iOpt.

7.3 Implementation in iOpt

This section will show how Simulated Annealing can be used for the optimisation of RTBI’s business performance frameworks. As the problem model already available (see 6.2.3) can be used in conjunction with every search component of iOpt, it does not need any adaptation. Therefore, the next task is the assembling and configuration of the search.

7.3.1 Simulated Annealing in iOpt

The simulated annealing algorithm is a component of the iOpt-toolkit that belongs to the group of neighbourhood searches, shown in figure 5.4.1. It is a subclass of the abstract ThresholdneighbourhoodSearch class, that is used for a family of neighbourhood searches, that consider a move accepted if a specific condition (the threshold) has been satisfied. In case of SA, this condition matches the requirements of 7.2.2.

Internally, all parameters are defined as instance variables that can be accessed via get-, and set-methods. The probability function is identical to the original function 7.3 and the cooling schedule is realised by subtracting an offset value of the current temperature. Hence, the only necessary steps for the initialisation is the constructor call, the setting of the neighbourhood and the adjustment of the parameters.

7.3.2 Setting up the Neighbourhood

The setup of a neighbourhood in iOpt consists of three steps. First, an instance of the abstract Neighborhood-class has to be created, by calling the constructor of one of their
subclasses. Second, a decision variable selector has to be added to the neighbourhood, that
decides which variable should be altered next. Third, a value selector has to be added, that
controls the way the variables are altered. The selectors can operate by iterating through
the array of all values/variables, by processing them in order via the `Comparable`-interface
or by random.

The code below shows how the neighbourhood in this project is created:

```java
Neighborhood myNeighborhood =
    new AssignMoveDecisionVariableValueneighbourhood();
DecisionVariableSelector decisionVariableSelector =
    new DecisionVariableSelector();
decisionVariableSelector.setMovesVersion(Selector.RANDOMORDER);
SetDecisionVariableValueSelector valueSelector =
    new SetDecisionVariableValueSelector();
valueSelector.setMovesVersion(Selector.RANDOMORDER);
myNeighborhood.add(decisionVariableSelector);
myNeighborhood.add(valueSelector);
```

### 7.3.3 Implementing the Search

To realise the search, the tree of its components has to be created, as described in 5.4.1.
At first, the algorithm has to be instantiated and then the neighbourhood is assigned to
it, with the typical method of iOpt: by adding the component to it. The generation of an
initial solution for the heuristic problem is done in a dedicated subclass, of `Generation-
SingleSolutionMethod`. The alteration of the local environment of an solution is done with
the abstract search component `LocalSearch`. The role of this component is to use a given
heuristic solution to generate a new heuristic solution resulting from its own treatment. As
the simulated annealing instance should be used for this processing, it is applied. Further-
more, the local search contains the information about maximum number of performed or
evaluated moves, that are defined with the appropriate set-methods.

To execute the initial generation and the actual search in sequence, they are added to a
`CompositeSingleSolutionMethod`, that starts processes in order they are added to it. This
component is added to the root of the tree, a `SingleSolutionHeuristicSearch`, that acts
as the interface of the whole search for the user. Figure 7.1 shows the dependencies of the
elements and how the problem is processed until its solution is returned. This is the flow of
the process:

1. generate initial solution \( s \) (`GenerationSingleSolutionMethod`)
2. generate a set of potential moves \( M(s) \) using the neighbourhood instance (see 7.3.2)
3. select the best move \( m \) in \( M(s) \) (SimulatedAnnealing)

4. perform move \( m \) on \( s \) (LocalSearch)

5. while stopping condition not satisfied, go back to 2. (LocalSearch)

Hence, in contrast to the concept described in 7.1, a neighbourhood in iOpt is a collection of possible moves instead of actual solutions and search algorithm evaluates the moves instead of performing them. This provides the advantage that different types of neighbourhoods and searches can be combined freely.

![Diagram](image)

Figure 7.1: The data flow during the search process

In the actual implementation shown below, the possible number of iterations is bounded by composite single solution search and a ten second time limit.

```java
// generator for initial solutions based on the settings of the problem model
GenerationSingleSolutionMethod sGen = new UsingCurrentProblemGeneration();

// init the search and add the neighborhood
NeighborhoodSearch sa = new SimulatedAnnealing();
sa.setMaxEvaluatedMoves(Long.MAX_VALUE);
sa.add(myneighborhood);

// init the local search add the algorithm and set the stop criteria
LocalSearch ls = new SingleLocalSearch();
ls.add(sa);
ls.setMaxPerformedMoves(Long.MAX_VALUE);
```
Regarding the selection of the classes: During the development, the only finished demonstration framework that could be used for testing had a rather small search space. Therefore, the selection of the classes for the neighbourhood and the generation of the initial solution did not affect the results. With the upcoming complexer frameworks, an evaluation of the chosen classes and parameters could increase the quality of the results.

### 7.4 Executing the Search

With the problem model and the search available, the optimisation can be executed with iOpt. Therefore, the class of the problem model is extended with the code from the above sections and a search()-method, that creates and executes the procedure. Afterwards, the best solution found during execution is used to create an HeuristicSolution instance, that gets assigned to the problem model with its connect()-method. To update the framework, the names and values of all invariants with the type or super type RealExp² are passed to the updateThinBEVO(String name, double value)-method of the FrameworkVO-class. The caller of the optimisation interface can use this value object to update the framework by changing RTBI’s value objects, and therewith the database and the graphical representation. In the case that no result could be found that satisfies the target values of the strategic objectives, a messages asks the user to change the settings. The results are presented in chapter 9.

²iOpt creates other invariants necessary for execution that doesn’t belong to nodes of the framework.
8 The Formula Wizard

8.1 Motivation

The solution presented so far can be used to optimise frameworks according to a function that consists of strategic objectives connected with mathematic operators and functions. This is suitable for frameworks with a moderate number of leaf nodes and objective functions representing simple targets, like the minimisation of the cost with respect to customer satisfaction. However, rising complexity and the formulation of more sophisticated targets can not be handled in a user-friendly way. The main problem is that creating large functions and comprehending their impact on the optimisation process is a task that needs a lot of experience. Furthermore it is not intuitive, limits the possible range of users and prohibits rapid testing of different targets. Another problem is the weighting of objectives, that has to be done manually. If, for example, two objectives should be equally important, but have a different value range, one of them has to be multiplied or divided with a constant to even it out. This gets even worse with value ranges that do not begin at zero, as it requires additional shifting with addition or subtraction. Doing this for multiple nodes is a task that most users will experience as unpleasant and time consuming. All this leads to the demand of an entirely new user interface, to resolve these problems.

This chapter presents the concept and the realisation of a formula wizard, that allows the definition of target functions without any mathematical knowledge.

8.2 Requirements

The formula wizard has to provide enough options to the user to define a wide selection of different targets, while remaining easy to understand. Furthermore, the usability should not be influenced by the number of strategic objectives and their different value ranges. It has to be possible to disable or configure every node individually, but the correlation between them should be visible as well.
The targets should be defined in a literal way to hide the underlying mathematical details and provide four parameters for each node:

1. The \textit{target value} for the optimisation that can differ from the one in the related process.
2. The \textit{strength} of the constraint for the target value.
3. The \textit{importance} of the node, to represent the weighing.
4. The \textit{affinity}, indicating if the value of a node should stay next to its target.

This means that the optimisation should keep the values of the strategic objectives below (or above for maximisation) their target values and then optimise them further when the affinity is low. If the targets are not hit, a penalty should be imposed, according to selected strength value. The priority of the different nodes is defined by their importance. To set these values, the interface has to provide appropriate sliders for each strategic objective.

To simplify the handling, there should be only one saved setting for each framework, that is applied to the sliders when the dialogue is opened. Therefore, the settings have to be automatically assigned when they are saved to the database. Hence, the applied way of optimisation depends on the editor last used.

\section*{8.3 Realisation}

This section shows how the interface of the wizard is implemented and how the settings can be realised in iOpt. The components are arranged following the model-view-controller (MVC) pattern.

\subsection*{8.3.1 The View}

The GUI as shown in Figure 8.1 is fully implemented with the Java Swing toolkit and consists of a \texttt{JDialog} with a \texttt{FormulaWizardPanel} as the content pane, that encloses all other components. The top panel contains the names of the different parameters as \texttt{JLabels} and is not influenced by the scrollable area below. This area holds a control panel defined in the class \texttt{SliderPanel} with four sliders for each strategic objective. Changes to the settings of the sliders do not have to be processed until the formula is saved, as they do not influence anything directly. An exception is the slider that represents the target value in the first column, as it has a text field containing the current value that allows manual inputs. Additionally, a coloured bar shows the proportion of the value range accepted as a
valid solution, depending on the direction in which the value should be optimised. The slider and the bar are encapsulated in the class `ColorSlider`, that handles their connection with its implemented `ChangeListener`-interface. Every time the slider is moved, the `stateChanged(ChangeEvent)`-method is called and updates the internal `ColorBar`-class that contains two coloured panels, divided at a given parameter.

![Figure 8.1: The GUI of the formula wizard](image)

Mapping the settings of the slider to text field and vice versa requires two steps: First, the `SliderPanel` has to implement the `ActionListener`-interface and add itself to the listeners of the textfield. Second, the custom `ColorSliderListener`-interface has to be implemented and the class has to call the `register(ColorSliderListener)`-method of its `ColorSlider`-instance. When its value is changed, it iterates over all registered classes (currently only one), casts them as `ColorSliderListener`s and calls their only method called `stateChanged()`. Then, the realisation of this method updates the text field. This concept provides the advantage that the colour slider can operate independently, without having a dependency of being assigned to-, or being inside of another class, that allows its reuse.

When a value \( v \) is entered into the text field, the `actionPerformed(ActionEvent)`-method is called, and the colour slider is set accordingly. Therefore, the value of one percent of the
value range called $a$ has to be calculated, as the slider has a fixed range from 1 to 100:

$$a = \frac{100}{(v_{\text{max}} - v_{\text{min}})} \quad (8.1)$$

Afterwards, the percental value $p$ is calculated, with respect to the possible offset:

$$p = v \times a - v_{\text{min}} \times a \quad (8.2)$$

Hence, the current value $v_{\text{cur}}$ is calculated from the slider position $s$ like this:

$$v_{\text{cur}} = v_{\text{min}} + s \times \left(\frac{(v_{\text{max}} - v_{\text{min}})}{100}\right) \quad (8.3)$$

To avoid invalid entries into the value text field, a class called `DoubleTextField` extends the regular `JTextField` and overrides its `processKeyEvent(KeyEvent)`-method. If a non-numerical character is entered, it gets blocked, except for four cases:

1. The cursor is at the position 0 and the entered character is a minus (‘-’).
2. The cursor not at position 0, the character is an ‘e’ or an ‘E’ and is not contained.
3. The cursor not at position 0 but before an ‘e’ or an ‘E’ and the character is a minus.
4. The character is a dot (‘.’) and the text does not contain it.

These exceptions are necessary to enable the input of exponential representations like $-19.2e - 10$, that are important for small value ranges. If the entered value exceeds the value range, the slider is set to the minimum or maximum.

A final aspect of the text field is that the `actionPerformed(ActionEvent)`-method is only called when the input is finished by pressing the return-button. To enable entries followed by a mouse-click on another component, the `SliderPanel`-class has to implement the `FocusListener`-interface as well and update the slider when the focus is lost from the text field.

When an objective should not be used for the optimisation, its panel can be disabled with the check box in the lower right, like the third panel in figure 8.1.

The UML-diagram of the main classes of the GUI is shown in figure 8.2.
8.3.2 The Data Model

All values are stored inside the class `SliderVO`, that contains the name of the associated strategic objective, its minimum-, maximum-, and target value, the direction, if the panel is enabled and the settings of all sliders. It can be passed to the constructor to initialise the panel with specific settings and can be requested from other classes to process the settings.

8.3.3 The Controller

The GUI is created and controlled by the class `WizardControl`, that is instantiated with an array of the `SliderVO` objects for the initialisation of the `SliderPanel` and `ServletCon` to save and assign settings. It implements the `ActionListener`-interface and adds itself to the confirm button of the GUI, to be notified when the user input is finished.
The settings are saved to the formula table with this syntax:

\[(<id>[<name>%<target-value>%<strength-val>%<affinity-val>%<importance-val.>]])\]

These settings Strings are assembled with a StringBuffer and translated back into SliderVOs with a StringTokenizer and the parseDouble(String)-method of the class Double. The identifier in the beginning is used to distinguish them from the regular formulas. As the control opens the wizard as dialogue, the rest of the application is blocked while it is opened.

### 8.4 Realisation with iOpt

This section presents a solution for optimising RTBI’s frameworks with the settings made by the wizard.

As the settings are treated like the regular formulas, no adjustment of the BfOptimisationSessionBean is necessary. Instead, the implementation of the optimisation interface detects the type by comparing the first characters of the formula with the identifier. If the identifier is found, the alternative objective is generated and added to the problem model.

#### 8.4.1 Linear Function Objectives

The requirement for this objective function is that it realises the wizard’s settings, while unifying the different value ranges. Since the quality of a solution is evaluated from its value from the objective function, a linear penalty function can guide the algorithm in the desired direction. To significantly increase the penalty when a value exceeds its target, a step has to be added. By connecting it to the strength value of the wizard, the user can influence its height, and thereby the target priority of the different node.

![Figure 8.3: The penalty step function](image-url)
The gradient of the section before the step can influence if the value should be just below or above the target, or if it should get optimised further. As the algorithm will always try to find the best solution, just setting the slope to zero will not keep the value close to the target. To realise the affinity, the gradient has to be negative, so that moving away from the target will result in a higher total cost.

The whole cost function with a strength $s$, a maximum value $e$, a slope that reaches the height $a$ before, and the height $b$ after the target, is shown in figure 8.4.

![Figure 8.4: The penalty step function with all parameters](image)

Hence, that the penalty $p$ for a value $x$ can be calculated with this formula:

$$p(x) = \begin{cases} 
  x < t : & a * x \\
  x \geq t : & \frac{(b-s)}{(e-t)} * (x - t) + s 
\end{cases}$$  \hspace{1cm} (8.4)$$

The only parameter left from the wizard is the importance. Since the entire objective function is the sum of all penalty functions, a factor is used to amplify them according to their importance $i$.

This means that an objective function of a framework with $N$ strategic objectives is built like this:

$$o(x) = \sum_{k=1}^{N} i * p_k(x)$$  \hspace{1cm} (8.5)$$

To ensure that all penalty functions have similar value ranges, every penalty function has to be multiplied with a factor $f$ depending on the highest maximum value. When a node does not have the largest value, this factor is calculated by dividing the largest value through its own maximum value.
8.4.2 The Implementation

The penalty functions can be realised with the invariant CompositeLinearFunction, that can assign any number of gradients to specified value ranges of an expression. The constructor requires four arguments: the associated real expression, an array of value domains, an array with the penalty bases and an array of slopes. In this context there are two value ranges: pre-, and post-target. They are represented with the implementation RealIntervalsDomain of the abstract class RealDomain, that gets instantiated with a String for the range, like the RealDomainVar in 6.1. The penalty base is set to 0 before the target and equals the strength value from the slider multiplied with 10000 plus 500 afterward. As the slope $s$ before the target depends on the affinity value $a$, it gets calculated with this formula:

$$s = 2 - a/25 \quad (8.6)$$

The slider has a value range from 1 to 100, so the maximum setting for affinity equals a slope of $-2$, while below values 50 result in a positive gradient. After the step, the slope is set to 200.

To handle nodes that should be maximised, the slopes and the penalty bases are changed, so that moving towards the maximum value will decrease the value of the function.

With these settings, the objective can be generated by iterating over all strategic objectives, creating their linear function and adding them, if necessary normalised, to the problem model. This means that in contrast to the previous solution, the cost function does not result directly from the value of the nodes, as the penalty functions are layered on top of them.

---

1 These values provided the best results during the testing of the framework described in chapter 9.
9 The Results

This chapter introduces a demonstration framework and compares the optimisation results from the different techniques.

9.1 Framework Server Farm

This framework is based on the business process of a serverfarm with a scalable infrastructure, that is used to handle customer requests for websites. The RTBI graph of the components is shown in figure 9.1 with the colours indicating the layers. All processes in the framework are realised with formulas.

Figure 9.1: The framework ‘Server Farm’
9.1.1 Operational Levers

The number of operating servers is specified with the measure Server and they are connected to a network with a specific Bandwidth. A high value for both of these factors results in high customer satisfaction, but also in high costs. The Page size describes the amount of content served and should be maximised.

<table>
<thead>
<tr>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Step</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>1</td>
<td>17</td>
<td>1</td>
<td>minimise</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>5000</td>
<td>20000</td>
<td>100</td>
<td>minimise</td>
</tr>
<tr>
<td>Pagesize</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>maximise</td>
</tr>
</tbody>
</table>

9.1.2 External Influences

The amount of necessary resources for achieving a high customer satisfaction, depends on the number of user Requests, the only external influences of the framework. For the testing it is set to its default value 210.

9.1.3 KPIs

The KPIs of the framework have the purpose of translating the value of the strategic objectives and external influences into the dissatisfaction. For example, a satisfying time to transfer the requested data to the customer demands sufficient settings for server and bandwidth according to the pagesize. The other dependencies are shown in figure 9.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Step</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransferTime</td>
<td>0</td>
<td>50</td>
<td>1</td>
<td>minimise</td>
</tr>
<tr>
<td>Timeouts</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>minimise</td>
</tr>
<tr>
<td>ResponseTime</td>
<td>0</td>
<td>50</td>
<td>1</td>
<td>minimise</td>
</tr>
<tr>
<td>TF</td>
<td>0</td>
<td>1066500</td>
<td>10</td>
<td>maximise</td>
</tr>
<tr>
<td>OverflowRate</td>
<td>0</td>
<td>100</td>
<td>10</td>
<td>minimise</td>
</tr>
<tr>
<td>ServRate</td>
<td>0</td>
<td>1000</td>
<td>100</td>
<td>maximise</td>
</tr>
</tbody>
</table>
9.1.4 Strategic Objectives

The success of the settings of the operational levers is measured with the two strategic objectives *Cost* and *Dissatisfaction*. Both nodes should have a minimal value but lowering one of them will raise the other. While the cost simply results from the number of servers plus 2 percent of the bandwidth, the dissatisfaction depends at least indirectly on every node of the framework.

<table>
<thead>
<tr>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Step</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dissatisfaction</td>
<td>1</td>
<td>100</td>
<td>0.01</td>
<td>minimise</td>
</tr>
<tr>
<td>Cost</td>
<td>5000</td>
<td>20000</td>
<td>100</td>
<td>minimise</td>
</tr>
</tbody>
</table>

9.2 Results

9.2.1 Technique: View on Table

Since this solution is not supported anymore, its results do not have to be examined in detail. The table below shows the result for an solution that weights both strategic objectives equally important. As this is the result of an exact algorithm, it is the reference that iOpt has to compete with.

<table>
<thead>
<tr>
<th>Name</th>
<th>Server</th>
<th>Bandwidth</th>
<th>Pagesize</th>
<th>Dissatisfaction</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>7</td>
<td>5000</td>
<td>20</td>
<td>0.02</td>
<td>7100</td>
</tr>
</tbody>
</table>
9.2.2 Technique: iOpt with Formula Objectives

This section presents the results of target functions with different weights for the objectives. Below there are the examined functions, with an initial priority for low costs, which gets gradually shifted towards customer satisfaction. Because of the different value ranges, the third function represents an equal weighing. Note that the last function does not consider the costs at all.

1. \( o(x) = \text{cost} + \text{dissatisfaction} \)
2. \( o(x) = \text{cost} + \text{dissatisfaction} \times 60 \)
3. \( o(x) = \text{cost} + \text{dissatisfaction} \times 200 \)
4. \( o(x) = \text{cost} + \text{dissatisfaction} \times 800 \)
5. \( o(x) = \text{cost} + \text{dissatisfaction} \times 1600 \)
6. \( o(x) = \text{cost} + \text{dissatisfaction} \times 2600 \)
7. \( o(x) = \text{dissatisfaction} \)

The table below shows the according results and their number of evaluated and performed moves. The dissatisfaction is entirely neglected when it has a low priority, as it can not sufficiently influence the total cost. When both objectives are equally important, the result is almost identical with the reference of section 9.2.1. The highest customer satisfaction is reached by maximising all resources and consequentially the costs as well.

The unchanged dissatisfaction of the different costs of 4 and 5, is caused by iOpt’s higher internal number of decimal places.

<table>
<thead>
<tr>
<th>( o(x) )</th>
<th>Server</th>
<th>Bandwidth</th>
<th>Pagesize</th>
<th>Dissatisfaction</th>
<th>Cost</th>
<th>Pe. Mov.</th>
<th>Ev. Mov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5000</td>
<td>35</td>
<td>100.00</td>
<td>1100</td>
<td>28</td>
<td>335</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8200</td>
<td>50</td>
<td>12.36</td>
<td>5164</td>
<td>56</td>
<td>1005</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>5000</td>
<td>23</td>
<td>0.03</td>
<td>7100</td>
<td>118</td>
<td>3437</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>5000</td>
<td>24</td>
<td>0.02</td>
<td>7102</td>
<td>202</td>
<td>5502</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>5300</td>
<td>25</td>
<td>0.02</td>
<td>7106</td>
<td>705</td>
<td>22714</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6600</td>
<td>31</td>
<td>0.01</td>
<td>7132</td>
<td>445</td>
<td>12962</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>20000</td>
<td>50</td>
<td>0.00</td>
<td>17500</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
9.2.3 Technique: iOpt with Penalty Function Objectives

The results of the penalty functions generated with formula wizard are shown in this section. Since an analysis of all combinations of settings would hinder the identification of the impact from the individual sliders, they will be examined one by one.

Importance

To compare the results of different importance settings, the target values are set to the maximum and the other options are set to their default values (importance = 50, affinity = 25)

1. cost = 1%, dissatisfaction = 100%
2. cost = 25%, dissatisfaction = 75%
3. cost = 50%, dissatisfaction = 50%
4. cost = 75%, dissatisfaction = 25%
5. cost = 100%, dissatisfaction = 1%

<table>
<thead>
<tr>
<th>o(x)</th>
<th>Server</th>
<th>Bandwidth</th>
<th>Pagesize</th>
<th>Dissatisfaction</th>
<th>Cost</th>
<th>Pe. Mov.</th>
<th>Ev. Mov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>5000</td>
<td>50</td>
<td>0.00</td>
<td>7214</td>
<td>4166</td>
<td>29104</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5100</td>
<td>24</td>
<td>0.02</td>
<td>7102</td>
<td>2558</td>
<td>292415</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>5100</td>
<td>24</td>
<td>0.02</td>
<td>7102</td>
<td>1044</td>
<td>36081</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7900</td>
<td>48</td>
<td>12.82</td>
<td>5158</td>
<td>25</td>
<td>478</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5000</td>
<td>15</td>
<td>100.00</td>
<td>1100</td>
<td>17</td>
<td>272</td>
</tr>
</tbody>
</table>

The results of the different importance settings are almost identical to the manually defined weights of 9.2.2.

Affinity

The impact of the affinity can be analysed by increasing its value for selected target values. In this case we set dissatisfaction to 25 and cost to 10000 to provide enough space to move away from the target.

1. cost = 25%, dissatisfaction = 1%
2. cost = 25%, dissatisfaction = 50%
3. cost = 25%, dissatisfaction = 100%
4. cost = 1%, dissatisfaction = 25%
5. cost = 50%, dissatisfaction = 25%
6. cost = 100%, dissatisfaction = 25%
7. cost = 100%, dissatisfaction = 100%

The first three settings have a cost fixed at 25% while dissatisfaction is increased from 1% to 50% and then to 100%. To examine the impact on cost, the next three steps do the opposite. In the last step both factors are set to their maximum value.

<table>
<thead>
<tr>
<th>o(x)</th>
<th>Server</th>
<th>Bandwidth</th>
<th>Pagesize</th>
<th>Dissatisfaction</th>
<th>Cost</th>
<th>Pe. Mov.</th>
<th>Ev. Mov.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>5000</td>
<td>23</td>
<td>0.03</td>
<td>7100</td>
<td>27670</td>
<td>122268</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5000</td>
<td>31</td>
<td>24.80</td>
<td>5100</td>
<td>7124</td>
<td>31906</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5300</td>
<td>35</td>
<td>24.82</td>
<td>5106</td>
<td>656</td>
<td>17976</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>8200</td>
<td>50</td>
<td>12.36</td>
<td>5164</td>
<td>17008</td>
<td>66759</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>18600</td>
<td>50</td>
<td>0.00</td>
<td>9372</td>
<td>492</td>
<td>1232</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>20000</td>
<td>50</td>
<td>0.00</td>
<td>9400</td>
<td>4822</td>
<td>21276</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>5100</td>
<td>32</td>
<td>24.41</td>
<td>9102</td>
<td>31</td>
<td>955</td>
</tr>
</tbody>
</table>

The result does not change much between 50% and 100%, as any negative slope makes a value next to the target preferable. In general the affinity does work very well and finds suitable solutions for multiple nodes, like in function 7.
10 Conclusion and Outlook

10.1 Analysis of the Achieved Work

The aim of the work was the realisation of different techniques for the optimisation of RTBI’s business performance frameworks. A main aspect was the integration of the iOpt-toolkit and the therefore required transformation of RTBI’s data representation. The concept had to consider the upcoming development, by providing a structure that can easily be extended with the new components. Finally, two user-friendly graphical interfaces for the definition the objective functions for the optimisation had to be implemented. As an additional condition, the components had to be integrated with a low influence on the existing program structure.

The development can be considered successful, as all mentioned aspects are functional in the present state of the RTBI project. The limited possibilities of the previous target optimisation was extended, but still its original architecture was preserved. There is even a benefit for the users of iOpt, as the was toolkit extended with a few new invariants and the converter, that can be used to create invariants from most formula representations.

In total, the main profit of the development is the integration of iOpt, as it enables a wide range of possible realisations for the optimisation. Another important achievement are the two GUIs, that cover different user demands and operate independently from the underlying technique.

10.2 Future Work

As the algorithm and its settings could only be tested with one framework with a moderate solution space, it could possibly be suboptimal in other contexts. This means that its settings need to be evaluated again for the upcoming frameworks. But as an adjustment is not a big issue with iOpt, this will be manageable.
Another aspect are the processes based on artificial intelligence and statistics, that were not implemented during the development although they will become important in the future. Supporting them with iOpt is a task with a complexity that is hard to determine and could require an adjustment of the settings.

Refining the current state of development could be used for enhanced performance and usability. New iOpt search components especially designed for RTBI could increase the performance as well. An interface for creating and selecting different settings would be a reasonable extension in this context.
List of Figures

2.1 An example business performance framework ......................................... 4
2.2 Monitoring the current real life values in a dashboard .............................. 7
2.3 The layered architecture of RTBI ............................................................. 8

4.1 The formula editor in the adapted version .............................................. 13

5.1 The components of iOpt [8] ...................................................................... 18
5.2 Class hierarchy of a local search component .......................................... 22

6.1 Class diagram of the new value objects .................................................. 25
6.2 Class diagram of the classes for conversion ........................................... 28

7.1 The data flow during the search process .................................................. 38

8.1 The GUI of the formula wizard ............................................................... 42
8.2 Class diagram of the formula wizard GUI ............................................. 44
8.3 The penalty step function ...................................................................... 45
8.4 The penalty step function with all parameters ....................................... 46

9.1 The framework 'Server Farm' .................................................................. 48
Bibliography


