# *Konzeptioneller Entwurf eines offenen Systems zur Optimierung der Verteilung von Hardware- und Software-Komponenten in Steuerungsnetzwerken von Fahrzeugen*

Diplomarbeit im Fach Informatik

vorgelegt von


***Christoph Peter Neumann***

geb. *04.05.1980* in *Emmendingen*


angefertigt am


**Institut für Informatik**
**Lehrstuhl für Informatik 2**
**Programmiersysteme**
**Friedrich-Alexander-Universität Erlangen–Nürnberg**
**(Prof. Dr. M. Philippsen)**


Betreuer: *PD Dr.-Ing. Gabriella Kókai*
*Dipl.-Ing. (FH) Bernd Hardung*

Beginn der Arbeit: *01.12.2004*
Abgabe der Arbeit: *10.06.2005*

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Informatik 2 (Programmiersysteme), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Diplomarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den *08.06.2005*

*Christoph Neumann*

# Diplomarbeit

**Thema:** Conceptional design of an open framework for optimizing the distribution of hardware and software components in control networks for vehicles

**Hintergrund:** Existing middleware architectures for control networks in the automotive industry allow transparent relocation of software components, if boundary conditions, e. g. according to ROM/RAM size or CPU usage, are met. An exemplary size of such control networks is about 80 electronic control units, connected by 4 possible types of bus networks and hundreds of functions and software components running.

Usually the distribution of the components is generated by senior engineers and their expert knowledge without any tool-support. Often it is not possible for them to optimize the distribution according to multiple objectives. Shortcomings are mostly recognized in a late construction phase, when the vehicles are already being built. For the future it is planned to use the relocation facilities for cost savings and the process shall be automated.

**Aufgabenstellung:** The student has to develop a framework, that allows the application of optimization algorithms to the distribution of hardware and software components.

The framework has to deliver the configuration of new location distributions of the components as output. Multiple objectives have to be considered as optimization criteria (like costs minimization or bus load balancing) as well as constraints (e. g. according to ROM and RAM size limitations or availability of hardware pins).

Initial distribution configuration is either given, or has to be generated from scratch. Furthermore, partial configuration that need strict adherence have to be definable, that means that the location of some components is statically determined. The most important tasks in this thesis are the conceptual design of the modular and configurable framework and its prototypical implementation. The required input data has to be identified and for storing the information a database model has to be developed.

The optimizing kernel is not a primary goal. A basic random-based generation could be used. But the incorporation of common optimization algorithms, like evolutionary algorithms or swarm intelligence, has to be prepared. Therefore the student has to deal with the variety of these algorithms by creating a generic interface. The framework has to allow modular integration of different objectives, but the main one is cost minimization. Similarly, constraints have to be dealt with by a generic mechanism. The constraints focused on are hardware availabilities, like ROM, RAM, pins and wire. To guarantee a broadness in consideration the student has to read up architecture description languages, like EAST EEA ADL.

An XML import and export for the distribution configuration is desired, an editor supporting the engineers in providing the hardware topologies et al. is not subject of this thesis.

**Betreuung:** PD Dr.-Ing. Gabriella Kókai

**Bearbeiter:** Christoph Peter Neumann

# Kurzzusammenfassung

Moderne Fahrzeuge besitzen ein umfangreiches verteiltes System von elektronischen Steuergeräten (ECUs), die durch mehrere Bus-Systeme miteinander verbunden sind. Die einzelnen ECUs wurden mit der Zeit immer komplexer und integrieren eine steigende Anzahl von Funktionen und Software-Logik, und sind mit dementsprechend vielen Hardware-Komponenten verbunden.

Die Zuordnung der Software- und Hardware-Komponenten auf die ECUs entwickelt sich von einer *1:1* Beziehung zu einer flexiblen *N:1* Beziehung. Mehrere Kriterien, wie z. B. Kosten, Ruhestrom oder Buslast, können bei der Optimierung dieser Zuordung betrachtet werden. Ausserdem muß eine Zuordungsverteilung der Komponenten viele Zwangsbedingungen erfüllen, wie zum Beispiel die Verfügbarkeit von Resourcen.

Um Unzulänglichkeiten einer Zuordung bereits in frühen Phasen der Fahrzeugentwicklung zu erkennen, ist es notwendig, den Entscheidungsprozess durch Software zu unterstützen. In dieser Diplomarbeit werden die Anwendungsfälle, Kriterien und Bedingungen für die Problemdomäne von Fahrzeug-Steuernetzwerken analysiert und ein Datenmodell vorgestellt, das als Basis der Optimierung dient. Die Architektur und Implementierung der Optimierungsumgebung wird in zwei Teilen präsentiert.

Der erste Teil der Optimierungsumgebung trägt die Bezeichnung HeurOpt. Er stellt eine Architektur zur Verfügung, die die skalierbare Integration von mehreren Optimierungskriterien und -bedingungen unterstützt, sowie eine Schnittstelle, welche die austauschbare Anwendung von verschiedenen populationsbasierten Optimierungsstrategien erlaubt. Dieser erste Teil ist nicht auf eine bestimmte Problemdomäne spezialisiert, sondern unterstützt heuristische Optimierung mit mehreren gleichzeitigen Kriterien im Allgemeinen.

Der zweite Teil der Umgebung ist eine Problemdomänen-spezifische Erweiterung des ersten Teils namens "Multi-Objective Optimization of Vehicles Electronics" (MOOVE). Dazu gehört einerseits das Datenmodell, ergänzt durch Zugriffsobjekte, die relevante HeurOpt Schnittstellen unterstützen; andererseits werden die Implementierungen zu den konkreten Optimierungsbedingungen und -kriterien anhand des Datenmodells realisiert. Ausserdem ist ein erster prototypischer Optimierungskern implementiert, auf Basis evolutionärer Algorithmen. Weitere Optimierungsstrategien als Kern des Systems werden in Zukunft durch andere Projektteilnehmer zur Verfügung stehen.

# Abstract

Modern vehicles have a complex distributed system of electronic control units (ECUs), networked by several bus systems. ECUs are becoming more complex, integrating an increasing number of functions and software logic, and are connected to several hardware components.

The mapping or distribution between the software/hardware-components and the ECUs evolves to a complex many-to-one relation. Multiple objectives like costs, quiescent current or bus load have to be considered for optimizing the mapping as well as many constraints that have to be fulfilled, e. g. the availability of resources.

In order to recognize shortcomings of a distribution in earlier phases of the development it is necessary to support the decision process with a software framework. In this thesis the use cases, objectives and constraints for our problem domain are analysed, and a data model is presented as a basis for optimization. The architecture and implementation of the framework is presented in two-parts.

The first part of the framework is called HeurOpt. It provides an architecture that supports scalable integration of multiple objectives and constraints, as well as an interface that allows the interchangeable application of different population based optimization algorithms. This first part is not specific to the problem domain, but relates to multi-objective heuristic optimization in general.

The second part of the framework is a problem domain specific extension to the first part, called "Multi-Objective Optimization of Vehicles Electronics" (MOOVE). On the one hand it consists of the database model, supplemented with access objects that adhere to generic HeurOpt interfaces; on the other hand it implements the concrete objectives and constraints on the basis of the data model. A first prototypic optimization kernel has been implemented based on evolutionary algorithms. Different kernels are in preparation by other project members and theses.

# Contents

# List of Figures

# 1. Introduction

Modern vehicles have a complex distributed system of electronic control units (ECUs), networked by several bus systems. There are up to eighty network nodes in a single car. ECUs are becoming more complex, integrating an increasing number of functions and software logic, and are connected to several hardware components like sensors, accessors or actuators.

The mapping between the software/hardware-components and the ECUs evolves from a one-to-one relation to a complex many-to-one relation. Multiple objectives like costs, quiescent current or busload have to be considered for optimizing the mapping, not to mention the many constraints that have to be fulfilled like memory, pin, power electronics or bandwidth availabilities.

Human experts usually have problems to effectively optimize multiple objectives simultaneously. In order to recognize shortcomings in earlier phases of development, it is necessary to support the decision process with a software framework.

In this thesis the use cases, objectives and constraints for our problem domain are analysed, and a data model is presented as a basis for an optimization framework. The architecture and implementation of the framework is presented in two parts. It is implemented in Java. From the perspective of the user, it will finally provide a Pareto set of best solutions to the human expert[1].

The first part of the framework is named HeurOpt. It provides an architecture that supports scalable integration of multiple objectives and constraints, as well as an interface that allows interchangeable application of different population based optimization algorithms. This first part is not specific to the problem domain, but relates to multi-objective heuristic optimization in general.

The second part of the framework is a problem domain specific extension to the first part, named *Multi-Objective Optimization of Vehicles Electronics* (MOOVE). On the one hand, it consists of the database model, supplemented with access objects that adhere to generic HeurOpt interfaces. On the other hand, it implements the concrete objectives and constraints based on the data model, as well as a first prototypic optimization kernel using evolutionary algorithms, with different kernels in preparation by other project members and theses.

The remainder of this thesis is organized as follows: Section 2 gives a brief introduction into control networks for vehicles and optimization strategies. Section 3 explains the architecture of HeurOpt, the problem domain unspecific framework. Section 4

[1] An explanation for “Pareto set” and “best” solution is provided in section 2.3.1 and 2.3.2

presents the database model for the SW/HW-component distribution in the control networks for vehicles. Section 5 describes MOOVE, extending the HeurOpt with its problem domain specific implementations. Section 6 concludes with an outlook on further work and the summary of the thesis.

# 2. Motivation and Background

There is a great interest in a framework to facilitate the application of several optimization strategies to a problem, because future research will compare the results of the different strategies with regard to performance as well as quality and structure of the result sets. The population-based strategies that the framework aims at are for example evolutionary algorithms, simulated annealing, particle swarms or ant colonies.

A basic motivation has been the cooperation with a major automotive manufacturer. This partner provided technical background information and gave support in many vehicle-related questions. Because multi-objective optimization is a greater challenge than the single-objective one – demanding special support from a framework – the objectives "costs", "busload", "weight" and "quiescent current" had initially been defined as relevant. Additionally, the distribution mapping has to honour several constraints. Also, the supported distribution modes include an automated and a semi-automated one, which strictly adheres to manually provided fixed mapping information.

Existing implementations for machine learning and heuristic optimization, with a single objective as well as multiple ones, usually are purpose-built for specific problems and are often programmed in C or C++. In contrast, the framework architecture that is presented in this thesis was required to be as independent as possible from the vehicle distribution problem, ensuring its applicability to different problems in the future; nevertheless an in-depth analysis of the vehicle problem domain was also required and is available in form of a data model and concrete implementations.

The first section provides some basic requirements and design goals to the architecture of the framework. Then an introduction is given to the technologies and the development process of modern control networks for vehicles. Finally, an overview of population based optimization algorithms is provided, because they are considered as potential kernels of the framework.

## 2.1. Design Goals and Requirements

In addition to the required support of multiple objectives and different optimization strategies, a framework has to fulfil several generic requirements: The design goals of the framework are the separation of concerns, flexibility, simplicity, portability and small overhead. This section concretely associates these goals with the framework, and outlines some basic design decisions.

*Separation of Concerns.* Because the framework intends its application to arbitrary problems, but is driven by a concrete vehicle problem, it is required to strictly distinguish between problem specific and problem unspecific domains. Object-oriented programming is used for structuring the spheres of responsibility by inheritance and aggregation: The Java inheritance hierarchy and package structure of the framework adheres to the hierarchy of the problem domain layers.

*Flexibility.* As the primary goal of the framework is to allow the application of different optimization strategies to the distribution problem of vehicle electronics, it was required to provide interfaces for solutions, populations, objectives and constraints in a flexible way. Furthermore, a declarative definition of the Java class names, that should be used for a framework run-time configuration is required. This will be solved in the style of deployment descriptors from J2EE application servers, with optimizer classes having to fulfil a simple component model.

*Simplicity.* In spite of the complex requirements, a kind of simplicity is achieved by adherence to software development patterns from the well-known "Group of Four" [GHJV94] and "POSA" [BMR+96, SSRB00] books. For example, factories and repositories are used as well as the model-view paradigm.

*Portability and platform independence.* Although C++ does perform better due to object creation than Java, modern JVMs do not lack general performance any more. The comprehensive support of JDBC persistence layer frameworks [Apa05a], XML parsers [Apa05b] and sophisticated development environments [Ecl05] in addition to the built-in platform independence were significant for the early decision to use Java as programming language for the framework.

*Small Overhead.* To ensure a minimal overhead, efficient object cloning is required, rebuilding the complex relationship between solution, objectives and constraints. Another possibility to reduce overhead is provided to objective and constraint implementations by informing them of solution (or concrete: mapping) changes in an incremental way. For example, mutation and cross-over operations from evolutionary algorithms will change only parts of a solution. The objectives and constraints which will have to evaluate and rate the new solution can either decide to use the update information[1] if this decreases the overhead of evaluation, but they are also free to access the whole solution/mapping information if necessary.

## 2.2. Control Networks for Vehicles

Every modern vehicle has a complex network of interacting software components. The software is located on *Electronic Control Units* (ECU). Basic hardware electronics that is located on ECUs are for example microprocessors, ROM/RAM, network interfaces and power electronics, as well as pins that allow the attachment of *HW-components*.

[1]The update information is incremental from an old solution to a newly generated one.

HW-components are for example sensors and actuators: Sensors usually provide information to the ECU (e. g. pressure or temperature) and actuators usually execute commands from the ECU (e. g. motors or bulbs). HW-components are connected to an ECU. First of all, they consist of hardware electronic resources that are not inside an ECU; but there are also hardware electronic resources that are physically located on the ECU, but which are assigned and allocated to the HW-components – which will be relevant for cost estimation.

### 2.2.1. Standardization of ECUs

In the past, software components have often been implemented for an unique ECU, because of their unique hardware conditions, and because only few hardware abstraction layers were provided for component developers.

Several manufacturers have made efforts to standardize abstraction layers for ECUs. For example BMW, DaimlerChrysler, Opel, Renault and Volkswagen in form of the OSEK[2]/VDX [Sie04] architecture. The common goal is to increase the reusability of components over different ECUs. Today, standards for *Hardware Abstraction Layers* (HAL) and *Communication Abstraction Layers* (CAL) are available to component developers. The HAL and CAL are implemented as an operating system for ECUs, providing the software components with interfaces for network access or device drivers.

An example for a HAL is the OSEK-OS, which is part of the OSEK/VDX architecture, and an example for a CAL is the OSEK-COM, allowing components to transparently access different network types. Notable network types are the *Controller Area Network* (CAN)[Rob91, HIS03], *Local Interconnect Network* (LIN) [LIN00], *Media Oriented System Transport* (MOST) [MOS05] and *FlexRay* [Fle04]. Similar modules are also specified and standardized for other hardware like digital IO or access to flash memory [HIS04].

For component developers these standardization efforts also allowed to unify their development processes: designing software systems with *interconnected* vehicle functions, and enabling *modular* implementations of software components, or even model-based implementations.

### 2.2.2. Middleware Concepts

Aside from the abstraction of hardware on the control unit itself, it was subject of research to examine ways for achieving a location abstraction of sensors and actuators in relation to functional software components. It has also been examined how to achieve a location abstraction between the software modules themselves. This requires

[2] "OSEK" is a German abbreviation for "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug" with the official translation as "Open systems and the corresponding interfaces for automotive electronics".

a software layer between the hardware abstraction layer and the application software, therefore called middleware.

Several research projects developed middleware concepts for the automotive domain. For example, there have been *Architecture Electronique Embarquée* (AEE) [AEE02], *Titus* [FRHW00], and *Embedded Architecture and Software Technology – Embedded Electronic Architecture* (EAST-EEA) [ITE04, TEF+03, HWK+03]. Their common idea is to define a middleware layer to achieve a location abstraction of software components in a network. SW-components are in the so-called function software layer and/or adaptation software layer. They are subsumed as *application software*, as defined in [HKK04]. In contrast to standard software, which is potentially in every ECU, application software exists only once in a vehicle.

SW-components do not communicate with the network drivers or the ECU-local interface drivers directly. SW-components only know the interface of the middleware. After the configuration of the middleware, all signals used by the SW-component are provided in the required form. This is also illustrated in figure 2.1, where dotted lines symbolize communication between SW-components. In the automotive domain the configuration of the middleware is done at compile time, not at runtime, in order to save resources in the electronic system.

This concept of middleware adds a lot of flexibility to the location of application software. Software components can be distributed more freely in the network. Additionally, with this approach it will even be possible to relocate hardware accessors with little effort – the middleware will care for the transportation of the signals to the according software components via network.

The research resulted in the *Automotive Systems Architecture* (AUTOSAR) project [AUT04], which is still going on and whose members include more than 40 companies. Its goal is to develop a middleware standard using the results from research projects like AEE, Titus and EAST-EAA. The AUTOSAR middleware will allow to move SW/HW-components from one ECU to another in a standardized way.

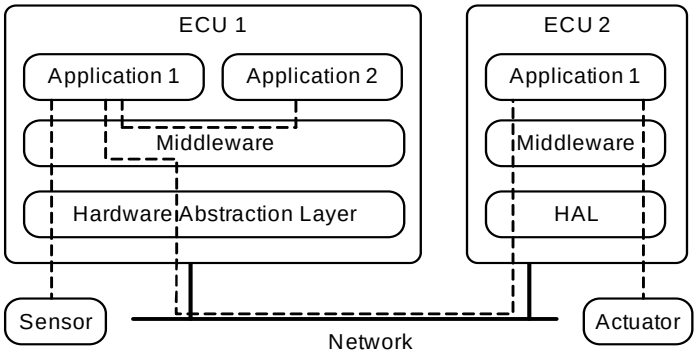


Figure 2.1.: Middleware between hardware abstraction and application

Components are referred to either as software or as hardware components, or as an aggregation of both. Hardware can be located on ECUs (like power electronics or pins) or outside ECUs like sensors and actuators which are connected to the pins. An aggregation allows to model hardware and software dependent on each other as single components. For example it is a common case that actuators need power electronics directly connected on an ECU and that driver software must be on the same ECU as well, due to timing and/or safety requirements.

With the upcoming availability of sophisticated abstraction layers, considerations on a longer run arise: The focus does not lie on the integration requirements of a single vehicle series, but on different vehicle series – or lies on vehicles of the same series, but with different combinations of extra equipment.

### 2.2.3. The Task of Distributing the Components

With the knowledge of how to map functions on different locations within the electronic architecture, another question arises, which is not addressed by the above projects: "What is a good location to move the components to?". During the development of a new type series of a car the system architects have to define which control units communicate with each other and which functionality to put in which control unit.

This problem is subject to research in form of static task allocation for several years and there exist one-pass greedy algorithms [BS98] that could be applied to the vehicle domain as long as the aspects are limited to CPU and memory consumption. Some of these algorithms even solve global resources such as the bus, but without consideration of whole networks of bus systems with its routing problems.

It is important to notice that this mapping has to be done in very early phases of development, when no implementations are available. This is due to the fact that most of the functions are developed together with the suppliers of the according ECU.

The distribution of the components usually is generated by senior engineers and their expert knowledge, without any tool-support. After the distribution decision tools like DaVinci [Vec04] can be used to configure the middleware and to generate code that performs the actual communication of a selected component, adopting to the specific network(s) that are available on the chosen ECU.

During the mapping of components to ECUs, several solutions are compared and assessed. Multiple objectives and constraints influence this decision. In section 2.3.1 and 2.3.3 a first overview about objectives and constraints is given. Human experts often encounter problems in effectively optimizing multiple objectives simultaneously. Shortcomings are often recognized in a late construction phase, when the vehicles are already being built. Therefore it is necessary to support or even automate this decision process.

## 2.3. Heuristic Optimization

Heuristic optimization is usually applied to problems with a range of solutions so big that an exhaustive search is infeasible. Especially optimization problems with multiple objectives have vast solution spaces due to the multiple dimensions. This section gives a brief introduction into basic terms and techniques that are required to understand multi-objective optimization strategies.

At first, objectives are described as well as the Pareto front, followed by an outline of constraints and violation handling. Additionally, an overview is given over population-based optimization algorithms that we consider as potential kernels of the framework.

To allow associations for objectives and constraints, the concrete ones of our problem domain are outlined. The problem domain unspecific architecture description in section 3 does not require a more detailed understanding of them, so that their in-depth description is not given until section 4, in which the use cases, problems and the data model of the vehicle domain are analysed.

### 2.3.1. Objectives

Objectives are views on a solution for evaluating the quality of a solution. The quality must be computable and consist of a numeric value.

An economical objective for any kind of manufactured good like vehicles is the "minimization of costs". For example, costs during the production depend on the installation rate of ECUs, which depends on the order rate of equipments.

Other objectives are provided by the network. At the moment we consider busload, but delay times are not considered. Busload allows two objectives: The "maximization of the busload balance" over all networks, and the "minimization of the worst-case (or average) busload" in every network. To estimate the busload, signal communication is taken into account, for example, between two SW-components that are not located on the same ECUs.

The "minimization of weight" is another objective, which is for example influenced by cable lengths. Because HW-components need direct serial connection with their mapped-to ECU, the distribution mapping has to be evaluated according to the geometrical positions of HW-components and ECUs. Cable lengths have to be calculated by the position and a cable duct model.

The "quiescent current minimization" is an objective with growing importance. For example, features like 'radio remote controlled central locking' or 'anti-theft protection' require sensor components to be active all the time. The number of necessary sensors cannot be influenced, but if such a sensor or its controlling SW-component is mapped to an ECU, this ECU needs to be active, too. Mapping all these components to a single ECU, or at least a minimal set of ECUs, allows the deactivation of a maximal set of ECUs, so that the amount of quiescent current is minimized.

This first overview of concrete objectives comprehends the objectives that are considered during this thesis. But in preparation for future requirements, the framework must be able to support an arbitrary number of different objectives.

From the above examples one can notice two common facts about objectives in general: Objectives are often in competition to each other; and according to quality, objectives are hardly correlated.

The competition between objectives can easily be seen in association with quiescent current: For its minimization we could map all relevant components to a single ECU. But wherever this ECU is located, a high amount of serial connections would have to used, because the sensors of contactless locking and alarm devices are distributed all over the vehicle, so that "weight" and "costs" will increase. If in contrast HW-components are mapped to their nearest ECU, a minimum of cables is required, so "weight" and "costs" are decreased, but the "quiescent current" consumption will increase, because many ECUs will be required to be active. There is quite more competition between our concrete objectives, but for now we just need to acknowledge this competition.

In spite of this rivalry, there is little correlation between different objectives considering the quality estimation: In general it cannot be formalised, whether for example an improvement of 0.5kg in weight is "better" than 4% less busload on some networks (under the assumption that constraints are not violated in any case). Obviously costs could be considered most important, but even costs are in trade-off with so many other objectives that it is not possible to define "how important". This fact is common to multi-objective optimization problems and motivates the concept of the Pareto front.

### 2.3.2. Pareto Front

The Pareto front[3] is the set of "best solutions". This set will always be known after each optimization iteration. It is evaluated for all solutions that are currently known, and is a subset of those. In relation to the whole set of potential solutions that fill the solution space, the set of all solutions that are currently known will itself be a subset. Because heuristic optimization does not search the solution space exhaustively, the Pareto front is just the set of best solutions that have already been tested. (Usual termination conditions for optimization algorithms are a fixed number of iterations or the conformance to minimum thresholds on each objective.)

The Pareto front is found in accordance to the quality ratings of the objectives, and the concept of *dominance*:

**Definition [Deb01, 28ff.]:** *A Solution $s^{(1)}$ is said to dominate the other solution $s^{(2)}$, if both conditions 1 and 2 are true:*

1. *The solution $s^{(1)}$ is no worse than $s^{(2)}$ in all objectives*
2. *The solution $s^{(1)}$ is strictly better than $s^{(2)}$ in at least one objective*

[3]The Pareto front is sometimes called "Pareto trade-off frontier" or "Pareto set".

The Pareto front is the set of solutions that are *non-dominated* by any others at all, as is shown in figure 2.2. More information can be found in [Deb01].

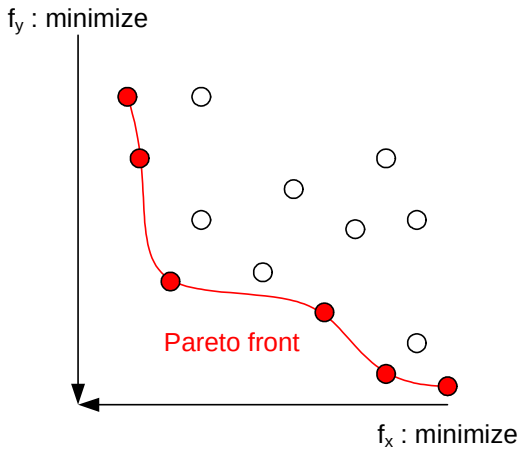


Figure 2.2.: A population of several solutions and their Pareto front for two objectives and their objective estimation functions $f_x$ and $f_y$

In contrast to providing the human expert with the Pareto front, the system could actually apply an aggregation of the quality ratings – plain aggregation models are for example linear combinations like weighted sum method, goal programming or goal attainment [Coe00]. This way the one single best solution could be determined.

Nevertheless, considering that most objectives are hardly correlated and the formalization or quantification of importance is usually not possible, aggregation-models do not give proper results. Anyway, there is little necessity for aggregation, because human experts, provided with a Pareto front, are superior in selecting the single best solution for their problem in short time.

### 2.3.3. Constraints

As it was done for objectives, the constraints for the control networks in vehicles are outlined at first to provide concrete association. Thereafter, different types of constraint handling are listed, concerning the support that the framework will have to provide.

Basic constraints for automation of the distribution mapping are hardware availability constraints. To produce valid mapping solutions, a SW-component may only be mapped to an ECU with enough memory (RAM/ROM) available. HW-components may only be mapped/connected to an ECU if pins and power-electronics of correct type and in sufficient amount are available. These basic constraints prevent the algorithm from mapping all components to a single ECU. Chapter 4 will again present enhanced analysis, for example considering OS resources like interrupts as well.

Additional threshold constraints can be provided for any objective. For example, the bandwidth usage might not exceed a certain value, just as with weight or quiescent current. In addition, safety aspects have to be taken into consideration. Certain components are only allowed to be mapped to a limited number of ECUs. Redundant components for the same feature, like two airbag controls, are required to be mapped on separate ECUs.

Constraint violations in general can either be handled by tabu violation or by degrees of violation. The *tabu violation* discards the whole solution, if any constraint is violated.; whereas *degrees of violation* estimates the violation using a function, so that constraints could be considered as additional objectives.

If degrees of violation are used, the Pareto front can optionally apply *constrained-domination*: the ranking in relation to constraint-objectives dominates rankings according to any normal objectives. This implicates that a kind of aggregation-model is necessary for constraint-objectives, because all constraint-objectives have to be in effect before "real objectives" are considered. For constraint-objectives a plain aggregation is usually sufficient as model.

A third kind of constraint violation handling is the *penalty approach*, where constraints are not treated as objectives. Instead, a penalty value for a constraint violation is added to each objective function. There exist several strategies for adding the penalty, e. g. static, dynamic, annealing or adaptive penalties [OY05].

The HeurOpt framework supports degrees of violation for all constraints, and tabu violation if a threshold is defined for the constraint. There are no default implementations of the penalty approach strategies yet, but all necessary information is available for an optimization strategy to apply penalties on its own.

## 2.3.4. Population Based Optimization Algorithms

Population based optimization algorithms are characterized by a set of solutions. The solutions are called "individuals" and the set "population". Population-based optimization algorithms evolve generation for generation. Typical examples are *evolutionary algorithms* and *swarm intelligence*.

### 2.3.4.1. Evolutionary Algorithms

Evolutionary algorithms (EA) [Hol92] represent potential solutions as *individuals*. The individuals are encoded as *genomes*. A whole set of solutions/individuals is called *population*. Imitating nature's breeding, the EAs apply operations like mutation and cross-over to genomes or pairs of genomes in order to evolve the population. Because the cardinality of the population usually should remain constant or in some boundaries, a selection of the genomes is applied based on the numeric *fitness* of a genome. For multiple objectives the fitness is a vector.

There are two views on the individuals, the genotype and the phenotype. The *genotype* is the encoding and syntax, represented by the genomes of the individuals. The *phenotype* is the semantics of the information encoded by the genotype. The phenotype is used to interpret the genotype for darwinistic judgement.

The *seeding* of the population refers to the creation of solutions from scratch. The seeding can be done initially or iteratively. Initial seeding means that after the initialization phase, new solutions are created exclusively by applying evolutionary algorithms on existing individuals. Whereas iterative seeding means that even after the initialization phase the algorithm may decide to create new individuals from scratch.

The *generation* of a population usually is counted as an integer number. The ancestry between individuals may be stored as a reference. The framework does not provide standard facilities to store inheritance information for individuals.

Standard operations for EAs are mutation and breeding. The *mutation* operation prevents the population from becoming too similar, and to avoid local minima. The *breeding* is a recombination of usually two individuals. The equidistant breeding operations are named *cross-over*. Several examples are the one-point, two-point, uniform (UX) and half-uniform (HUX) cross-over. A non-equidistant breeding operation is for example *cut and splice*, and will change the length of the genome.

Three basic classes of evolutionary strategies are named *genetic algorithms* (GA), *genetic programming* (GP) and *evolutionary strategy* (ES). They differ for example in their used genotype and in their most important genetic operation. More information can be found in [Deb01, 81ff.].

#### 2.3.4.2. Swarm Intelligence

For swarm intelligence (SI) [BDT99] there are two important kinds: The *particle swarm optimization* (PSO) [KE95] algorithms and the *ant colony optimization* (ACO) [CDM91] algorithms.

An PSO algorithm uses the analogy of a swarm of birds or a fish school. The particle's position in the solution space is not only defined by its *location*, but also by a *velocity*. Based on the location and the velocity, the decision process for the next iteration uses not only the "own experience" of a particle but also the other particle's experience. For example, the update includes the weighted current velocity for diversification, the randomly weighted distance to the personal-best for local intensification and the randomly weighted distance to the swarm-best for global intensification.

The ACO algorithms use the analogy of ants finding food. The update decision is influenced by randomness and pheromones. The pheromone concentration is higher on shorter paths, because ants travel on them more frequently. The changing concentration can be used as feedback system, so that the decision made by an individual ant for high-pheromone paths is more probable than for less-pheromone paths. For the sake of diversification, the probability for less-pheromone paths is above zero. The goal is to find new solutions, not to affirm known ones.

In the abstract problem space there is a defined start point (colony) and end point (food); the *abstract shortest path* between the two points represents a possible composition of a solution. In nature the ant simultaneously travels and pours out pheromones; the ACO performs the travelling and the pheromone distribution in separated phases: An agent memorizes its way during one iteration, and the pheromone evaluation is done after the end of the algorithm, leading to a pheromone adjustment by simulating the same way back to the colony. The digital pheromone is *accumulative* for several ants at the same location, and *volatile* by reduction after each iteration. An heuristic function is applied for each parting of the ways, being influenced by the pheromones. A bad heuristic at the beginning leads to favourism of solutions at the end.

Often in multi-objective optimization, separate ant colonies are used for each objective [MM99]. Shelokar et al. [SAV+00] extend this approach with global ants that search for good regions in the whole search space, whereby local ants improve the fitness locally within a given region.

#### 2.3.4.3. Related Algorithms

From the viewpoint of programmatic interfaces, simulated annealing could also be considered to have a population of cardinality "1", as well as most of the available one-pass greedy algorithms that are available for static task allocation [BS98][4]. Therefore, they could be integrated into a population-based framework, in order to compare all strategies according to performance and quality of results.

Nevertheless, most of these algorithms are limited to CPU and memory consumption, as has been mentioned in section 2.2.3. Some of these algorithms even solve global resources such as the bus, they do not consider whole networks of bus systems and routing problems or other kinds of objectives and constraints. An adoption of these algorithms to support arbitrary objectives needs further evaluation.

## 2.4. Summary

Section 2 provided the motivation and background information for control networks for vehicles, as well as optimization strategies. After the introductory motivation, some design goals and requirements were outlined in section 2.1. The introduction to the control networks for vehicles was given in section 2.2, wherein the standardization process for ECUs was described, as well as middleware concepts and an explanation for the task of distributing the components.

Section 2.3 provided information about heuristic optimization and related strategies. At first, section 2.3.1 gave examples for objectives, as well as interrelationships between several objectives. This gave motivation for the concept of dominance and

[4] Beck and Siewiorek provide quite a good bibliography for static task allocation algorithms of graph theory, mathematical programming and simulated annealing.

the Pareto front in section 2.3.2. Section 2.3.3 introduced constraints, gave examples and discussed different types of violation handling.

Finally, population based optimization algorithms were outlined in section 2.3.4. First, the principles of evolutionary algorithms were explained. Then, a basic introduction in swarm intelligence was given, with its most important representatives particle swarm optimization and ant colony optimization.

# 3. HeurOpt Open Framework

This section describes the multi-objective optimization framework. Although the development work was done in the concrete problem domain of "vehicle electronics", one of the architectural goals was to separate domain specific logic from domain unspecific logic, as shown in figure 3.1.

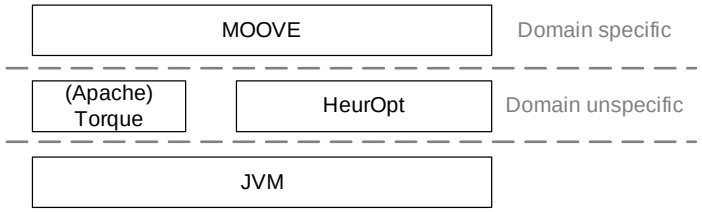


Figure 3.1.: HeurOpt and MOOVE software layers

The HeurOpt framework has been designed to deal with solution populations, solutions, objectives and constraints in a generic way by defining interfaces and implementing abstract classes, component repositories and configuration management. The interrelationship between solutions and objectives/constraints is implemented based on the model-view paradigm, as will be seen in section 3.5.

Because the problem of mapping one set (of targeting items) completely to another set (of targets) does not immediately relate to vehicles, implementations for this abstract problem domain are also part of HeurOpt. They can be reused for similar mapping problems. They will be the basis for the MOOVE project discussed in section 5. For example, the SW/HW-*component instance* class will extend the *targeting item*; and the *network node* (ECU) class will extend the *target*.

After this section has provided the principal architecture for the multi-objective framework, the next section 4 will analyse the vehicle domain specific problems and use-cases in form of a database model. Subsequently, section 5 will describe the MOOVE extension to the HeurOpt kernel. Torque from figure 3.1 is a database persistence layer by the Apache group. It will be used by MOOVE. Although the HeurOpt interfaces and classes are domain unspecific, in this section we will use the concrete objectives and constraints of our domain as examples. These have already been outlined in sections 2.3.1 and 2.3.3.

## 3.1. Initialization

The central part of HeurOpt is the initialization process shown in figure 3.2. The relationships between population and solutions as well as between solution and objective estimations are defined during createPopulation(), and are explained in sections 3.4 to 3.6. A HeurOpt optimizer is the aggregation of a termination condition and a solution population. The configurator initializes the termination condition with a reference to the population.

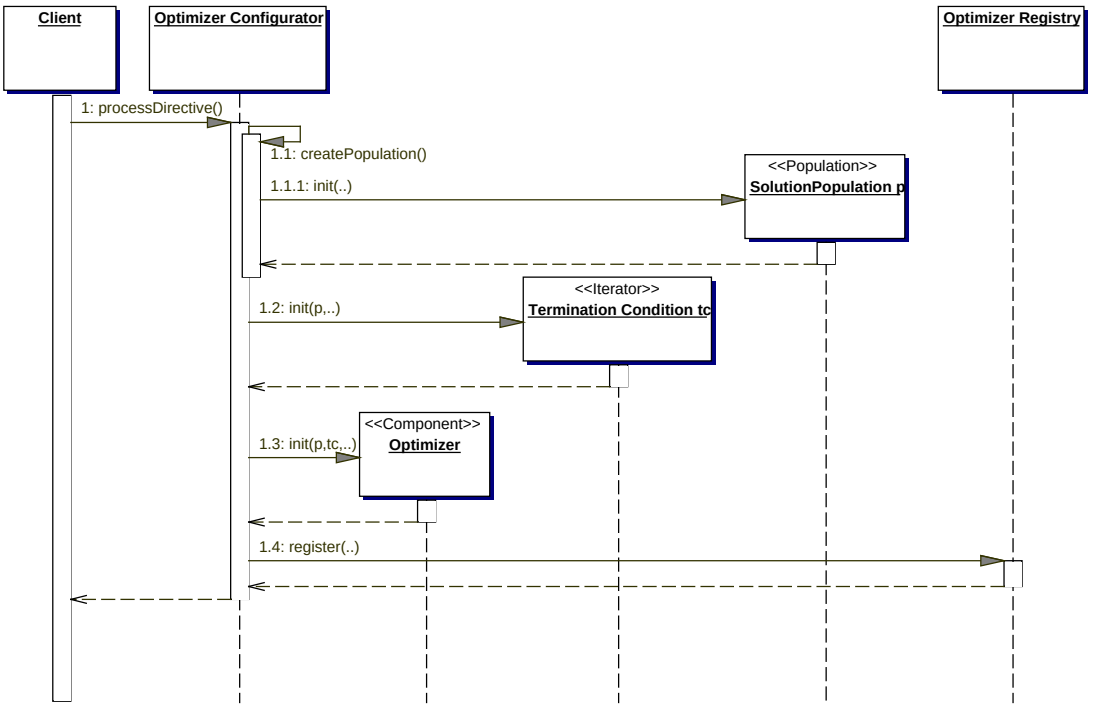


Figure 3.2.: Initialization of the optimizer component

The interfaces for the component model can be found in the package de.fau.cs.i2.pattern.component, with its UML interface description shown in figure A.1 of the appendix. Actually, the optimizer implements the ResumableComponent interface and not the basic Component interface. The resumable component supports run(), resume() and suspend(). The basic component[1] interface just provides standards for initialization, finalization and information retrieval.

The component model from the de.fau.cs.i2.pattern.component package, is generic and independent of the HeurOpt framework. It is enhanced by the framework for multi-

[1]Example classes for the basic Component interface are the factory classes for the objective estimation objects, described in section 3.6.

objective optimization: In package de.fau.cs.i2.heuropt.kernel.configuration the Optimizer interface extends the ResumableComponent, for example requiring the method getParetoFront(). The overview of this package is shown in figure A.4 of the appendix.

The package de.fau.cs.i2.pattern.component also provides the interfaces and abstract implementations for the component repositories. The interface for a ComponentRepository requires the insert(..), remove(..) and find(..) methods. The component objects are managed in the repository by String identifiers. The string IDs consist of class names and run-time generated integer IDs. There are separate repositories for the two types of component, because only the repository for the ResumableComponent objects will be accessible to a client application; all other Component objects are only for internal usage. The repositories are available as part of the de.fau.cs.i2.heuropt.kernel.configuration package.

Finally, the OptimizerConfigurator in figure 3.2 is also part of the package de.fau.cs.i2.heuropt.kernel.configuration. It implements the ComponentConfigurator interface from package de.fau.cs.i2.pattern.component.configurator. A configurator is the first object visible to the client application. The client provides a ComponentConfigurationDirective to the configurator's processDirective(..) method. The directive in figure 3.2 is used for creation, but a directive could be considered for several purposes, e. g. for suspending an optimizer and reconfiguring it at run-time. Additionally, the configurator allows the client application to get a reference to the component repository.

Component, component repository and component configurator are patterns from [SSRB00, 75–107]. The current implementation does not use XML descriptor files, but the configurator internally uses handler methods, string representations and Java reflection for all dynamic information as a direct preparation for XML parsing.

## 3.2. Highest-order Iteration

Before we discuss the population with its solutions and objective estimations, a short overview of the interaction between client, component and termination condition is given, shown in figure 3.3.

The Optimizer in its role as a resumable component allows to suspend() and resume() the optimization iteration. The suspension can be triggered asynchronously and will suspend the component before the next generation would be generated. The optimizer will use the termination condition as ConditionalIterator, using hasNext() to find out whether the condition is still not met, and calling next() to initiate the next iteration.

The interface for the ConditionalIterator is defined in the package de.fau.cs.i2.heuropt.kernel.termination, shown in figure A.5 of the appendix. A simple implementation based on the number of iterations is available as CountedConditionalIterator in the de.fau.cs.i2.heuropt.domain.generic.termination package. More sophisticated termination conditions can easily be integrated, e. g. ones that are based on thresholds for objectives and the evaluation of the Pareto front, which can be accessed by getParetoFront() like it is

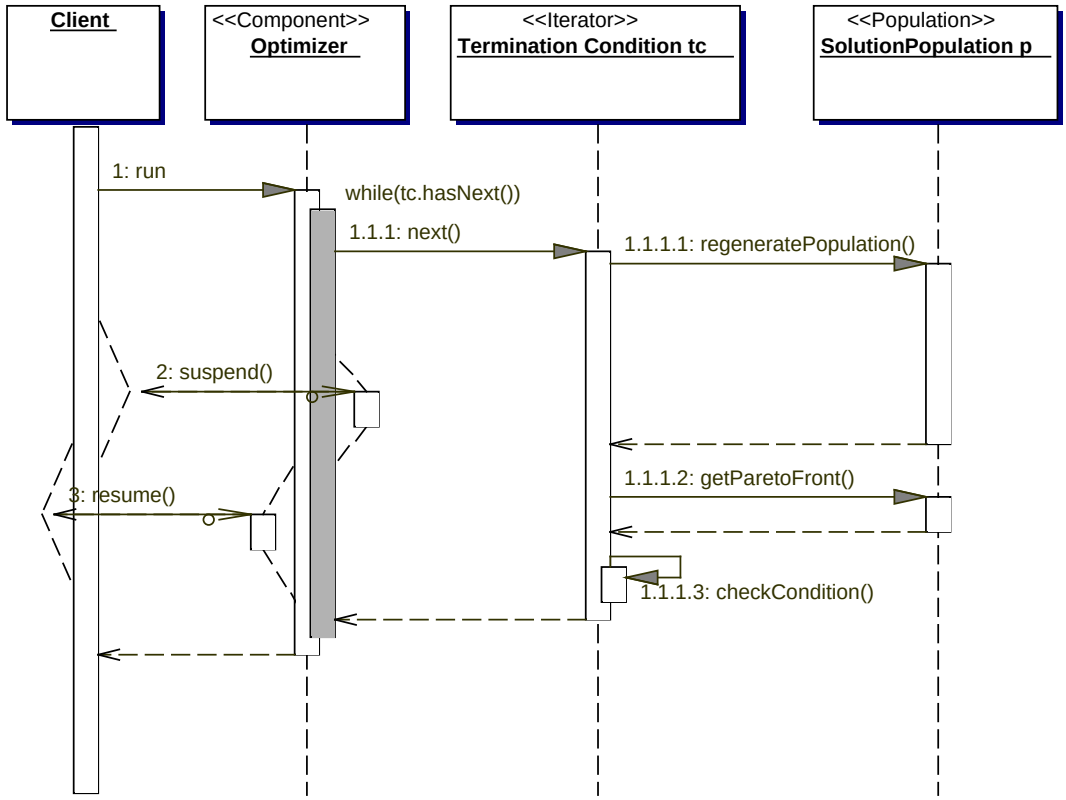


Figure 3.3.: Running the optimizer component

outlined in figure 3.3. Because most iterators need access to the population for deciding its termination condition, the invocation of regeneratePopulation() on the population is not done by the Optimizer itself, but is completely delegated to the ConditionalIterator.

## 3.3. HeurOpt Layers

Figure 3.4 gives an overview of the HeurOpt classes. Optimizer, configurator and repository as well as the termination condition have already been discussed. The optimization strategy itself is represented by a concrete implementation of the population and solution interfaces.

The following sections will describe the different modules. The population aggregates solutions. The solutions are linked to objective estimations, which will result in the quality ratings for a solution according to an objective. Constraints are considered objectives with an enhanced interface for violation handling.

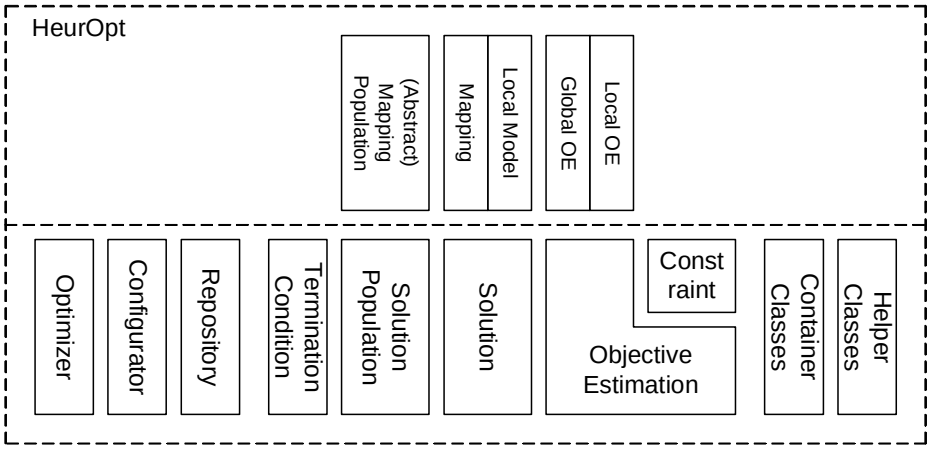


Figure 3.4.: The HeurOpt layers

The top layer of figure 3.4 consists of classes that support the mapping problem of distributing one set (of targeting items) completely to another set (of targets), bearing no relation to vehicles. Vehicles are not considered until section 5, with its descriptions of the MOOVE extensions.

The package structure can be outlined as follows: In general, the de.fau.cs.i2.heuropt.kernel package with its subpackages contains only interfaces and abstract classes. Concrete classes are hardly ever part of this package, but are mostly implemented in the subpackages of de.fau.cs.i2.heuropt.domain. Implementations that are considered to be usable for many kinds of multi-objective optimization problems are part of the de.fau.cs.i2.heuropt.domain.generic packages. In contrast, the de.fau.cs.i2.heuropt.domain.mapping packages will be related to the domain of distribution mappings. So, for example, a concrete implementation of the Optimizer interface is the GenericOptimizer in package de.fau.cs.i2.heuropt.domain.generic.configuration, shown in figure A.11 of the appendix.

## 3.4. Populations

The basic overview of this section is outlined in figure 3.5. The population is defined by the SolutionPopulation interface, as part of the de.fau.cs.i2.heuropt.kernel.population package, shown in figure A.6 of the appendix. The population stores the solutions, using a collection based on the SolutionSet interface from package de.fau.cs.i2.heuropt.pareto, shown in figure A.8 of the appendix. In fact, the Pareto front implementations adhere to the SolutionSet interface (see section 3.7).

The SolutionPopulation and Solution interfaces standardize methods, e. g. for initialization and information passing, but they do not standardize the interaction between themselves according to an optimization strategy: The Solution interface allows generic storage and access to the quality ratings of its objectives, but the operations that the

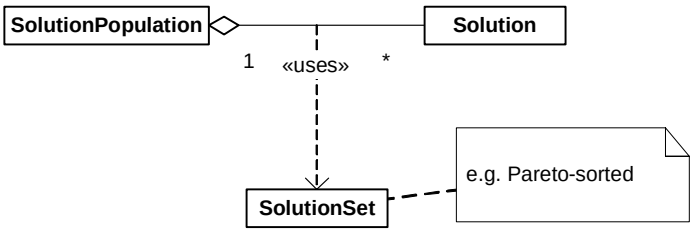


Figure 3.5.: Section overview: Populations

population will apply to its solutions in order to evolve is heavily dependent on the optimization strategy – for example whether it is a colony of ants, a swarm of particles or a genetic population of individuals[2]. Because of that, a population must explicitly define a set of supported solution classes. The class names of population and solution can be configured.

A population is required to choose itself a default solution class, which must be accessible by getDefaultSolutionClass(). If a dedicated solution class is actually configured by a directive, the population is asked by supportsSolutionClass(..) whether it is compatible, and it will be used instead of the default one. Specialized implementations for an optimization domain are free to use Java reflection to ensure interface compatibility with special child-interfaces of the Solution.

An important functionality of a SolutionPopulation is the method generateInitialSolutionPopulation(..). This method is supported through the SolutionPopulation's generateInitialSolution(..) method and the Solution's getRawsolutionClone() method: During initialization a first single solution is set up, more specifically the objective estimations are set into relation with this raw solution object. For evolving a population there has to be an efficient way to clone a solution object. Both the relationship between solution and OEs, and the cloning mechanism are described later. For the moment it is just important that the getRawsolutionClone() allows to get a clone of this first *raw solution*, with its structure being set up completely, but which is an empty solution that does not contain any domain specific solution information.

On the basis of a raw solution, the population has to implement the generateInitialSolution(..) which leads to an *initial solution*: In contrast to the raw solution, the initial solution is allowed to contain some actual solution information that will be constant to all solutions of one optimization cycle. For example when fixed parts of a solution are provided to a semi-automated optimization, these will be part of the initial solution. Additionally, heuristics can be used to deduce additional parts of the solution that will not be changeable. The information stored in the initial solution can help to reduce the solution space significantly.

[2]It is the goal of future work to provide problem domain independent interfaces and default implementations for the specific operators of evolutionary algorithms, ant colony optimization and particle swarm optimization.

The population is, of course, free to use the raw solution as initial solution. In any case it will have to implement the generateInitialSolutionPopulation(..) method that creates the first set of solutions. The framework does not constrain these solutions in any way. They may be incomplete or even invalid.

At the moment, the creation of the initial population is enforced by the framework during initialization of an optimizer, actually as the last step. It would be possible to separate it from the initialization into an independent optimization phase, but has not yet been done. After the initial population is created, the optimizer is ready to run(), which essentially leads to iterative invocations of the population's regeneratePopulation().

## 3.5. Solutions

The basic overview of this section is outlined in figure 3.6. The Solution interface, like the SolutionPopulation, is part of the de.fau.cs.i2.heuropt.kernel.population package, shown in figure A.6 of the appendix. The most important role of a solution is the one as a model, according to the model-view paradigm, which is primarily defined in Java by the java.util.Observable interface. An alternative is the model-view-controller (MVC) pattern in [BMR+96, 125–143]. There is some semantic gap between the Sun JDK definition of model and view and the one from the MVC pattern, because the MVC pattern differentiates between observer and view and the model-view paradigm treats them synonymously.

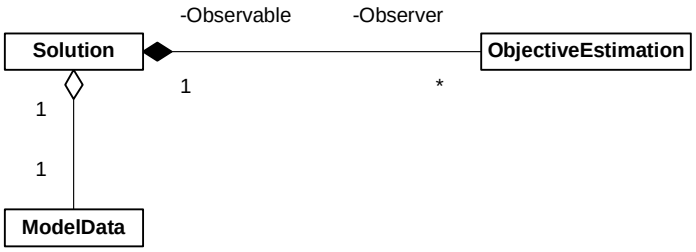


Figure 3.6.: Section overview: Solutions

In the de.fau.cs.i2.pattern.MVC package, shown in figure A.3 of the appendix; the complete MVC pattern is defined according to the design pattern, but the framework only makes use of the Model, ModelData and Observer interfaces. The term "view" is used with the semantics of an observer, in accordance to the Sun definition and in contrast to the MVC pattern definition. The Model interface will be implemented by solutions. The view that will actually be implemented by the objective estimations (OE) is the ObserverOfSolution interface as part of the de.fau.cs.i2.heuropt.kernel package. The two standard use-cases of the model-view pattern are shown in figures 3.7 and 3.8: the initialization as well as the notification and update.

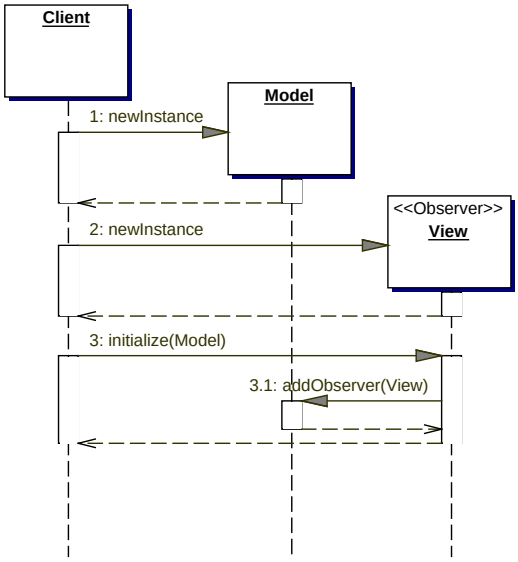


Figure 3.7.: Instantiation and initialization of model and view

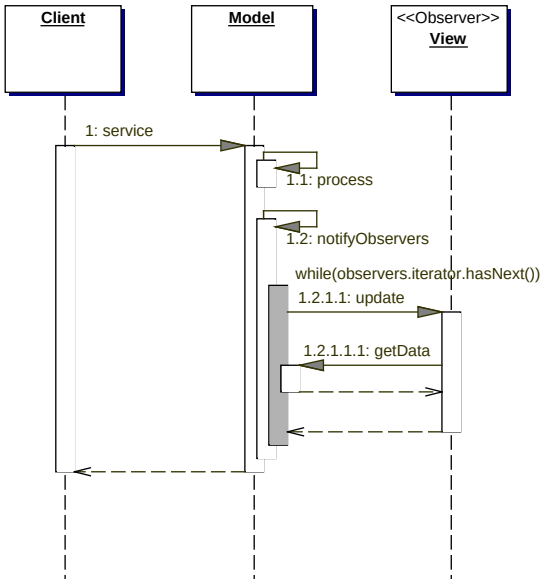


Figure 3.8.: Notification and update between model and observers

Although we will discuss the OEs in full detail separately, it is necessary to outline them in this context: The ObjectiveEstimation objects have to evaluate a solution according to an objective. The result is provided as a numeric value, represented as QualityRating objects[3]. In section 3.6 we will see how OEs are created by factories and how constraints are extending their interface. The most important role of an ObjectiveEstimation in relation to a solution is the one of an observer/view.

The Model has two roles. The first one is to store the data of the model. In the same package as the model and view lies the ModelData interface, which is extended by the SolutionModelData for optimization strategies in general, as part of the de.fau.cs.i2.heuropt.kernel.population package. The data model is further extended by the several optimization domains, for example in the de.fau.cs.i2.heuropt.domain.generic and de.fau.cs.i2.heuropt.domain.mapping packages.

The second role of the Model is the one of an Observable. This means the management of the observers that will be registered at the observable. In the context of optimization iteration, the ObjectiveEstimation classes are the only classes that implement the Observer interface. Because the solution will implement the role of a model, and because the objective estimations will implement the role of an observer, there are two equivalences to remember: the Solution/Model/Observable and the ObjectiveEstimation/view/Observer.

[3]For example, an evolutionary algorithm names such a quality rating a “fitness”.

The significant methods of the Model are addObserver(..), deleteObserver(..) and notifyObservers(). Especially the addObserver(..) is used during initialization of the framework to register the OEs at a solution. The notifyObservers() has to be invoked when the model is changed. The model will iterate over its observers invoking their update(..) method. According to the design pattern, the parameters of the update methods contain a reference to the model and an object that should represent the model change. Although each observer/view stores a reference to the model, the reference to the model is still provided as parameter of the update method because it would be possible for an observer to register at different models. For the solutions and OEs this is not necessary, because separate OE instances are used for each solution. Support for sophisticated cloning is provided instead of reusing OE instances for multiple solutions, which would introduce management overhead to the OEs for the demultiplexing, especially for incremental updates. Nevertheless, the interfaces adhere to the design pattern and would allow such alternative implementation of solution and objective estimation.

Beyond the role of a model, the Solution interface requires the method getQualityRatingMap() to access a map consisting of objective types and their concrete quality ratings, which will be calculated on access by the registered OE objects. Additionally, the OEs themselves are accessible by getObjectiveEstimationVector(). The solution also has to implement a dominates(..) method that compares itself to another solution according to their quality ratings and the dominance definition from section 2.3.2.

The getRawsolutionClone() method that is used by a solution's population to build its initial solution set (see section 3.4), is the last requirement for a generic Solution. An abstract default implementation for all mentioned methods is provided by the classes in the de.fau.cs.i2.heuropt.kernel.population package, which allows a developer to concentrate solely on the implementation of the optimization strategy's algorithms.

## 3.6. Objective Estimations

The previous section described how objective estimations (OE) implement the role of an observer/view, being registered to a solution as their model. During the optimization iteration, the OEs are informed about model changes via their update(..) methods. Each optimization domain is advised to define special model-change objects, and to extend the abstract model and view classes with typecasted notifyObservers() and update(..) methods. This allows type safety and reduces run-time overhead. The classes in the de.fau.cs.i2.heuropt.domain.mapping subpackages demonstrate this.

The remainder of this section at first describes the registration process between solution and OE during the initialization phase, which is based on factories. This will lead to individual OEs that handle several types of objectives simultaneously. Furthermore, the access to the quality rating(s) is described. Finally, the extension from objectives to constraints is discussed.

Let us remember the initial configuration steps for an optimizer component: The ComponentConfigurator will collect all names from the directive for the configurable classes like population, Pareto front, solution and OEs. It will transform the names to Class objects using Java reflection. The class for the population is instantiated, using the empty default constructor. The initialization of the SolutionPopulation is standardized by the init(..) method. It requires three parameters: an EstimationfactoryToObjectivetypesMap, the Class object for the solution-instantiation and a domain specific property HashMap.

The creation of the EstimationfactoryToObjectivetypesMap requires an explanation of the OE-factories and the objective types. The factories are components on their own, and they are in charge of OE classes that may be responsible for several objective types. The basic overview of the relevant classes is outlined in figure 3.9.

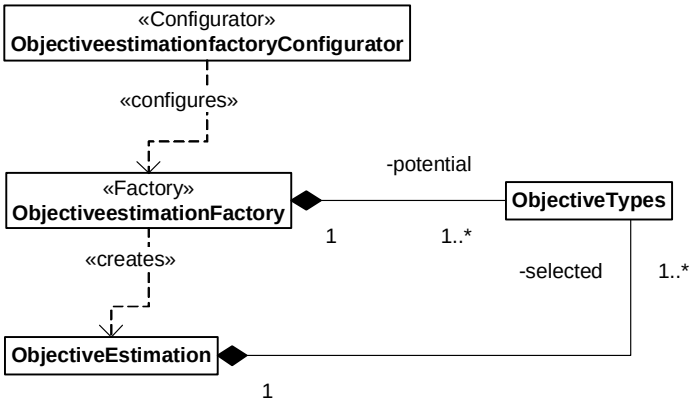


Figure 3.9.: Section overview: Objective Estimations

### 3.6.1. Objective Estimation Factories

The factories adhere to the basic and non-runnable Component model (see section 3.1), and are part of package de.fau.cs.i2.pattern.component, shown in figure A.1 of the appendix. The instantiation of the OE-factories is part of the creation process for the EstimationfactoryToObjectivetypesMap, done by the ComponentConfigurator for an optimizer. The relevant sequence is shown in figure 3.10.

An ObjectiveestimationFactory component is responsible for exactly one ObjectiveEstimation class. The default implementation of the factory can be instantiated with any OE Class object, so that the developer of an OE will not have to implement a factory. The factory is necessary because an ObjectiveEstimation may be responsible for several objective types.

The factories for the OEs have their own configurator, in form of the ObjectiveestimationfactoryConfigurator class in the de.fau.cs.i2.heuropt.kernel.configuration package.


Figure 3.10.: Creation of the EstimationfactoryToObjectivetypesMap object

The potential objective types that are supported by the OE class, are defined in a declarative form. At the moment a database is used: The database contains a relation of objective types, identified by Integer-IDs. Another relation keeps the OE class names and a third relation relates them to the objective types, thereby expressing all potentially supported objective types for a single OE class. The same information could be provided with XML deployment descriptors for each OE.

If the ObjectiveestimationfactoryConfigurator is requested to create an OE-factory, it first tries to find a previously instantiated factory for the OE in the HelperobjectsfactoryRepository, using the OE class name as identifier. This repository is for internal usage and may contain any Component object. If the factory is not yet available, the configurator will create the factory and will register it to the repository, using the OE class name as String-ID. Before factory creation, the configurator reads the list of all supported objective types for the OE class into an ObjectiveTypeSet. This is achieved by accessing the database, but could also be done by parsing an XML descriptor. The ObjectiveTypeSet is part of the de.fau.cs.i2.heuropt.container package, and is delivered to the initialization method of the factory. Later the factories will be available to the

population, which creates OE instances that probably handle only subsets of their potentially supported objective types.

Now back to what the ComponentConfigurator does for an optimizer: The directive is parsed, so that an OE class name is mapped to a list of objective type IDs. For each OE class name that is requested, the ComponentConfigurator will use the ObjectiveestimationfactoryConfigurator and HelperobjectsfactoryRepository to get a factory for the OE class. Then the ComponentConfigurator transforms the actually requested list of objective types into an ObjectiveTypeSet. This set of objective types has not (yet) been passed to the factory, but will later be used to create OE instances, instrumenting the factories. The OE-factories and the set of requested objective types are put together into an EstimationfactoryToObjectivetypesMap. This map is part of the de.fau.cs.i2.heuropt.container package and is finally the last parameter that is needed to initialize the population.

### 3.6.2. Instantiation of Objective Estimations

After the ComponentConfigurator has created the special EstimationfactoryToObjectivetypesMap, it uses Java reflection on the solution's class name, declared by the directive, to create the according Class object. Then it calls this.createDomainSpecificPropertiesForSolutionPopulation(), which has to return a HashMap that will be passed as third parameter to the init(..) of the population. Domain specific configurators are free to extend the directive and to pass arbitrary properties to their populations.

By invoking init(..) with three parameters, the configurator delegates the rest of the optimizer's initialization process to the population object: The population uses Java reflection to instantiate the raw solution object. Then it iterates over the key set of the EstimationfactoryToObjectivetypesMap, invoking create(..) on each ObjectiveEstimationFactory. The whole process is shown as sequence diagram in figure 3.11.

The OE-factory's create(..) method requires a Solution and an ObjectiveTypeSet as parameter; the population uses the raw solution and the objective type set from the EstimationfactoryToObjectivetypesMap. The OE-factory verifies that the requested objective type set is actually supported by the handled OE class. Then it constructs the ObjectiveEstimation object using Java reflection and the default constructor. The initialization of the OE object is done twice, for its two roles separately: At first the init(..) in the view role is invoked, passing the solution as model. According to the design pattern, this init(..) includes the registration of the view to the model in form of an addObserver(..) invocation. Then another init(..) for its intrinsic role as OE is invoked, passing the ObjectiveTypeSet of handled objective types and a domain specific properties HashMap, which defaults to null and may be used arbitrarily.

After the population has iterated over all OE-factories, the raw solution is initialized according to its role of a model. Its intrinsic role as solution has still to be initialized, which will for example result in the creation of specific data structures, e. g. the structures that represent the distribution mapping of SW/HW-components in control

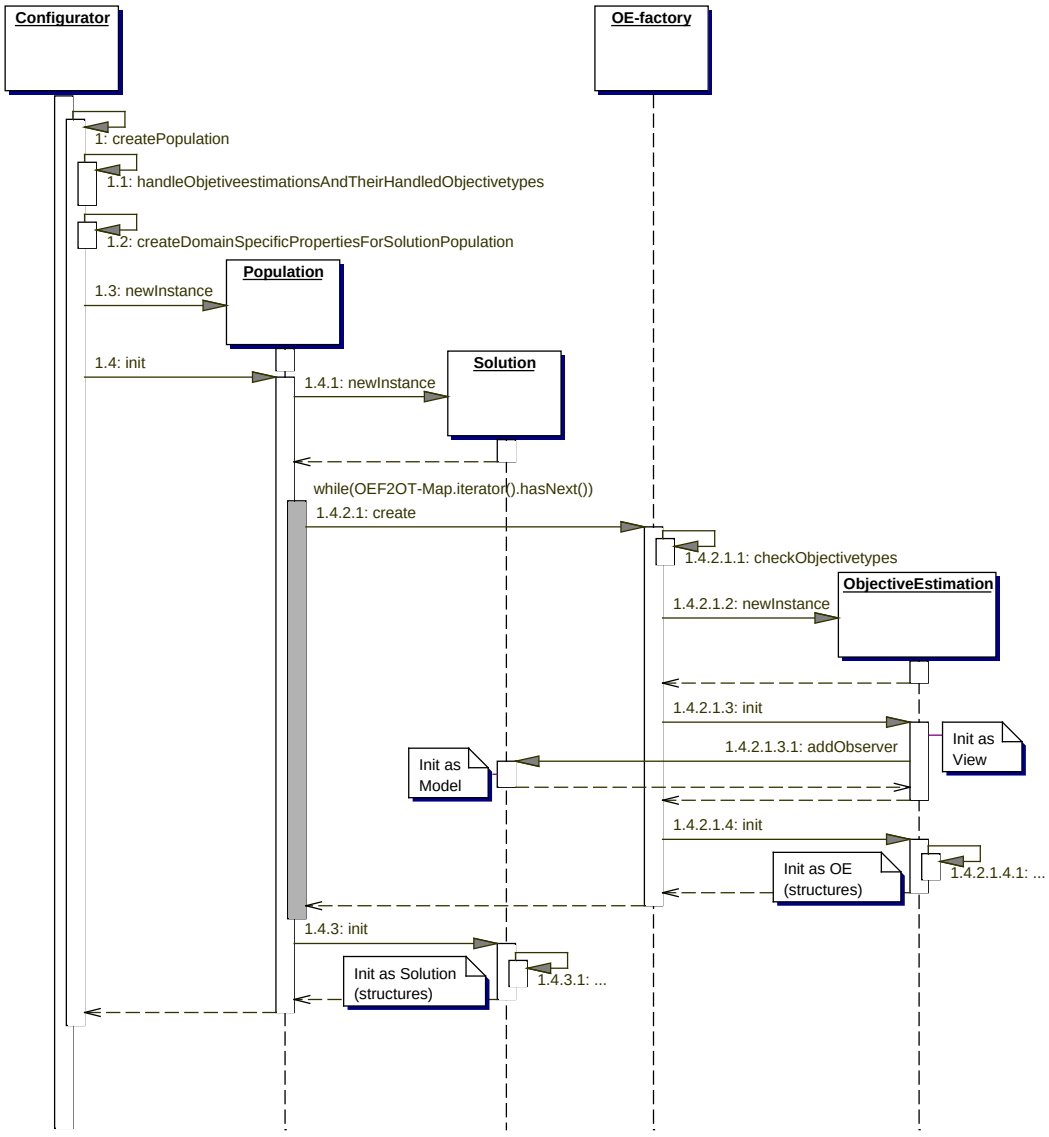


Figure 3.11.: Instantiation of objective estimations

networks for vehicles. For this purpose, an init(..) of the solution is invoked by the population, passing again a domain specific properties HashMap that defaults to null. The HeurOpt implementations for the upper part of figure 3.4 extend this init(..) in a non-trivial way, which will be discussed in section 3.9.

### 3.6.3. Constraints

Constraints have to implement the ConstrainedObjectiveEstimation interface, a part of the de.fau.cs.i2.heuropt.kernel.objective, extending the ObjectiveEstimation. The constraints are declared like objective estimations and are free to provide degrees of violation (see section 2.3.3) in form of their quality rating. If a tabu violation is to be signalled, the isViolated(..) method has to return true.

The isViolated(..) information has to be polled, either by the population or the Pareto front. The alternative would have been the exception throwing during solution/ model update notification. During the notifyObservers() iteration over all observers an exception would disrupt this method, because it needs to pass the exception up to the population enabling it to decide how to treat the constraint violation. But this disruption leads to different states of the diverse objective estimations – the ones that have updated their information before the exception was thrown, and the ones that have not yet been informed due to exception throwing. Therefore, a synchronization would be necessary. Further work has to decide wether such complexity finally reduces overhead by avoidance of the polling.

## 3.7. Pareto Front Implementations

In package de.fau.cs.i2.heuropt.pareto there exist interfaces for Pareto-sorted solution set implementations according to the introduction in section 2.3.2. The SortedSolutionSet interface defines the getParetoFront() method in particular. The basic SolutionSet interface is used for non-dominated subsets, for example the Pareto front itself.

At the moment, there are two different Pareto-sorted implementations which have not been programmed by myself, but by Bernd Hardung: The de.fau.cs.i2.heuropt.pareto.leveled package implements the algorithm from [Deb01, 40ff.] that classifies the entire population into various non-dominated levels. In the de.fau.cs.i2.heuropt.pareto.SPEA2 package, an implementation of the SPEA2 [ZLT01] algorithm exists. The SPEA2 does not have several levels, but just the Pareto front and the rest. It uses a fixed population size and supports sophisticated updating of the Pareto front upon insertion of a new solution. The size will be reduced to the fixed number. The reduction adheres to diversification and keeps the most distant and diverse solutions always as part of the set. Both implementations are additionally provided as UML diagrams in figures A.9 and A.10 of the appendix.

## 3.8. Model Data for Mappings

This section will introduce some generic classes that deal with the mapping problem of distributing one set of *targeting items* completely to another set of *targets*. These classes/interfaces will be used by the MOOVE extension in section 5: the targeting items will be the SW/HW-components that have to be distributed to the target ECUs. The classes in this section are not directly related to the optimization and the strategy, but will be aggregated by solution objects to represent their model data. Anyway, this section will provide understanding of our mapping problem, and will be necessary to understand the concept of *local models* in the following section 3.9. All the classes of this section are shown in figure A.15 of the appendix.

To ensure type safety, the Target and TargetingItem interfaces exist, which are equivalent to their super-interface IdObject. An implementing class is required to provide the equals(..) method as well as the compareTo(..) and hashCode() methods to allow storage to HashMap collections. For expressing basic sets of targets and targeting items, there exist the TargetSet and TargetingItemSet classes that ensure type safety and that are organized as hash-sets.

Finally, the mapping between a set of targeting items and a set of targets must be expressed. For this purpose the concept of edges is applied: The EdgePK aggregates a single target and a single targeting item object, and therefore represents a primary key for an edge of the mapping.

There are two usages for the edge: First, an edge has a static status type, expressing whether it is actually mapped or not. For this purpose there exists the enumeration object StatusType, which realizes not only the types MAPPED and NOTMAPPED, but also the types INIT for initialization purposes with the semantics of "unknown", and the types ALWAYSMAPPED and NEVERMAPPED for fixed mapping information. The aggregation of an edge-PK with a status type is standardized as EdgeStatus class.

The second usage of an edge-PK is to identify dynamic changes of the mapping. Such changes are called operations, and the OperationType enumeration class provides the ADD and REMOVE types as well as the ADD_FIXED and REMOVE_FIXED types for fixed mapping information. The aggregation of an edge-PK and an operation type is available as EdgeOperation class.

In relation to both usages, the static status information and the dynamic changes, there exist two kinds of sophisticated collections, the ones that express the mapping itself using status types, and the others that manage sets of applied operations. All collections for the mapping structure and the change sets of operations provide type safety and iterators.

The information for the mapping itself is implemented as I2TAttributedFullMap. Its initialization requires a TargetingItemSet and a TargetSet. It will construct a matrix, with the status type INIT as coefficients of the matrix. The coefficients will be changed during the initial phase of filling in the fixed mapping information, and later by the

dynamic operations that change the mapping during optimization iteration. Figure 3.12 shows an example for the mapping structure.

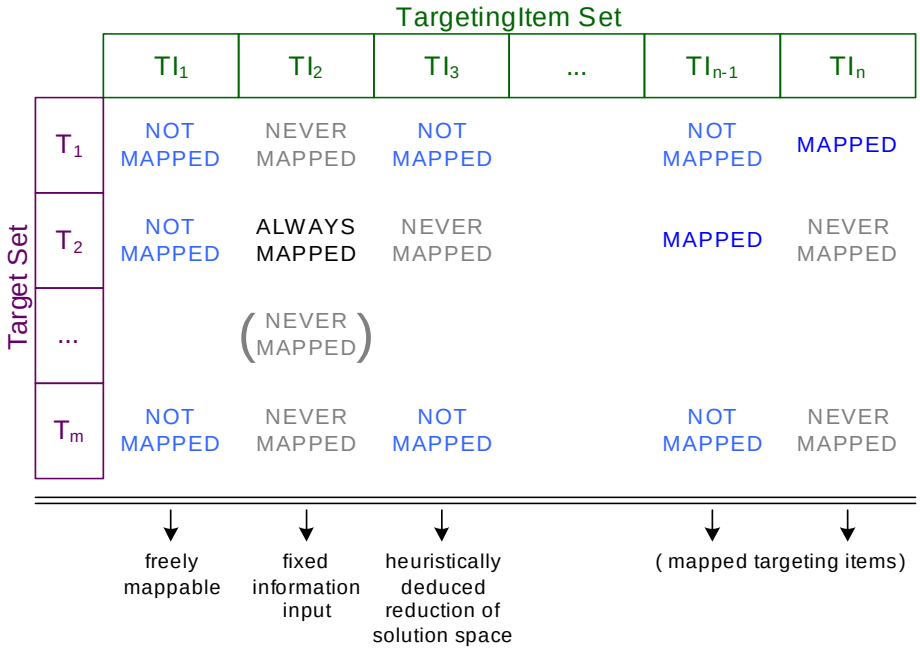


Figure 3.12.: The mapping information structure (example)

By definition it is not allowed for a targeting item to be mapped on two different targets at the same time. The I2TAttributedFullMap checks such constraints and will throw exceptions on violation. It further provides the getMappedTarget(..) for a TargetingItem parameter to find out onto which target the item is currently mapped. There are several get(..) and put(..) methods, for example for EdgeStatus objects. In preparation for the optimization algorithms also there exists the getTargetCandidates(..) method. It has a TargetingItem as parameter and will return all possible targets on which the item could be mapped, making transparent the different status types and their compatibility.

For the second usage type, the dynamic changes of the mapping, the class EdgeOperationSet is provided, which collects arbitrary EdgeOperation objects in an unordered form. In preparation for the framework there exists the PartitionedEdgeOperations collection that partitions the edge operations per target. The PartitionedEdgeOperations are used as change-objects to the update(..) notification between solutions and OEs.

The GlobalMappingSolution, as part of the de.fau.cs.i2.heuropt.domain.mapping.population package, shown in figure A.14 of the appendix, uses the I2TAttributedFullMap as model data. The GlobalMappingSolution will be further extended by the MOOVE extension in section 5, introducing the vehicle semantics. The implementations in the heuropt

packages deal with targets and targeting items in a generic way. The Global-prefix will be explained in the next section, when the *local models* will be introduced.

An EdgeOperation or a PartitionedEdgeOperations set is used as change-object by the notifyObservers() and update(..) methods between a GlobalMappingSolution and its GlobalMappingObjectiveEstimation observers. The interface of the latter ones is part of the de.fau.cs.i2.heuropt.domain.mapping.objective package, shown in figure A.16 of the appendix. The GlobalMappingSolution provides an apply(..) method that will transform EdgeOperation objects into model change and update notifications. The applyAll(..) method takes a whole PartitionedEdgeOperations set as parameter and will perform the update notification more efficiently than using multiple apply(..) invocations.

## 3.9. Local Models

This section introduces a concept of how to integrate inner/local optimization loops into outer/global optimization iterations. The motivation of the work lies within our concrete problem domain of control networks for vehicles. It is possible to apply a variant optimization for a single ECU. The problem of "variant optimization" [FRHW00] will look at an ECU, and in relation to the currently mapped SW/HW-components and their order-rates it will generate a set of variants that will substitute the original ECU. The variants optimize costs. Without explaining and understanding how variant optimization exactly works, it can be intuitively understood that the potential mappings, which the outer optimization iteration generates, have varying potential in variant optimization application. Because variant optimization is very complex on its own, it is desirable to solve it with heuristic optimization on its own. Therefore, a local optimization loop may influence objectives of the global optimization.

The framework supports an integration of target-local solutions into a global mapping solution, which means for example the update and notification between global and local solutions as well as the according global and local OEs. Additionally, the cloning mechanism must support the rebuilding of these non-trivial interrelationships. The framework does not support any decision process of "when to run the local optimization". The local optimization might introduce significant overhead; it will often not be applicable to run it for each newly generated global mapping solution and all dependent local models. The algorithm must be aware of the local models and make this decision on its own.

The target-local solution is represented by the LocalMappingSolution interface. It extends the Solution interface, but also extends the ObserverOfSolution interface, being the observer of the GlobalMappingSolution objects.

At the end of section 3.6.2 it has been indicated that the init(..) of the solution is invoked by the population during initialization (e. g. initializing the model data structures) and that the implementations for the mapping domain (the upper part of figure 3.4 in section 3.3) extend this init(..) in a non-trivial way. Now, this section

provides the description of that init(..) for such a solution that uses local models. The descriptions below apply to the abstract implementation of the GlobalMappingSolution interface, that is also part of the de.fau.cs.i2.heuropt.domain.mapping.population package.

In contrast to the global mapping solution which represents a mapping, the LocalMappingSolution just represents a single Target with a set of targeting items that are (currently) mapped to the target. The TargetingitemsPerTargetSet is used as model data, as part of the de.fau.cs.i2.util.mapping package in figure A.15 of the appendix; it is a partition of the I2TAttributedFullMap used by the global solution.

The global optimization will change the set of targeting items for each target, because the mapping-edges are permutated. During a local optimization, e. g. the variant optimization, the LocalMappingSolution represents the model/solution, so its set of targeting items must not change. The part that is changed for the local optimization is for example the set of variants. For now, it is not important what is actually optimized by a local optimization, but that the model of a local solution is a set and not at all a map. The set is basically represented by a TargetingitemSet. In the first instance, a local solution is not conceptually related to "mappings". But our goal is to integrate local solutions as the aggregates of a global mapping solution, in order to synchronize the local models to the state of the global optimization. Just by doing so, the local solutions are set into relation to the global mapping solutions.

Figure 3.13 provides an exemplary overview of the structure and interrelationship that will be the result of the initialization. The upper half of figure 3.13 shows a solution X with its registered OEs. As examples for the OEs we just use the concrete ones from the MOOVE project (C: costs, BL: bus load, W: weight, QC: quiescent current, HWA: hardware availability). The HWA in figure 3.13 is a constraint and in the same role as the other objective estimations. The lower half of the figure represents the local models and their relation to the global objects. The arrows indicate access availabilities.

The first thing that should be noticed in figure 3.13 is that there are two possible paths to the local OEs. The update notifications of the mapping changes are propagated from the global solutions via the local solution to the local OEs, because this update path reuses the standard interfaces, allowing the application of optimizer components, configurators and repositories for local optimization. The access path from the global OEs to the local OEs is provided, because there are many (global) OEs that represent plain aggregations of information that is evaluated per target. In order to reduce the overhead for a combined global/local optimization, the global OEs should delegate as much calculations as possible to the local OEs.

If the global OEs are advised to aggregate the information from the local OEs, the framework must ensure that the mapping change information is propagated to the local OEs first. For this purpose, the GlobalMappingSolution reimplements its role as observable, so that it manages its two kinds of observers separately. On the one hand the getLocalSolutionVector() provides access to the local solutions – in addition to the getObjectiveEstimationVector(). On the other hand, the local solutions are notified

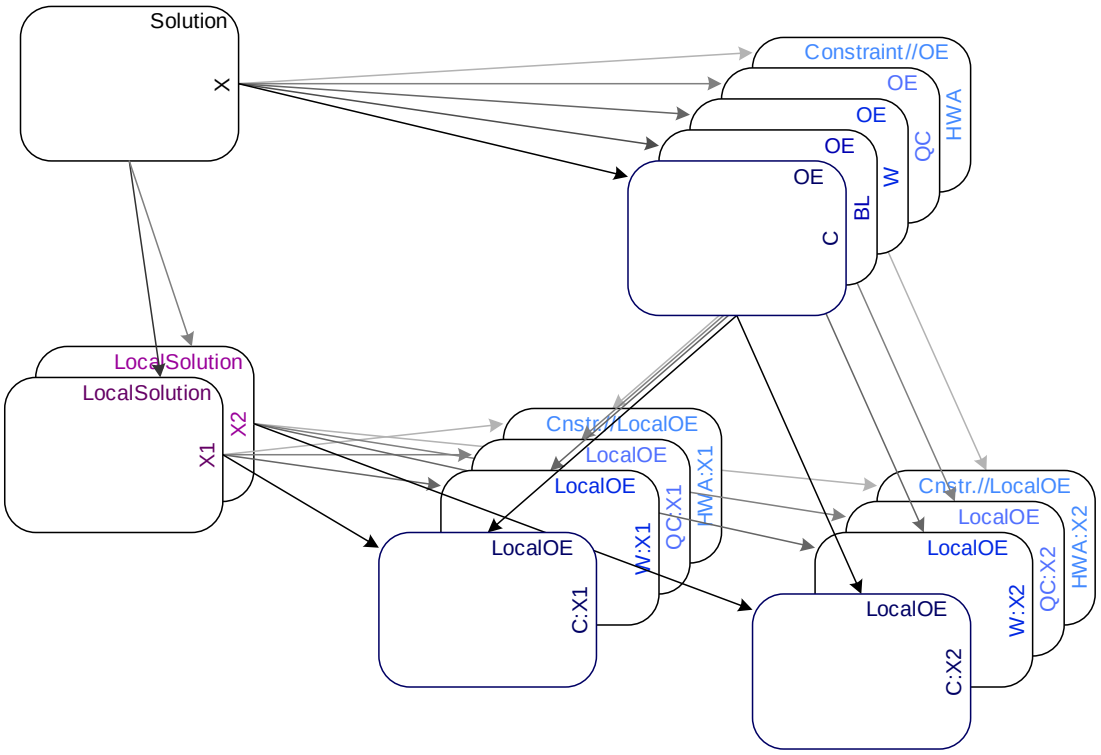


Figure 3.13.: Connection structure between solutions and objective estimations

before the (global) objective estimations. Because the notified local solutions will immediately update their model and notify their local OEs, the global OEs will be notified after their local OEs.

A GlobalMappingObjectiveEstimation can decide to use no local OEs. It must indicate this by returning true for the isPureGlobal() method. In figure 3.13 the example for this is the busload (BL), because the busload only needs the global view under any circumstances. The isPureGlobal() is mainly used during the creation process for the local OEs, skipping such global OEs. It is also possible to provide a set of purely local OEs[4] to the directive of the MappingOptimizerConfigurator, in case that local objectives or constraints need to be evaluated by the local optimization, but which must not influence the global optimizer's decisions.

In section 3.6 it has been explained how the population uses OE-factories to create and register the (global) OEs as observers to a solution. During the now discussed init(..) of the (global) solution, the global OEs are already available. So, we need to discuss the creation of the local OEs: The factories for the local OEs are the global

[4]An example for purely local OEs has not been integrated in the example of figure 3.13, in favour of lucidity.

OEs itself. For this purpose they have to implement the reduced RawObjectiveEstimationFactory interface. The ObjectiveEstimationFactory interface actually extends the raw one. The first difference between both is that the raw interface does not enforce an initialization, because local OEs that are factorized by a global OE may not support more objectives than the currently supported ones from their global OE, and the global OE will already be completely initialized. The second difference is that the global OEs in the role of a RawObjectiveEstimationFactory will not be registered as components to the repository, as the ObjectiveEstimationFactory objects have been.

The population will create a local solution/model for each target of the declared target set. For each target, it will iterate over the vector of its global OEs, skipping the ones with isPureGlobal() set true, and requesting them to create(..) the local OEs.

After a global OE as factory has created a local OE, the same two-phased initialization for the local OE is performed by the solution, as has been done for the global ones by the population in section 3.6.2: The solution invokes two kinds of init(..) on the local OE. The first one for the local OE in its role as observer, whereupon the local solution is provided as parameter, for which the local OE registers by using addObserver(..). The second init(..) allows the local OE to create its specific structures – the mapping directive is extended so that domain specific parameters can be passed arbitrarily by a HashMap.

Although the global OEs create the local OEs, they will not automatically store the reference of the local OE, because this would not conform to the role of a factory. Therefore, the global solution immediately invokes register(..) at the global OE, providing the just created local OE as parameter. Now the global OE stores the reference to the local OE in its collection, for later aggregation of all the local quality ratings into a global rating for the mapping solution. This separation of concern between create(..) and register(..) will be necessary during the cloning mechanism. The above steps are illustrated in the top half of figure 3.14.

It has been indicated before that the mapping directive additionally allows the declaration of purely local OEs. These are not created by a global OE, but by a standard OE-factory component. The GlobalMappingSolution will create these local OE the same way as the global OEs had been created in section 3.6. Therefore, this is only sketched in figure 3.14.

After the creation and initialization of the local OEs, the local solution is now also initialized according to its first role. In contrast to the global solution which has two roles, the local solution has three roles: The first has been the role of the model for the local OEs. The second one is the role of an observer/view in relation to the global solution. For this initialization purpose, the global solution invokes init(..) with itself as Model-parameter. The local solution will store the reference to its model as member, and uses the global solutions's addObserver(..) to register as observer for global solution/mapping changes. The third role of the local solution is again the intrinsic role of a solution, this means that the global solution invokes the second init(..) with a domain specific parameter HashMap, during which the local solution

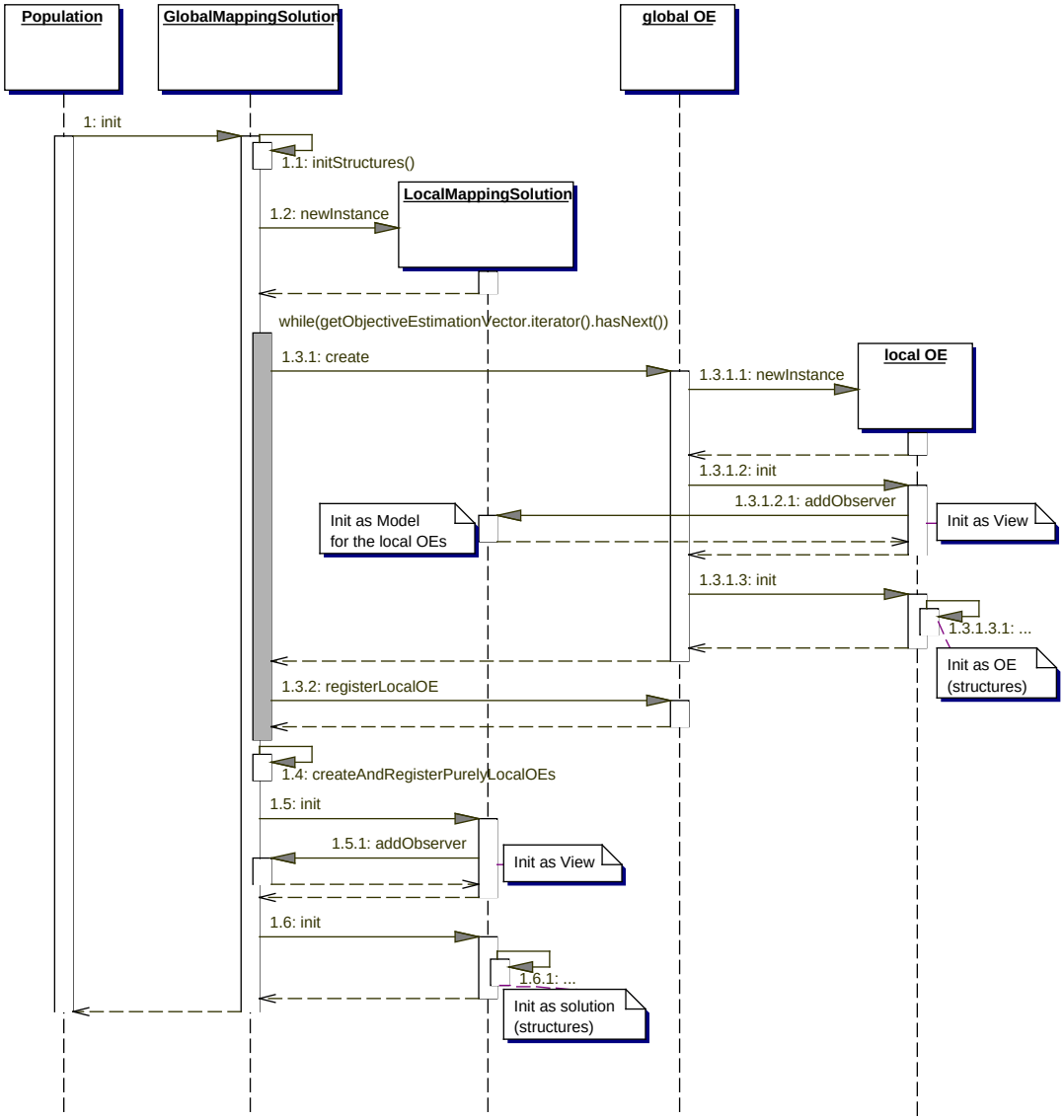


Figure 3.14.: Instantiation and initialization of the local objective estimations and the initialization of the local solutions

creates its special data structures. The complete process is shown in figure 3.14. This results in the structure and interrelationship that had been shown in figure 3.13.

## 3.10. Cloning

The cloning mechanism is based on the DeeplyCloneable interface in the de.fau.cs.i2.util package, which extends the standard Cloneable interface. Its implementation requires the cloneDeeply() method. The semantics of clone() is usually not to clone referenced objects, as for example for the collections in the java.util package. The cloneDeeply() method means that the cloning will propagate as deeply as possible, wholly cloning the complex structures.

The DeeplyCloneable interface is extended by most interfaces like the SolutionPopulation, the SolutionSet, the Solution and the ObserverOfSolution with its sub-interfaces ObjectiveEstimation and LocalMappingSolution. For example, the deep cloning of a population leads to the cloning of the solution set with all its solutions. For each solution the model data is duplicated. Each OE is cloned and the clone is registered to the new solution clone. Even the more complex recreation of the local models (and local OEs) is completely rebuilt.

The cloning process resembles the initialization process as much as possible. A divergence is introduced by the non-existence of the client role during cloning: For example, the relation between solution and objective estimation is originally established by the population as client. Because a solution aggregates all OEs, and because its cloning must be self-contained, the solution takes the additional role of the client during cloning, between itself and the OEs.

The major reason that allows the resembling of the initialization processes has been the consequent separation of concerns and roles. The clarity has been verified by reverse engineering the code basis into sequence diagrams, using the facilities of the Borland Together [Bor05] CASE-tool.

## 3.11. Utilising the Framework

Templates for a population class and a solution class exist in the de.fau.cs.i2.heuropt.template package. Both are available in the appendix as listings A.1 and A.2. Both template classes extend the abstract implementations from the de.fau.cs.i2.heuropt.kernel.population package.

The most important functionality that is not standardized by the framework is the access on the problem description data. As will be seen in section 5, MOOVE extends the directive and configurator so that two database IDs are declared for a problem, deriving from them the sets of targets and targeting items. So the configurator's createDomainSpecificPropertiesForSolutionPopulation() can be used to create the domain

specific properties HashMap that will finally be passed to the population as parameter of its initDomainSpecific(..) method.

The solution also has an initDomainSpecific(..) method that should be instrumented to create the domain specific data structures for the optimization problem. The most important task for the solution will be the implementation of operations on their problem representation; for example the mutation or cross-over operations for genomes. These operations' signatures can be unrestrictedly defined, but their kind of changing the underlying data structures must use the inherited apply(..) and applyAll(..) methods: Because all these operations will change the underlying data structures, and because the registered OEs must be informed about such changes, the protected applyOnSolutionDataStructure(..) has to be implemented. The AbstractSolution already implements the apply(..) and applyAll(..) methods, which have a ChangeObject or a ChangeObjectSet as parameter. Both methods will invoke the abstract this.applyOnSolutionDataStructure(..) and then propagate the change objects to the registered observers. Each solution just has to implement the operation on its data structures, but has not to deal with the registered observers. For this, all operational methods just are required not to apply changes on the structure directly, but to create change objects, passing them to the inherited apply(..) and applyAll(..) methods.

The population's most important task is to implement the regeneratePopulation() method. Like the template in the appendix shows, the population basically clones its solution set and applies the operational methods that are available for the solution class. During the initialization phase of the framework, the population methods makeRawsolutionToTemplate(..) and makeTemplateToValidInitialSolution(..) are invoked. It is not required to actually implement their documented semantics, for example it is not required to ensure the validity of initial solutions, as long as the regeneratePopulation() will sometime create valid solutions.

The population must also provide a default solution class by getDefaultSolutionClass(). Population and solution should extend the clone() and cloneDeeply() methods. For the objective estimations there is no template. They just have to implement update(..) and calculateQualityRating(..) methods.

After all, the templates in the appendix demonstrate the major goal of the framework: The developer of an optimization strategy is released from many organisational tasks, becoming able to concentrate on the optimization strategy.

## 3.12. Summary

Section 3 explained the architecture of the HeurOpt framework. Section 3.1 described the basic objects and components during initialization, and section 3.2 set them into context to the client, describing highest-order iteration.

After these introductions, the internal HeurOpt layers and modules were presented in section 3.3. First, the population was discussed in section 3.4. Its aggregated

solutions were the topic of section 3.5, focussing especially on their role as model according to the model-view paradigm. The observers of the model are the objective estimations, discussed in section 3.6. Because they are created by factories, as are the optimizer components, their special initialization process was described, as well as their extension by the constraint interface.

Next section 3.7 outlined the existing Pareto front implementations that were created by other project members. A model data structure for mappings was introduced in section 3.8, based on two sets of targets and targeting items. Local models for individual targets and their interrelationship with the global mapping optimization objects were described in section 3.9.

Finally, the cloning mechanism that clones complete solution and population structures was outlined in section 3.10, before section 3.11 exemplified how the framework can be used for other optimization problems or new optimization strategies.

# 4. The Data Model for Vehicles

Under consideration of the problem outlined in section 2, this section gives an analysis and the data model for the use cases, objectives and constraints that belong to control networks in vehicles. The following objectives will be of special interest: costs, bus load, cables and quiescent current.

## 4.1. Model Creation

The model was designed with pristine entity-relationship diagrams (ERD) for relational modelling. We decided very early against object-relational modelling, because ERD is widely accepted and understandable, even for employees from the sales and distribution department, who will have to provide some heuristics in relation to order-rates. The ERD was modelled with the free Clay [Azz04] plug-in for Eclipse [Ecl05].

For programmatic access to the DBS a transparent JDBC access with Java class generation for the relation tuples was desired, and several tools for SQL to Java reverse engineering have been evaluated. Finally, Torque [Apa05a] was decided to be most suitable[1]. It is an open-source O/R mapping framework and persistence layer by the Apache group that provides model-based Java class generation with integrated services like caching and connection pooling, and a generic JDBC-based reverse engineering of existing relations into a Torque model.

The figures of the data models in this section have been created with Visio [Mic03] and express tables with a rectangle: The name of the table is at the top of each in a gray box and its columns below. Each column has tags for *primary keys* (PK), *foreign keys* (FK) and *unique* (U) constraints, if applicable. The data model is part of the MOOVE project, discussed in section 5.

The data model has been designed to be as generic as possible, in order to be sustainable for future requirements. Therefore, there are quite complex semantic relationships between attributes that require a human expert to input the data. For the application in productive environments, it will be required to provide an graphical interface, based on an expert system on its own and supporting the users. At the moment such a system is not available and the data is maintained manually.

[1] As alternative to Torque, sql2java [Sou04a] and the JDO-compliant XORM [Sou04b] framework had been considered, but Torque is stable, feature-rich and future-proof.

## 4.2. Distribution Mapping

A distribution mapping is not directly applied between the SW/HW-components and the ECUs. Due to redundant information between several vehicle series an abstraction layer was created to reduce this redundancy, which has the advantage that both components as well as ECUs can be instantiated multiple times in the same or different series. Components are instantiated as a *component instance* per vehicle series. ECUs are represented as *network nodes*; they are not only assigned to a vehicle series but also to topologies and networks. So our framework maps component instances to network nodes, as shown in figure 4.1. For convenience, we still talk about components and ECUs, if the context does not require an exact differentiation.

The distribution mapping between the two sets of component instances and network nodes is represented as a set of edges. For each application of an optimization there will be generated many mapping solutions – usually the set of the Pareto front after the last iteration will be stored. These solutions are identified by the mapping table with its ID. The optional_runtime_id allows identification and selection of the Pareto front. The runtime ID is optional so that human generated solutions can be stored with the ID being null. These human provided mappings can be read into the framework facilities to evaluate the mappings according to arbitrary objectives.

The edges itself are stored by the componentinstance_to_networknode_relation table, representing the raw mapping information. In addition to the identifiers of mapping, network node and component instance there is a variant_number, which defaults to 0. Based on order-rates of the customers, it is possible to generate variants of ECUs in order to decrease the average costs of the installed ECUs. Details about the variant optimization can be found in [HKKK05]; the model handles its use case.

A component_instance is related to its component and the vehicle series (vehicletype_ID). For HW-components, the location references an installation position, like it is also provided for all network nodes. The distance of the locations between the ECU and its mapped-to HW-components are required for cable length calculations. The calculation of the distance can be a plain Euclidean one, or can use complex models and algorithms that approximate the cable duct structure. The generic representation of cable duct structures is not yet standardized and not part of the data model.

An optional reference from a component instance to a component_unit forces the mapping algorithm to *atomically* map the unit's component instances to an arbitrary ECU. For example, an airbag must always have its SW- and HW-components on the same ECU, due to safety reasons.

Looking at the component relation, an ID-reference to a component_type can be found. For example, for HW it is "accessor", "sensor" or "actuator"; for SW it is "functional", "adaptive" or "firmware". There is also the "compound" type for assemblies of HW- and SW-components that are not wanted to be modelled in differentiated components; this obviously reduces the mapping algorithm's degree of freedom, because this compound component is atomically mapped by definition.

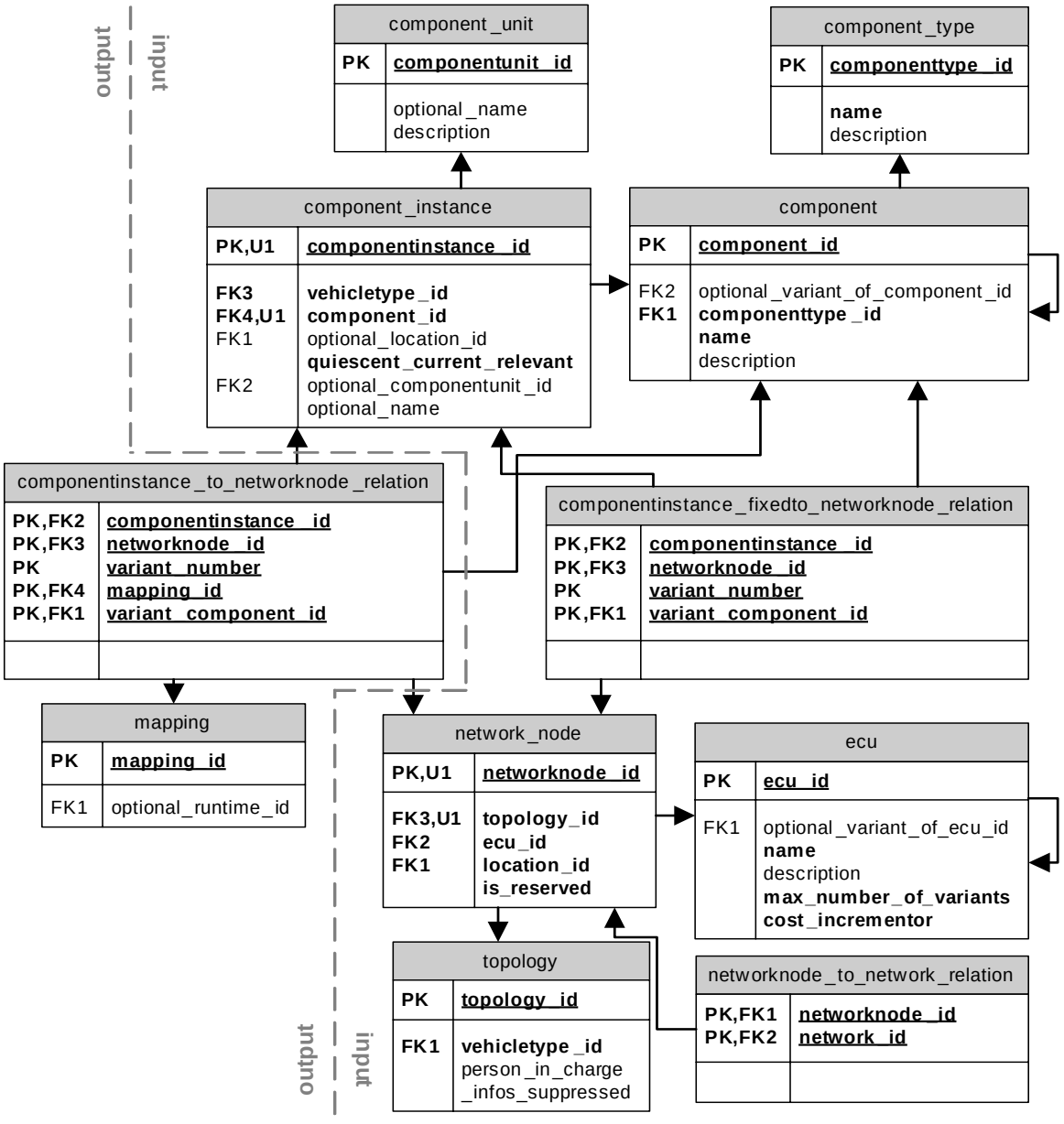


Figure 4.1.: Modelling the distribution mapping

The network_node references its ECU and is integrated into a topology. It is required to have a location for cable length calculation, as outlined above. A topology[2] can be understood as problem domains of vehicles, like body electronics, power train, chassis, or human machine interface. However, they can even have a finer granularity, and are unique per vehicle series. The networks that are available to a network node are provided by additional relations, discussed in section 4.7.

Finally, the componentinstance_fixedto_networknode_relation applies to the use case when human experts want special component instances mapped to defined network nodes, and only a semi-automated distribution mapping is desired.

## 4.3. Objectives and Quality Ratings

Aside from the mapping information with its edges between component instances and network nodes, the evaluated objectives and the according quality ratings also need to be stored. The relevant relations are shown in figure 4.2.

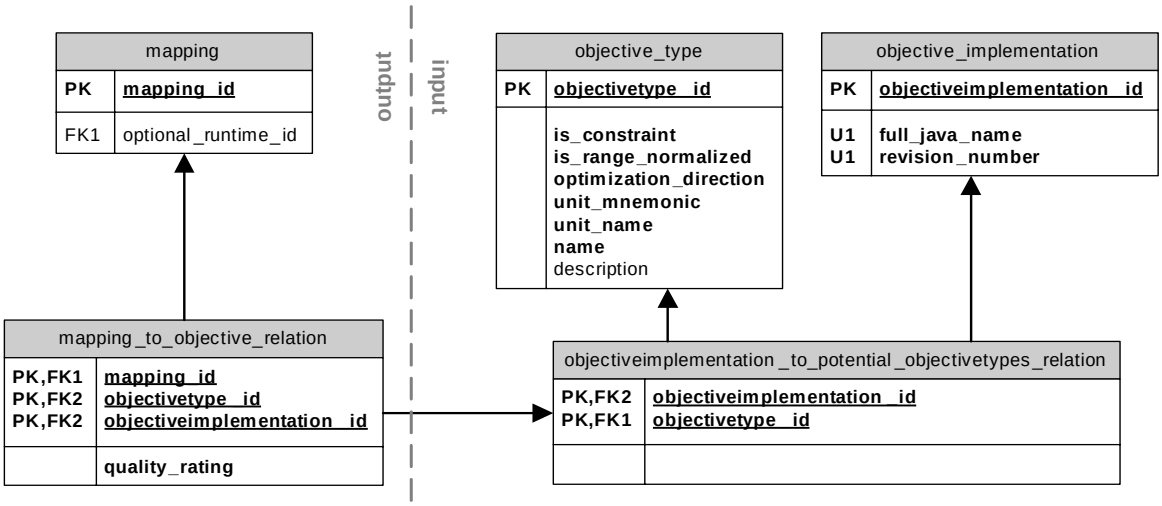


Figure 4.2.: Objectives and quality ratings in relation to mapping solutions

Objectives are represented by the objective_type relation. The is_range_normalized attribute provides information whether the range is normalized to the interval $[0; 1]$, which can be done if a minimum and maximum boundary for the range exists. If a tabu violation is applicable for the objective, the is_constraint attribute should be true. The threshold or conditions for a violation are not part of the data model, either because the threshold might be adopted during the optimization iterations, or because the condition could be quite complex. Furthermore, it is quite important whether small or big quality rating values are better, with the optimization_direction being "min" or

[2]The _infos_suppressed attribute substitutes arbitrary meta information about a topology.

"max". The unit of the quality rating values should be explained with the two unit_ attributes, e. g. whether it is a ratio or a counter and its semantics. The objective in general is described by the name and description.

Additionally, there is an objective_implementation relation, allowing several objective estimation implementations for the same objective (type). The integer identifier as PK is used for performance reasons, but the fully qualified class name is required to be unique on its own. Support for different revisions of a class implementation is provided by the revision_number attribute; it defaults to "0" and extends the unique constraint.

An objective estimation implementation is allowed to handle several objective types. The objectives that are potentially supported are stored in the objectiveimplementation_to_potential_objectivetypes_relation. The specific objective types and the used objective implementation that have been evaluated for a mapping solution can be stored in the mapping_to_objective_relation. The quality_rating attribute stores the value.

## 4.4. Resource Consumption

The resource consumption of components is another important part of the data model, besides the distribution mapping. The relevant relations are shown in figure 4.3 and described in the following.

ECUs provide resources. There is a special ECU (ID = 0) that is named "the environment". For example, this environment provides cables. It also reduces modelling input by allowing the specification of default values. In fact, the environment is not considered as an ECU of its own[3]. During the resource availability calculation for an arbitrary ECU, the environment's information is entirely copied and then enhanced or overwritten by the ECU-specific information. Because it is still a static operation and information, it can be provided by a (persistent) view.

This special copy-treatment of the environment enables interesting use cases. An important one is the consumption of cables. Instead of treating the offering and consumption of cables in a special way, the environment provides an unlimited amount of cables, with the price being represented as influence on the objective "costs". The amount of connections is constrained by the pin resources. Although cable estimation requires the special length multiplier that is calculated by the location distances, the environment allows easy integration and generic evaluation of globally available resources, as well as their influence on objectives. Similar resources are related to energy and quiescent current. There are even more use-cases for the environment beyond global resources, which will be discussed below.

A resource is measured in its unit_type, being relevant for the consumed and provided units. It is recommended to use the *Système International d'Unités* (SI) [Bur60] for all these types. Standard resources are RAM or ROM as well as several kinds of power

[3]The environment is therefore not allowed to be instantiated as a network_node, which can be enforced by SQL check constraints.

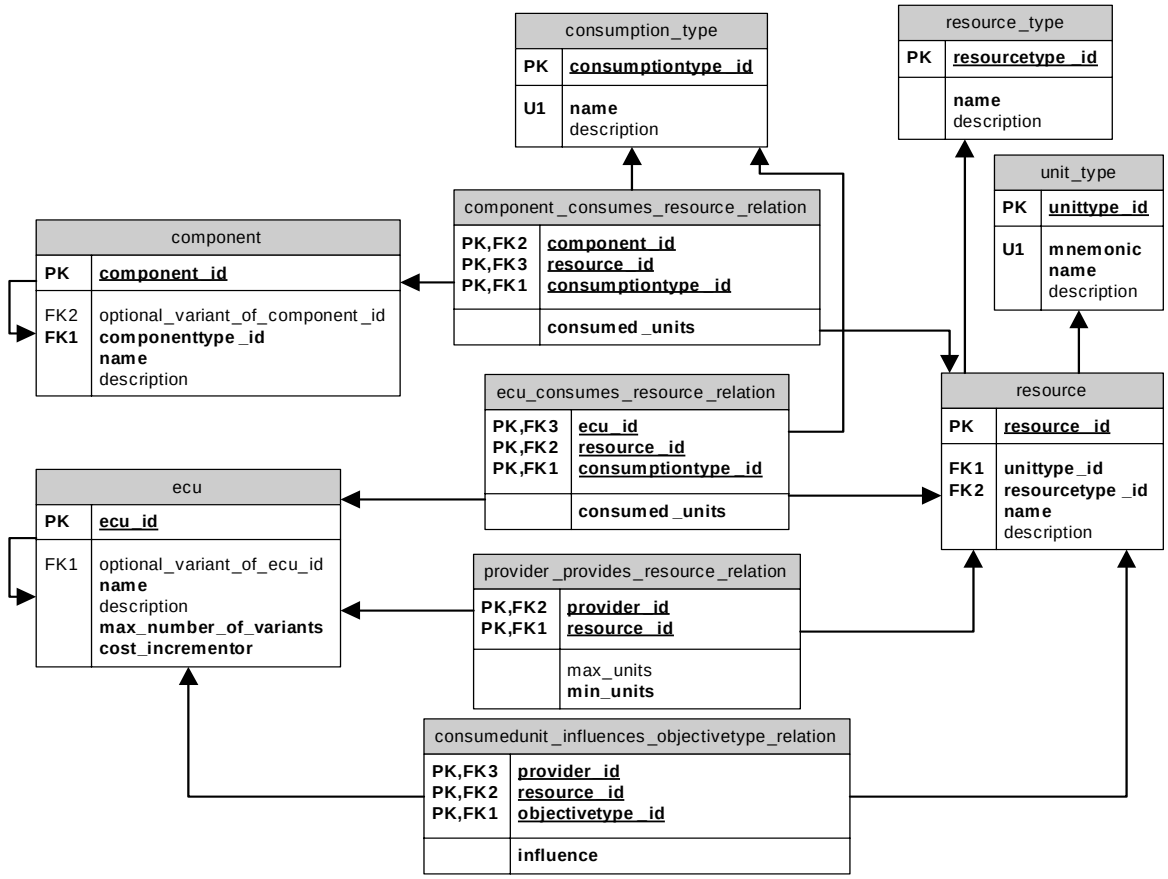


Figure 4.3.: Offering and consumption of resources

electronics, pins, cables or electrical power consumption. But interrupts also can be considered as resources, as could be CPU load consumption[4].

The model easily allows to add arbitrary resources. In particular ones that do not influence the component distribution can be supplied by the environment, deepening the cost structure without adding much computational overhead.

Resources have a type: For example there exist several kinds of cables. Each of them is modelled as an individual resource, but all cable-resources have the same resource_type "cable".

The offering of resources is represented by the provider_provides_resource_relation. The provider_ID is equivalent to an ECU_ID. It has been renamed to point out the requirement for the copy-treatment of the special environment-ECU. ECUs may consume resources that are offered by itself. For example, the equipped memory is consumed

[4]Unfortunately, measurement and specification of CPU consumption is quite controversial, particularly for software modules that will be used on varying hardware. We have neither input data for it, nor dared to specify a unit type.

by the OS and the middleware, without relation to any SW-components that are to be distributed. The relation for ECU self-consumption is the ECU_consumes_resource_relation. In contrast to the _provides_ table, it uses the ECU_ID directly, because the environment does not consume resources. ECUs provide resources in a minimum (min_units) and maximum (max_units) amount. The minimum value is used during objective estimation if the accumulated requirements of the components mapped to an ECU is below the value, for instance reflecting a minimal shippable equipment for an ECU. The maximum value is a constraint that must not be violated by the accumulated component requirements.

ECUs or components _consume_ resources at a number of consumed_units. Resource consumption may have an influence on objectives, being represented by the consumedunit_influences_objectivetype_relation table. The influence depends not only on the resource, but also on the providing ECU. For example, memory can have different costs on different ECUs. If no influence is specified for the particular ECU, then the influence will be taken from the environment; only if this one does not exist either, the resource will have no influence on this objective.

The consumption of resources, either by ECUs or components, is attributed by a consumption_type. The consumption type is related to the feature order-rate or the ECU installation-rate [HKKK05]. The resources that are consumed by an ECU itself and the ones that are consumed by HW-components on the ECU, like power electronics or pins, always have the type "ECU installation-rate dependent". The SW-components, too, consume resources like memory with the type "ECU installation-rate dependent". In contrast, sensors and actuators consume resources like motors or cables with the type "feature order-rate dependent", because they are only installed if ordered.

In fact there are more kinds of consumption types possible, because abstract resources like "licensing costs" could require types in relation to the used kind of apportion. We do not use such at the moment.

As we have discussed above for the airbag, we need its SW- and HW-components on the same ECU. An additional use case is of interest: Two independent airbag controls could be installed for reasons of redundancy and safety. Obviously, it is required to have the redundant components on separated ECUs. This can easily be achieved by creating an abstract resource that mirrors the airbag control. It does not have any "influence" on an objective, but is "consumed" by the airbag component. Offering this new resource by the environment (min_units = 0; max_units = 1) will copy it to all ECUs, as we have discussed above – so there is not much modelling expense to the introduction of this resource. This will automatically force the algorithm to map both airbag controls to different ECUs. Because the framework treats and resolves resource availabilities in a generic way, the implementation has not to be touched. This principle can be applied for similar problems like safety levels.

## 4.5. Vehicle Series and Features

The relationship between the features that are available for a vehicle series and the component instances that need to be distributed will be explained in this section. The customer order-rates for features and the structure and frequency of ordered feature combinations have a strong influence on the objective "costs", which will be explained by example in section 4.5.3. To begin with, figure 4.4 shows the basic relations.

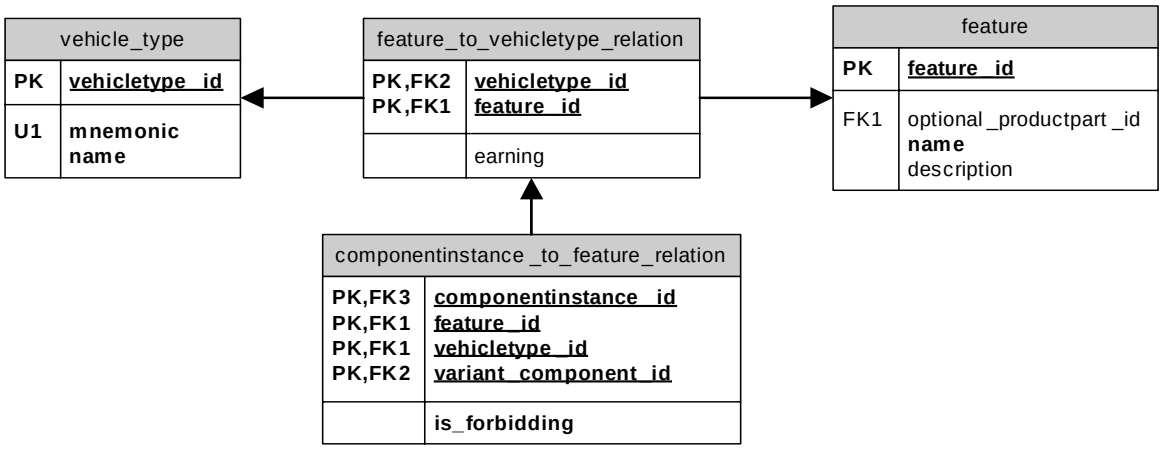


Figure 4.4.: Vehicle series, features and required component instances

A vehicle series is represented as a tuple in the vehicle_type relation. Aside from the artificially generated integer identifier, vehicle manufacturers often have unique mnemonics for the series.

The feature relation simply allows to define the features and to provide their semantics in form of name and description. The optional_productpart_ID references the product_part relation, shown in figure 4.5. The concept of product_part and product_family is commonly used by manufacturers to classify features for the sales or procurement department.

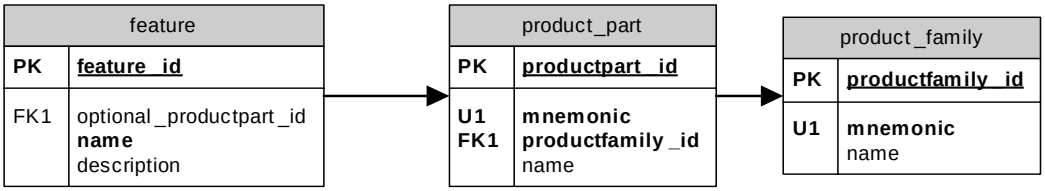


Figure 4.5.: Product part and family

The feature_to_vehicletype_relation defines the set of features that are available for order. The earning attribute can be used to specify the profit for the feature individually for vehicle series.

The set of component instances that are actually required to implement a feature for a specific vehicle series must be provided in form of the componentinstance_to_feature_relation. The attributes variant_component_ID and is_forbidding need to be explained in some detail.

The variant_component_ID is related to variants of features[5]: For example, there might be different features for different radio types in diverse countries. All these radio features can refer to the same component instance, but provide different component variant IDs. The resources required by the component variants, in supplement to the base variant, will not be required at the same time, but are mutually exclusive. In the end, the component instance with all asked for component variants is deduced to a set of resource requirements that will force the distribution optimization to map the component instance to an ECU so that manufacturing for different countries is supported.

The attribute is_forbidding is a preliminary substitution for the more accurate application of a feature tree model. Feature trees are independent of the vehicle manufacturing, but originated in *feature-oriented domain analysis* (FODA) [KCH+90], being used for development of software product lines and the according definition of products and configurations, describing the possibilities of a product line. They structure features and possible combinations hierarchically. Further information about feature trees can be read in the well-established book from Czarnecki and Eisenecker [CE00].

In the appendix, figure B.1 shows a database model for feature trees. One problem with feature trees is that models for feature attributes, in addition to the hierarchy itself, are yet subject to research. For vehicles we require attributes like order-rates and the frequency of combinations. Another problem has been that the manufacturers do not currently use feature trees to structure their features, although there is quite some interest [TH02, KFAK05]. Therefore, the MOOVE project does not use feature trees at the moment. Instead, we use the basic "requires" and "mutex-with" composition rules that allow the representation of a subset of the feature tree hierarchy [Rie03]. The is_forbidding attribute defaults to false, so that a normal tuple in the componentinstance_to_feature_relation equals a "requires" composition. If the is_forbidding attribute is true, then for this specific vehicle series the feature is in mutual exclusion to all other features that require the according component instance plus variant component combination.

### 4.5.1. Customer Orders

It is necessary to store information about ordered vehicles as preparation to calculate feature order-rates and ordered combinations. Figure 4.6 shows the relations for customers and their ordered vehicles on top of the already known axis between vehicle_type and feature.

[5] Variants of features must not be confused with variants of ECUs [HKKK05].

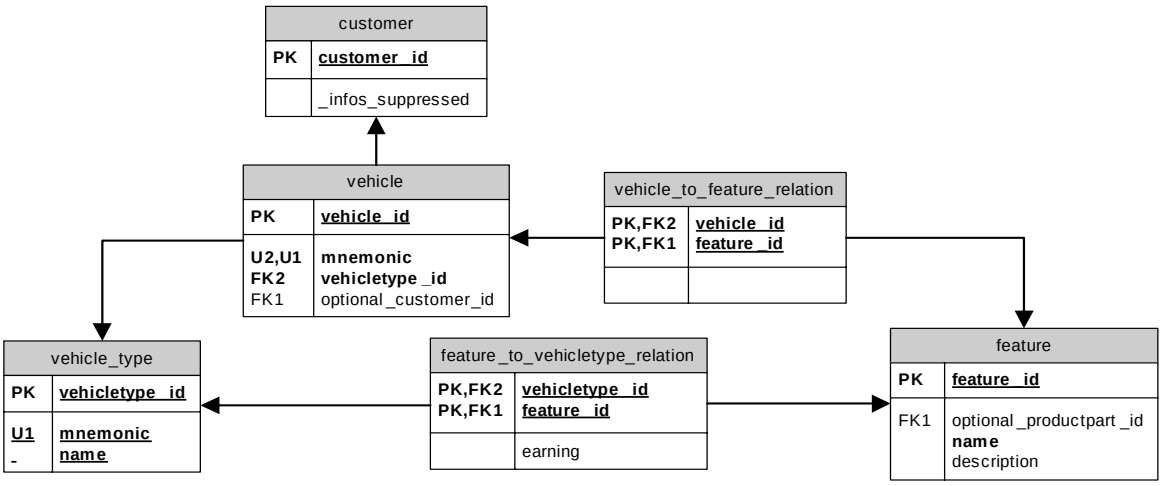


Figure 4.6.: Customers and their orders

The tuples in the vehicle relation represent individual vehicles. Usually, a manufactured vehicle gets a serial number, which can be stored as mnemonic attribute. The feature set of installed equipment is stored by the vehicle_to_feature_relation. The manufactured vehicles might be set into relation with the database information from the sales and distribution department, as it is outlined by the customer relation[6].

The vehicle relation actually keeps several million entries. Using the vehicle_to_feature_relation it is easily possible to calculate the order-rates of features. But as we will see in section 4.5.3, it is more efficient to aggregate the vehicles into partitions with the same feature combinations.

The relations vehicle and vehicle_to_feature_relation in combination can be understood as a table, with the dynamically growing number of vehicles from top to bottom, and with a dynamic number of features from left to right. Figure 4.7 shows an example, which will return in the next subsections. The exemplary features are the *electronic stability program* (ESP), *power windows* (PW) and *electric rear windows blind* (ERWB).

| | $F_{ESP}$ | $F_{PW}$ | $F_{ERWB}$ | ... |
|---|---|---|---|---|
| $Car_1$ | x | x | | |
| $Car_2$ | x | | x | |
| $Car_3$ | x | x | | |
| ... | | | | |
| $Car_v$ | x | x | x | |

Figure 4.7.: Example for customer orders

[6]The _infos_suppressed attribute substitutes arbitrary available information about a customer.

### 4.5.2. Partitions of Feature Combinations

The next step is to analyse the feature sets of the individual vehicles and to aggregate them into partitions with the same feature combinations, using the millions of entries from the vehicle relation. Figure 4.8 shows the relations for the vehicle_partition and their set of related features below the axis between vehicle_type and feature.

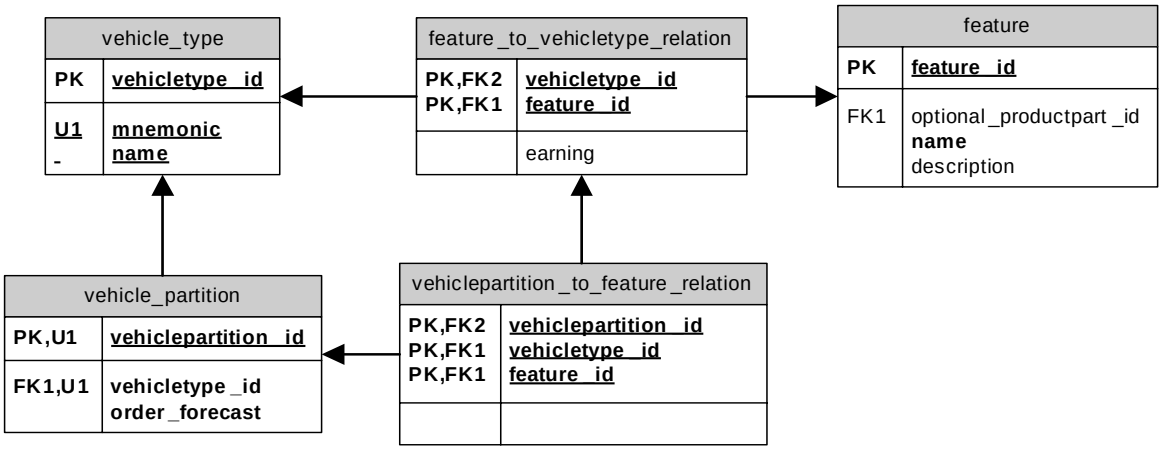


Figure 4.8.: Feature combinations and order-rates

The vehicle_partition relation references its vehicle series and stores the cardinality of its vehicles as order_forecast. Note that the optimization is particularly applied to new vehicle series. As it has been indicated above, the forecast for already manufactured features will primarily be calculated by aggregating the vehicle information from old vehicle series. But for new features there are no existing values, so that a human forecast is required. If no forecast is possible, then the new feature is simply added to all partitions. For this reason, the vehicle_partition and vehiclepartition_to_feature_relation are fully-fledged relations, instead of persistent views on the vehicle and vehicle_to_feature_relation.

Again, the relations vehicle_partition and vehiclepartition_to_feature_relation in combination can be understood as a table: From top to bottom the partitions with the same feature combinations are listed, and the features go from left to right. In contrast to the vehicle relation, the number of tuples in the partition relation is relatively constant. Using more orders will just change the number of vehicles in the order_forecast attribute, with a slight influence on the order-rates.

Figure 4.9 shows a simplified example, using the same features as in figure 4.7. The values will be reused in the next subsection, when it is explained how a different distribution of components has influence on the ECU installation-rate, based on the feature combinations and their order-rate.

| | #cars | $F_{ESP}$ | $F_{PW}$ | $F_{ERWB}$ |
|---|---|---|---|---|
| Partition1 | 2000 | | x | |
| Partition2 | 1000 | x | x | |
| Partition3 | 1000 | | x | x |
| Partition4 | 500 | x | x | x |
| Partition5 | 500 | x | | |
| Feature order-rate | | 40% | 90% | 30% |

Figure 4.9.: Example for partitions of feature combinations

### 4.5.3. Feature Combination-rates and ECU Installation-rates

This section provides a simple example for the influence that order-rates for feature combinations have on ECU installation-rates. The influence as well as the application of the matrix from the previous subsection had already been researched by my co-advisor Bernd Hardung. Together we just defined the data model and worked out the details. He continued with cost optimization in form of the variant optimization [HKKK05], which is not part of this thesis.

The ECU installation-rate equals the ratio of vehicles in which a specific ECU has actually to be installed, because if an ECU solely contains components that are related to features which are not needed/ordered for the vehicle, the ECU will not be installed, and its costs are saved. For a given mapping, all components that are mapped to the ECU have to be selected. Every component belongs to one or several features. Therefore, a feature set that is covered by the ECU can be gained.

Figure 4.10 shows the pivoted version of figure 4.9. The exemplary features are the same as in figure 4.7: The *electronic stability program* (ESP), *power windows* (PW) and *electric rear windows blind* (ERWB). The installation-rate $i(\text{ECU}_x)$ for an ECU is calculated as follows: All the features that are covered by the ECU have to be selected, and the rows have to be merged, in order to sum all the order_forecast numbers for the relevant partitions.

| | Partition1 | Partition2 | Partition3 | Partition4 | Partition5 |
|---|---|---|---|---|---|
| $F_{ESP}$ | | x | | x | x |
| $F_{PW}$ | x | x | x | x | |
| $F_{ERWB}$ | | | x | x | |
| order_forecast (#cars) | 2000 | 1000 | 1000 | 500 | 500 |

ECU installation-rates $i(\text{ECU}_x)$:

$F_{ESP}$ : $i(\text{ECU}_x) = 2000/5000 = 0.4$
$F_{PW}$ : $i(\text{ECU}_x) = 4500/5000 = 0.9$
$F_{ERWB}$ : $i(\text{ECU}_x) = 1500/5000 = 0.3$

$F_{ESP}+F_{PW}$ : $i(\text{ECU}_x) = 5000/5000 = 1.0$
$F_{ESP}+F_{ERWB}$ : $i(\text{ECU}_x) = 3000/5000 = 0.6$
$F_{PW}+F_{ERWB}$ : $i(\text{ECU}_x) = 4500/5000 = 0.9$

$F_{ESP}+F_{PW}+F_{ERWB}$ : $i(\text{ECU}_x) = 5000/5000 = 1.0$

Figure 4.10.: Example for feature combinations and installation-rates

Now an example is provided in figure 4.11 that demonstrates the shifting of the installation-rates if the mapping between components and ECUs is changed. The

example assumes three available ECUs in the vehicle: one in the central part, and one on each side of the car. Each of the three considered features consists of a whole assembly of SW/HW-components, e. g. a set of sensor, actuator and the controlling SW-component. For the power window feature we assume two such assemblies, one for the right door and one for the left door. For the sake of simplification, the assemblies are always atomically mapped to an ECU. Furthermore, we assume that the four assemblies have similar hardware availability requirements, and that the ECUs are equipped equally. Therefore, we can formulate a constraint simply as "at the utmost, two SW/HW-component assemblies can be mapped to one ECU". If no such constraint would exist, the obvious best solution is to map them all to the same ECU.

In figure 4.11, a traditional solution is presented in the first quadrant (north-west): peripheral components are mapped to peripheral ECUs, and central components are mapped to central ECUs. Below each ECU is given its installation-rate, based on the values from figure 4.10. Because we assume that the ECUs are equally equipped and priced, we can sum up the installation-rates of the three ECUs as comparison value.

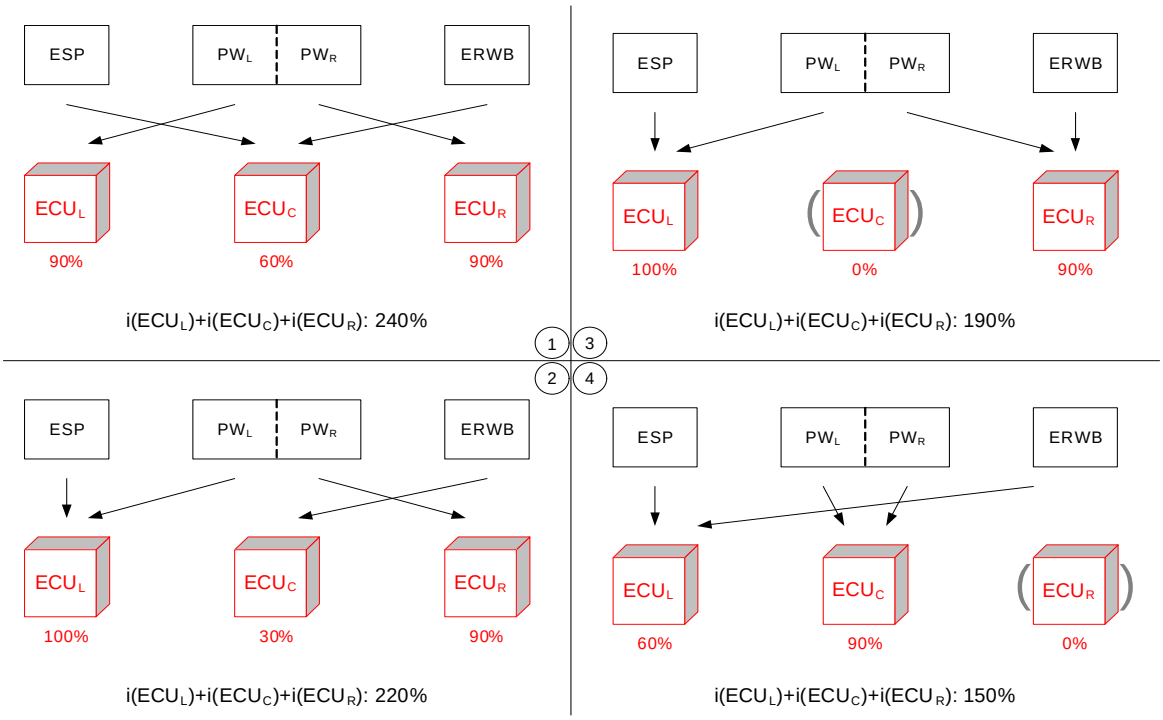


Figure 4.11.: Example for the influence of different components distributions on the ECU installation-rate

The next three quadrants of the example alter the distribution of the components. The installation-rates change according to the new sets of features that are covered by the individual ECUs. The best solution in this simplified example is the one in the last quadrant (south-east) with an accumulated installation-rate of 150%, because

mapping components that are ordered in similar combinations onto the same ECU optimizes costs.

Aside from the sole installation-rates, this last solution requires more cables to wire up the components to their assigned ECU, which again increases the costs to some extent. Existing middleware layers, and particularly the upcoming AUTOSAR specification, allow the distribution of the individual SW/HW-components of a single assembly to different ECUs. This results in a complexity that human experts can hardly handle without software support.

Furthermore, in section 4.4 it has been outlined that the resources being consumed by components have a consumption_type. For example, the motor in the doors for power windows has to be modelled as a HW-component that additionally consumes pins and power electronics. The motor as resource is _consumed_ by this HW-component with "feature order-rate" consumption type, because the motor is only installed if the power windows feature is actually requested. But the pins are located on the ECU and are therefore _consumed_ with "ECU installation-rate" consumption type. For the power electronics both types are possible.

Note that the distribution optimization changes only the ECU installation-rate; the portion of costs for resources that are consumed with feature order-rate is constant for all possible distributions. In order to calculate the general costs for a given solution mapping, it is possible to use plain database views for aggregation and calculation; but for the framework it would be too much overhead to use these views for calculation, so the objective estimation for "costs" has to adopt an internal calculation incrementally on mapping changes.

## 4.6. Suppliers for Components

Distributing SW- and HW-components freely to a set of ECUs can lead to the situation that an ECU keeps, for example, five components that have traditionally been developed by five different supplier companies. Figure 4.12 introduces the relationship between components and suppliers, allowing to evaluate and constrain the number of suppliers that are needed for a single ECU.

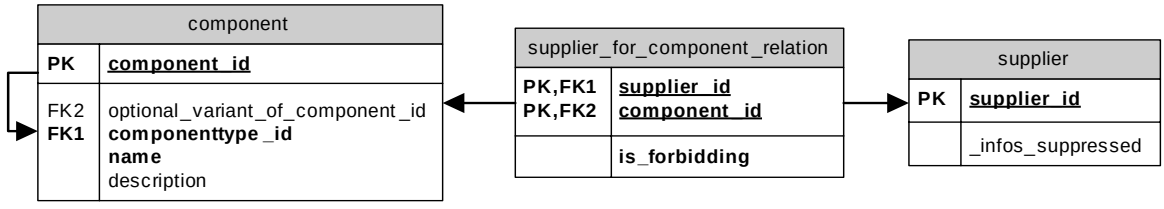


Figure 4.12.: Resource to resource

The supplier_for_component_relation provides the information which supplier is able to provide a component. The is_forbidding attribute defaults to false. Setting it to true enforces that the supplier must not participate in any ECU where such a component has been mapped to. This attribute is a preliminary substitution for a more complex representation of whole business models that could be necessary in the future.

The evaluation of the supplier_for_component_relation in order to find the minimum set of suppliers for a set of components is available as part of the MOOVE implementations, but will not be part of this thesis.

## 4.7. Physical and Functional Networks

Networks are considered in regard to bandwidth consumption. Every network_node can be connected to multiple physical networks, shown in figure 4.13. Every network is specified by its network_type, like CAN or LIN. The network type specifies its maximum busload as baud_rate.

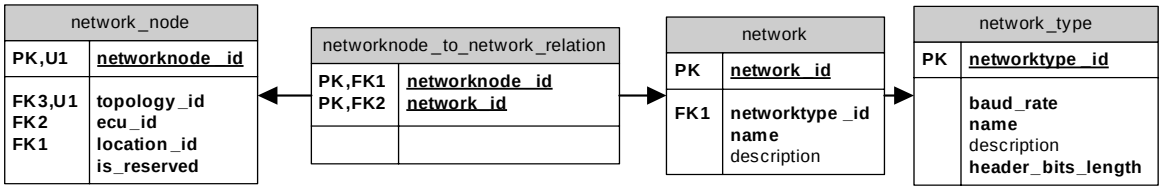


Figure 4.13.: Physical networks

The relevant data model of the functional network is shown in figure 4.14. The model supports a single-producer/multiple-consumer model. This is common for networks of the types CAN or LIN. The model is yet preliminary and has to be understood as a proof of concept for the objective busload. At the moment only a coarse estimation of the busload is required. As future work, it is considered to generalize and integrate the specific concept of signals into a generic concept and model of "resources that are shared among several ECUs at run-time". The resource table (from figure 4.3 in section 4.4) is able to express ECU-local resources, but future vehicle architectures will probably reconfigure and redistribute SW-components at run-time instead of compile-time, so that models for dynamic resource sharing will be required.

Since every component can be instantiated several times in form of component_instances as described above, an interface for the communication is modelled. Every SW-component can have interfaces. There is a distinction between the exported producer_interfaces and the imported consumer_interfaces. The set of imported and exported interfaces for a component defines its black-box behaviour and its compatibility to other SW-components. Every interface has an interface_type, specifying its bit vector length.

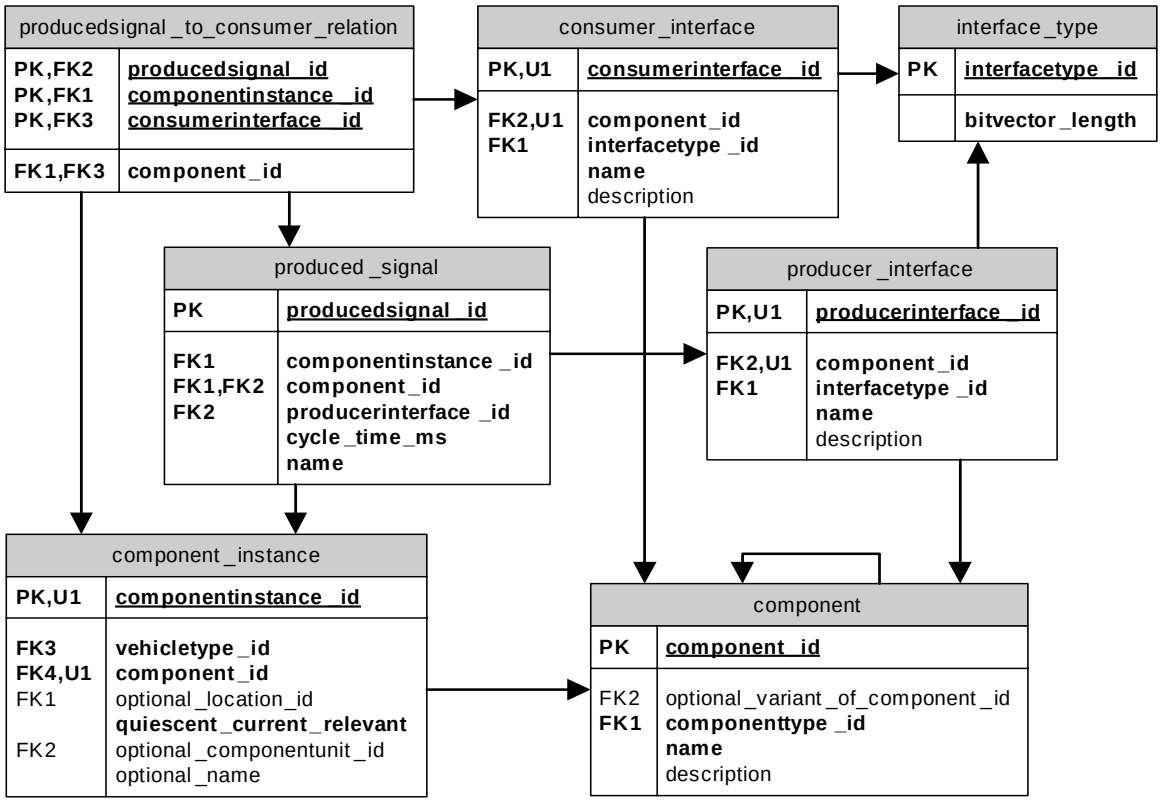


Figure 4.14.: Functional networks

The frame size on the network is protocol dependent, and therefore requires an evaluation based on component instances, network nodes and a concrete mapping between both: A component has no knowledge about its functional context. In contrast, the component_instance needs this knowledge. A functional network can be modelled by connecting the interfaces – or more specific by connecting every producer_interface with multiple consumer_interfaces using signals.

As shown in figure 4.14, a produced_signal instantiates a producer_interface to a signal by specifying both the component and the according component_instance. The produced_signal_to_consumer_relation connects the produced_signals to the consumer interfaces of the receivers, finishing the signal connection.

The MOOVE implementation for the busload estimation, which has been provided by another team member, actually handles SW-components that are linked multiple times over different network paths, applying a routing algorithm based on the *open shortest path first* (OSPF) [Moy98] protocol, a well-known routing protocol and algorithm for the internet. The frame size can be estimated with the bit vector length of the signal plus the header_bits_length of the relevant network_type.

## 4.8. Additional Model Extensions

The available data model contains some additional relations that are currently not used by the MOOVE project and have not been worked out in detail. They are provided in the appendix section B.1.

These relations include a model for resource-to-resource dependencies, particularly in order to integrate OS-resources: The fine evaluation of resource consumptions for the ECU's OS and middleware layers itself requires a differentiated modelling of OS-resources like different available task scheduling algorithms, as well as their influence on memory consumption or CPU. In fact, an enhanced feature trees would be required, as have been introduced in section 4.5 for the features of a vehicle that can be ordered by a customer.

Other relations that are available in the appendix relate to the framework configuration. The information about the configuration should be stored for a mapping solution, containing algorithms and meta information like the date and the user name of the person in charge.

## 4.9. Summary

Section 4 introduced a data model for control networks in vehicles. In section 4.1 the model creation and the applied tools were described. After this introduction, section 4.2 specified the relations for the distribution mapping – including ECUs, components, network nodes, component instances and the mapping. The next section 4.3 specified the relations for objectives, and quality ratings in relation to a mapping.

The resource consumption of components as well as the resource offering by ECUs were explained in section 4.4. The hierarchy from vehicle series to features, down to the set of component instances that has to be distributed to the ECUs, were topic of section 4.5. Furthermore, the model for customer orders was provided as well as the partitioning into feature combinations. The influence of the order-rates for feature combinations on ECU installation-rates was explained by an example.

Section 4.6 introduced the relationship between components and possible suppliers, in order to constrain the number of suppliers that have to cooperate for a single ECU. The model for physical and functional networks was given in section 4.7, based on single-producer/multiple-consumer signals. Finally, section 4.8 referred to some preliminary extensions of the data model, that are given in the appendix.

# 5. The MOOVE Project

Based on the HeurOpt framework, the MOOVE project consists of concrete implementations for the vehicle domain, and aims to solve the problem of distributing software and hardware components to ECUs. A database aware configurator is available as well as a simple optimization kernel based on evolutionary algorithms as proof of concept. The database model from section 4 is also part of the MOOVE project as well as objective estimation and constraint implementations.

## 5.1. Optimizer and Configurator

The generic HeurOpt implementation of the optimizer component is the GenericOptimizer[1]. It uses the termination condition and the population independently from domain specific adaptations and can be used without any extension.

A special configurator and directive is provided, which is aware of the database. It is based on two classes of the HeurOpt framework shown in figure A.13 of the appendix: The AbstractOptimizerConfigurator implements the handling of all configurable class names by Java reflection; the MappingOptimizerConfigurator extends the configuration with sets for targets and targeting items independently from the vehicle domain. Now, the MOOVE project provides the VehicleMappingOptimizerConfigurator as part of the de.fau.cs.i2.MOOVE.heuropt.configuration package, in figure B.4 of the appendix. It uses the database as input for both sets; for this purpose the directive is extended to include the vehicle series and topology as selection criteria. The Torque-generated class for accessing the tuples of the component_instance relation is extended so that the TargetingItem interface is supported. The class for the network_node relation is equally extended to support the Target interface.

As termination condition only the simple CountedConditionalIterator is used, which is based on a configurable number of iterations. It has already been mentioned in section 3.2 as part of the generic HeurOpt implementations.

[1] The GenericOptimizer is part of the de.fau.cs.i2.heuropt.domain.generic.configuration package, in figure A.11 of the appendix.

## 5.2. Population and Solution

A preliminary implementation of an evolutionary algorithm is available. The de.fau.cs.i2.MOOVE.heuropt.population package includes the VehicleMappingPopulation class as population and the VehicleGlobalMappingSolution class as its default solution, shown in figure B.5 of the appendix.

The solution class uses the generic I2TAttributedFullMap as model data and implements a mutate(..) and a crossoverUniform(..) operation. A randomFill() method is implemented based on the mutate(..) operation which can be used to create solutions for the initial population.

The population class implements the generateInitialSolution(..) interface. The manually provided fixed information from the componentinstance_fixedto_networknode_relation is accessed and applied to the initial solution, preparing the I2TAttributedFullMap like it was shown in section 3.8. Furthermore, the population class implements the generateInitialPopulation(..) interface using the randomFill() method to create mapping solutions for the initial population; obviously, these randomly filled mappings will usually not be valid solutions. The VehicleMappingPopulation uses the SPEA2 Pareto front implementation to store its solutions, as mentioned in section 3.7. In order to evolve the population, the VehicleMappingPopulation simply applies some mutate and cross-over operations during its regeneratePopulation().

According to the discussion of local models in section 3.9, the VehicleLocalMappingSolution is available as ECU-local solution and uses the generic TargetingitemsPerTargetSet as model data. Like the population and global solution it is also part of the de.fau.cs.i2.MOOVE.heuropt.population package, in figure B.5 of the appendix. At the moment it does not support ECU-local optimizations like variant optimization, but simply updates its model data. It also delegates update notifications to the local OEs using the notifyObervers() method which is inherited from the HeurOpt implementations.

Proper optimization kernels and algorithms will be implemented in the future, but are not the goal of this thesis. Implementations for an ant colony optimization strategy are in work, as well as for particle swarm optimization. One goal will be to compare different optimization strategies according to applicability, performance and the structure of their results.

## 5.3. Objective Estimations

Several objective estimations are available as part of the MOOVE project. Based on the data model in section 4, other team members have implemented the OEs, but they are not part of this thesis.

For the costs objective a database view was created as a proof of concept. It evaluates the resource consumptions and calculates the feature order-rates and ECU

installation-rates. It is not feasible to instrument the database view during optimization because of performance reasons. Therefore, a Java implementation of the view's functionality will soon be available, using incremental updates upon mapping changes. The evaluation of the quiescent current will be handled by the same implemenation, just using a different resource_type as selection criteria. Weight evaluation is also based on this facility, but additionally requires cable length calculation. Implementations for cable length and cable duct filling degree are in preparation.

The hardware availability constraints are handled generically for all types of resources: The resource consumptions of the components that are currently mapped to an ECU are accumulated. Then they are set into relation with the max_units attribute of the provider_provides_resource_relation (section 4.4). If the value is exceeded, a constraint violation will occur. The implementation of this (global) OE is a composition of local OEs for each ECU, which are registered to the local models.

An implementation for the busload objective allows to evaluate either the worst-case or average busload. It takes SW-components into account that are linked multiple times over different network paths, applying a routing algorithm based on OSPF, as was mentioned in section 4.7.

The evaluation of the supplier_for_component_relation in order to find the minimum set of suppliers for a set of components is also available, but is not part of this thesis.

## 5.4. Exemplary MOOVE Templates

In the domain of control networks for vehicles, special template classes for population and solution are available in the de.fau.cs.i2.MOOVE.heuropt.template package. Both templates are shown in the appendix as listings B.1 and B.2. They are provdided in addition to the HeurOpt template listings A.1 and A.2, because several domain specific implementations can be inherited from available MOOVE classes, as it is done by the templates.

For instance, the MOOVE template for the population already inherits an implementation for the initDomainSpecific(..) method, so that the sets of targets and targeting items are available as well as the implementations for creating the initial solution and initial population. The initSubDomainSpecific(..) is provided as a substitution, allowing to further integrate configuration data into the initialization process that is propagated to this method in form of its HashMap parameter.

In contrast to listing A.2, the MOOVE template for the solution does not have to implement any framework methods. They can be inherited as long as the I2T-AttributedFullMap is used as model data. The access to the map and to the sets of targets and targeting items is demonstrated in the listing. For the developer it becomes possible to focus exclusively on the operations for the optimization strategy.

## 5.5. Summary

Section 5 gave an overview of the MOOVE project and its available implementations. Section 5.1 characterized the applied optimizer and termination condition as well as the database aware configurator.

The preliminary optimization kernel based on evolutionary algorithms was described in section 5.2. It allows the application of evolutionary algorithm operations as well as the generation of randomly filled initial solutions. Section 5.3 outlined the available implementations for objective estimations. Finally, section 5.4 referenced two domain specific template listings in the appendix that utilise the MOOVE implementation.

# 6. Conclusion

## 6.1. Summary

This thesis introduced the architecture of a framework for multi-objective optimization. The concepts for population-based optimization algorithms were described, in particular the problems of multiple objectives that require the concept of a Pareto front for solutions, as well as constraints and violation handling. The strategies of evolutionary algorithms and swarm intelligence were outlined, because they are considered as potential kernels of the framework.

The architecture of the HeurOpt framework included a component model for optimizers as well as component repositories. The optimizer components are configured in form of declarative descriptors. The interfaces of termination conditions, populations and solutions were defined, allowing scalable integration of multiple objectives and constraints as well as the interchangeable application of different optimization algorithms. Furthermore, the integration of local optimization models was achieved. Implementations are provided for all interfaces, comprising cloning support for the complex structures.

An introduction to control networks for vehicles was given, focusing on the problem of distributing software and hardware components to electronic control units. A data model was presented for several aspects of this problem domain. The model for the mapping solutions itself was provided as well as their evaluated quality ratings. Several objectives were analyzed, including use cases and examples. Particularly the consumption of resources had been important, because of its influence on costs, cables and quiescent current as well as arbitrary hardware availability constraints. Following a discussion of costumer orders and feature order-rates, the influence of order-rates for feature combinations on ECU installation-rates was exemplified. Furthermore, a model to constrain the number of suppliers that have to cooperate for a single ECU was introduced as well as a model for physical and functional networks.

The data model served as preparation for the application of the HeurOpt framework to solve the distribution of SW-components to ECUs with heuristic optimization. The MOOVE project extended the HeurOpt framework, including database access to the data model as well as domain specific implementations of the framework interfaces. As proof of concept, a simple optimization kernel based on evolutionary algorithms was implemented.

## 6.2. Further Work

In relation to the HeurOpt framework, support is needed for tabu search and ancestry data structures in form of standardized interfaces and implementations. Integrated support for parallelisation has to be provided for arbitrary population-based strategies, and the penalty types for handling constraint violations should be implemented. XML support for deployment descriptors is desired as well as the XML import and export of Pareto fronts. Furthermore, different optimization algorithms need to be implemented in order to compare them in regard to their performance and their quality of results.

In relation to the MOOVE project, the next steps will include the optimization of ECU variants in the framework. A graphical user interface has to be provided for editing the data, integrating an expert system for the implicit semantic relationships between some attributes. Also the process of choosing one solution from the Pareto front should be supported by a graphical interface. The data model can be extended by feature trees for resources of the operating system. The integration of safety levels, probably in form of resources, and the evaluation of related constraints need further research.

# A. HeurOpt

## A.1. UML Class Diagrams

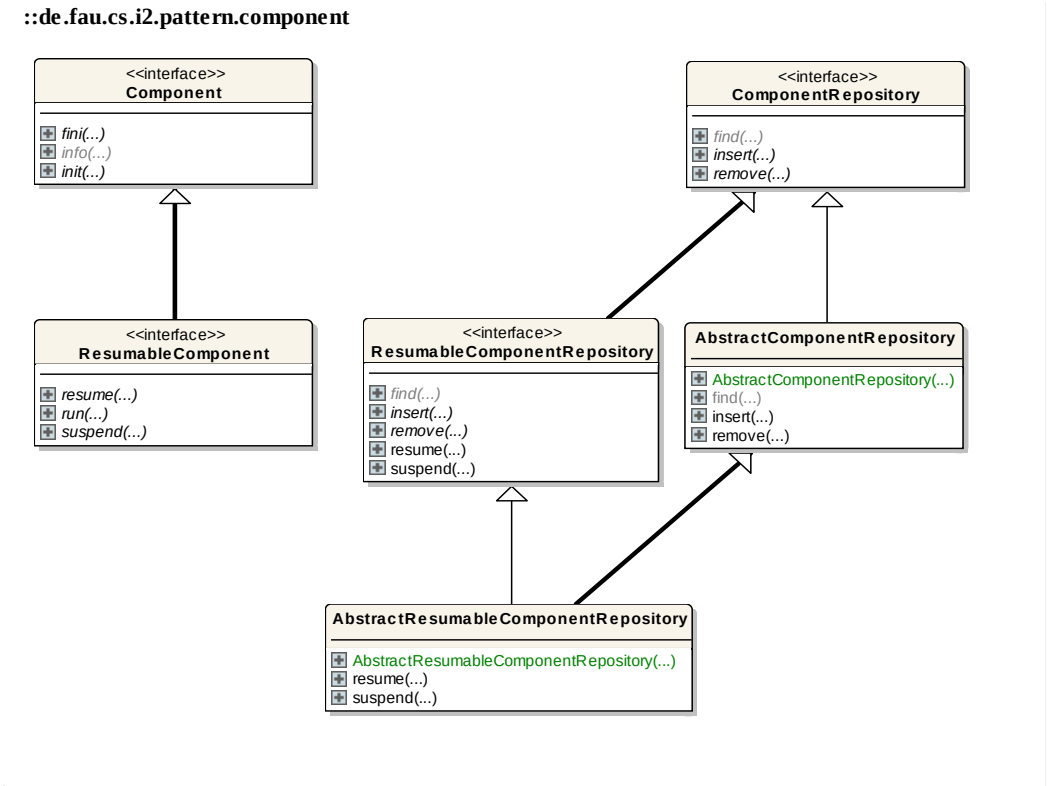


Figure A.1.: Package: de.fau.cs.i2.pattern.component

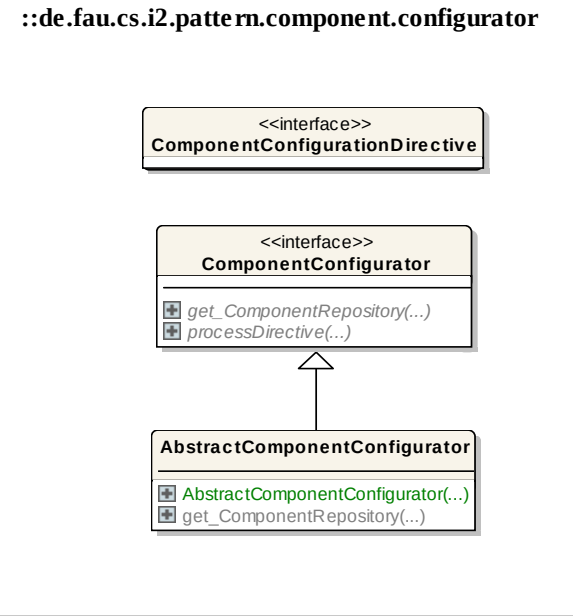


Figure A.2.: Package: de.fau.cs.i2.pattern.component.configurator

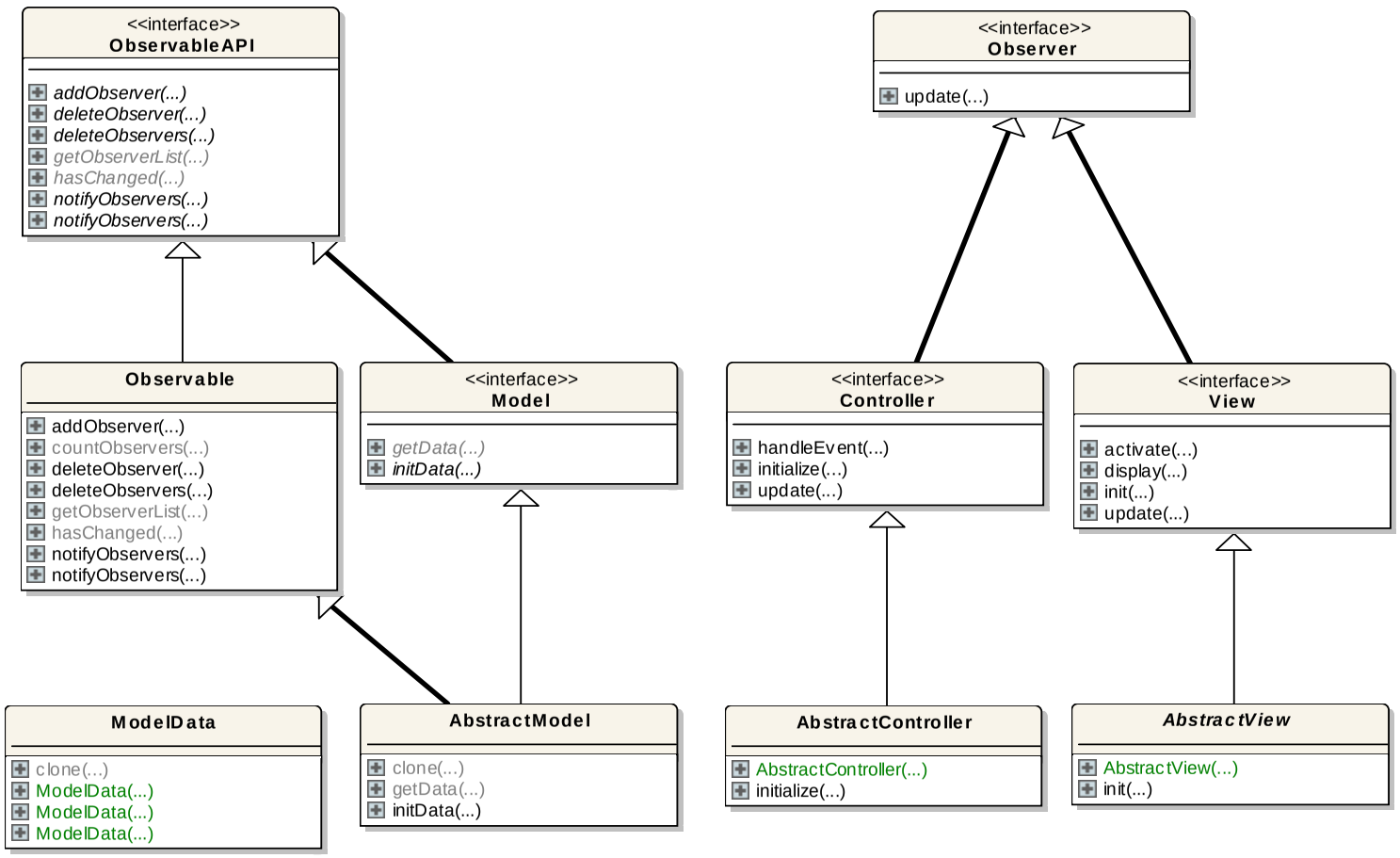


Figure A.3.: Package: de.fau.cs.i2.pattern.MVC

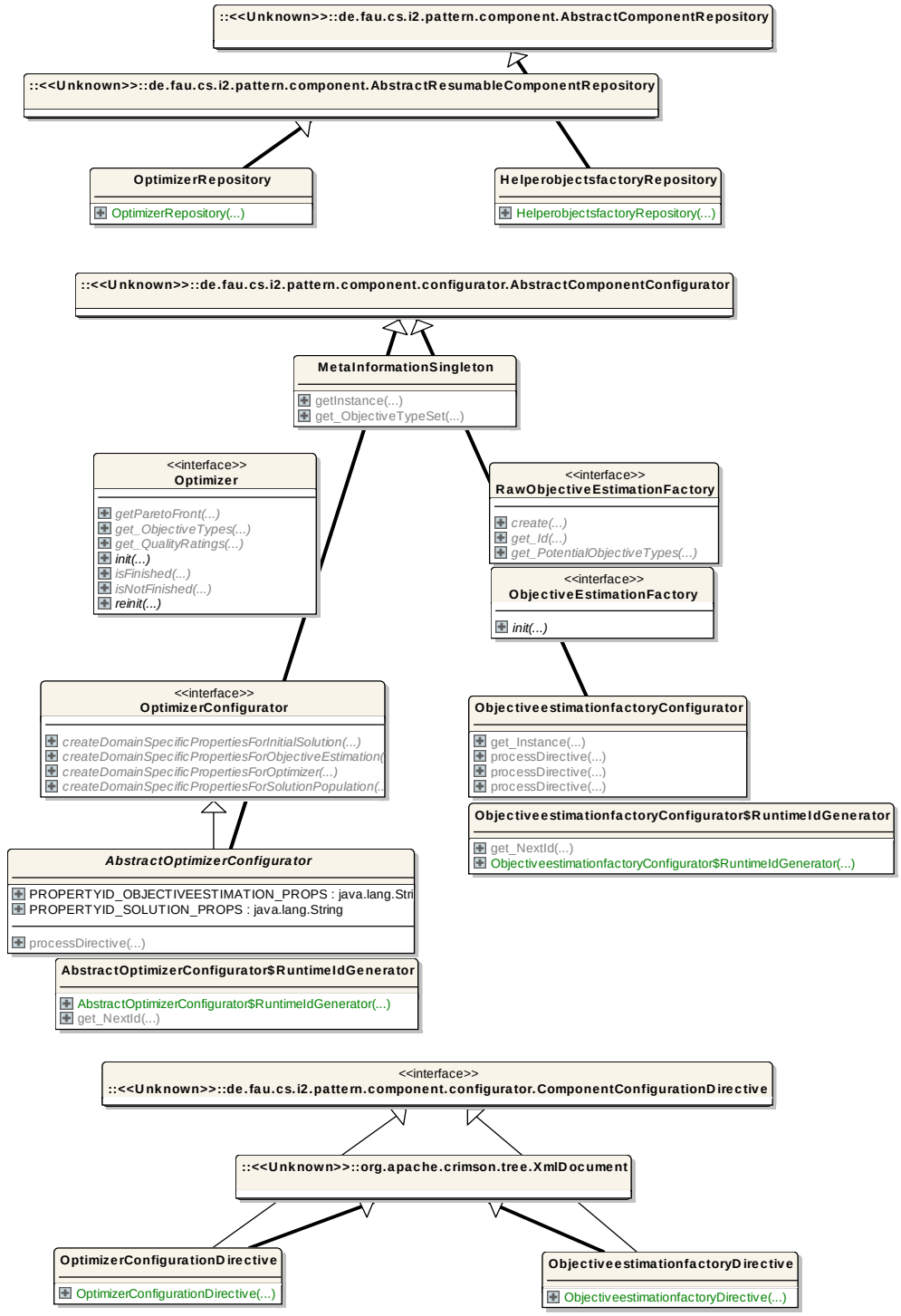


Figure A.4.: Package: de.fau.cs.i2.heuropt.kernel.configuration

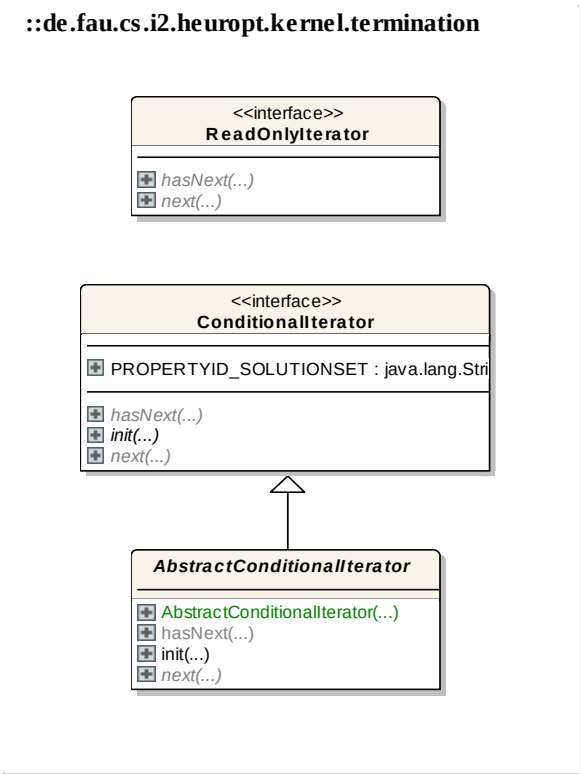


Figure A.5.: Package: de.fau.cs.i2.heuropt.kernel.termination

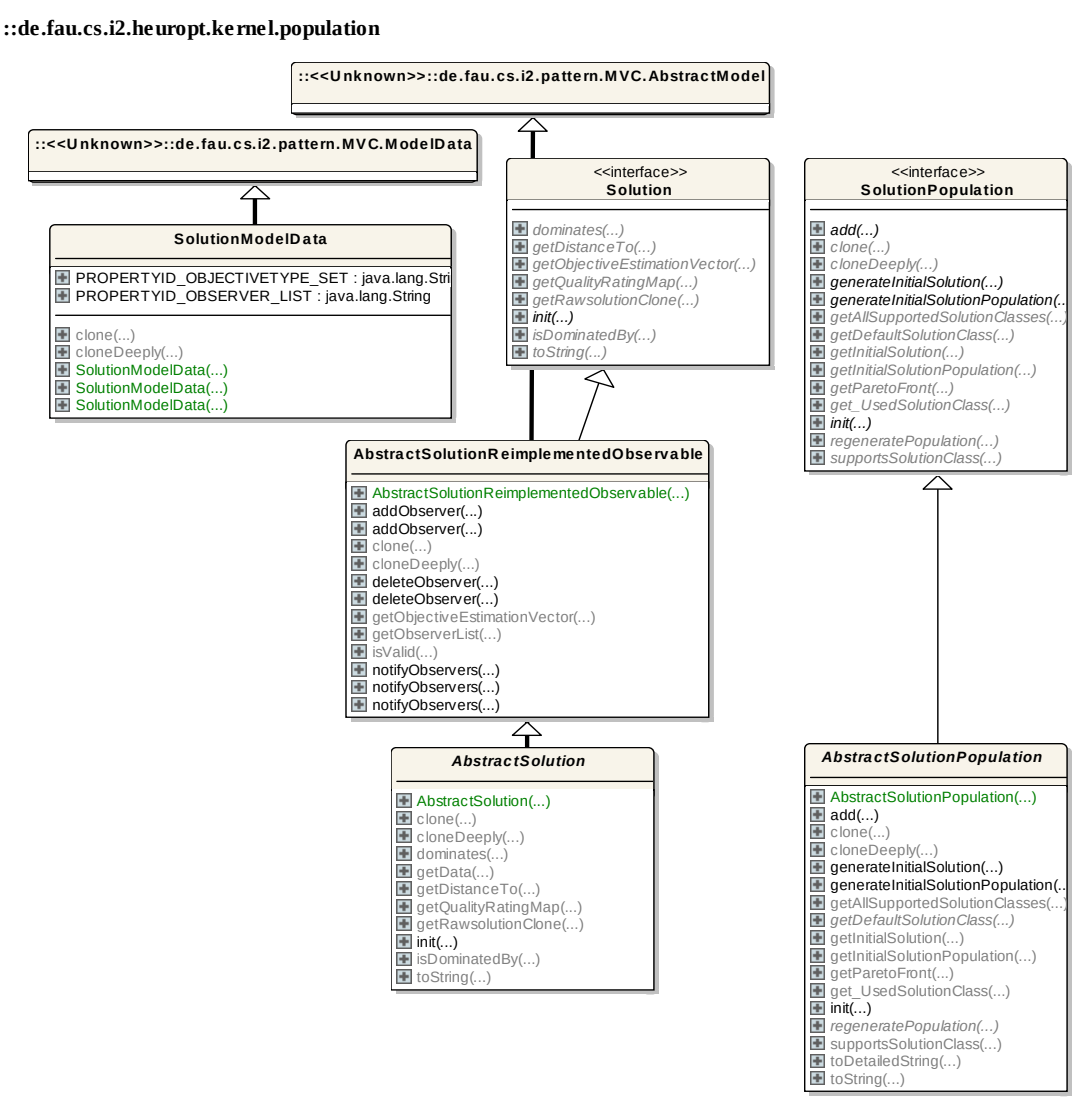


Figure A.6.: Package: de.fau.cs.i2.heuropt.kernel.population

**::de.fau.cs.i2.heuropt.kernel.objective**

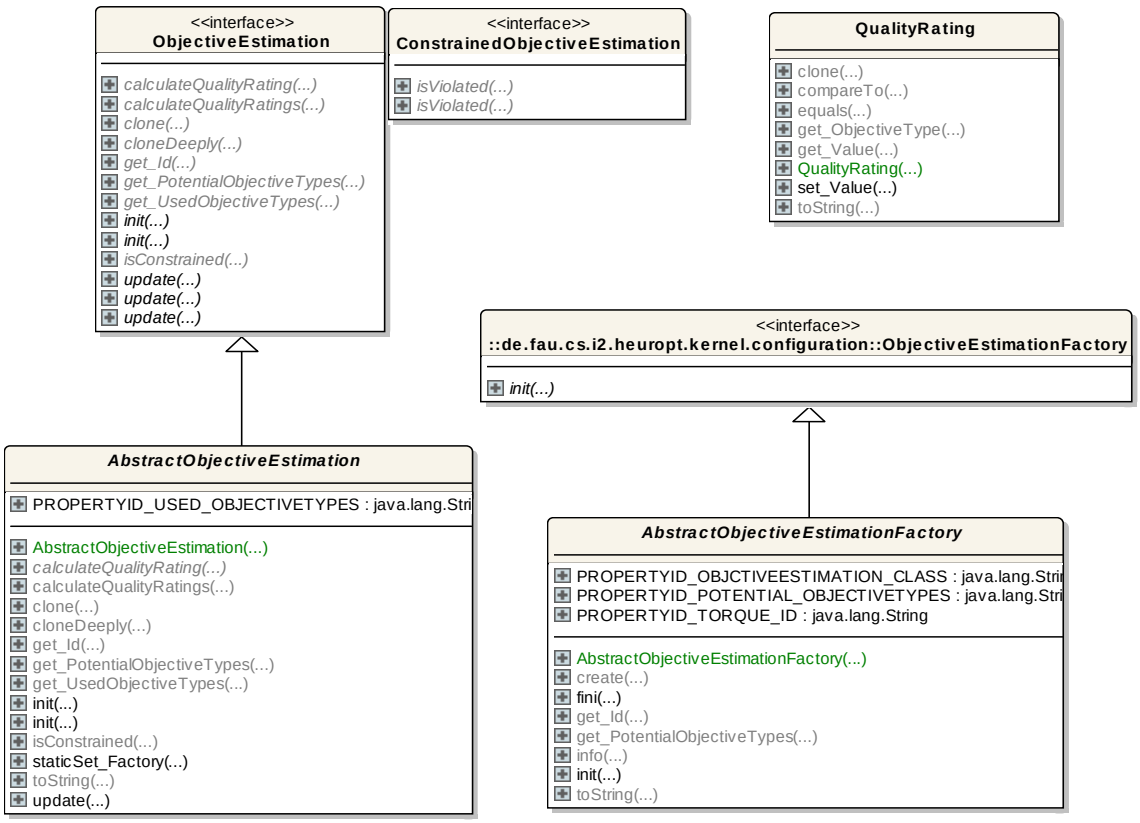


Figure A.7.: Package: de.fau.cs.i2.heuropt.kernel.objective

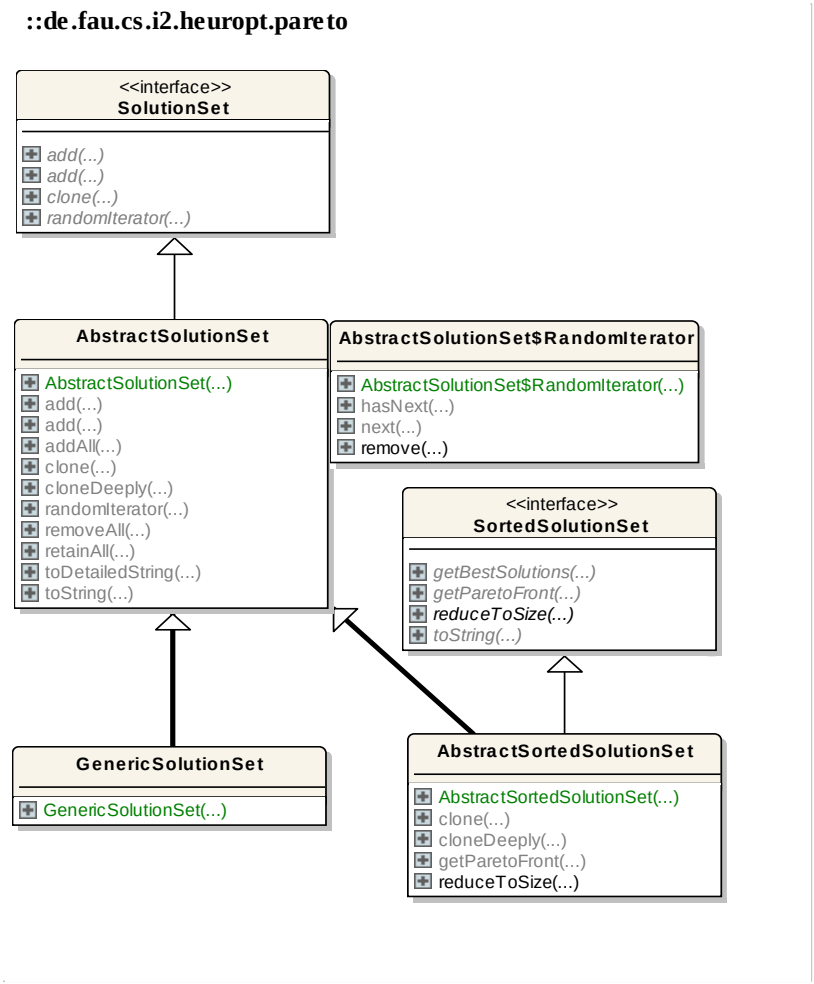


Figure A.8.: Package: de.fau.cs.i2.heuropt.pareto

Figure A.9.: Package: de.fau.cs.i2.heuropt.pareto.leveled

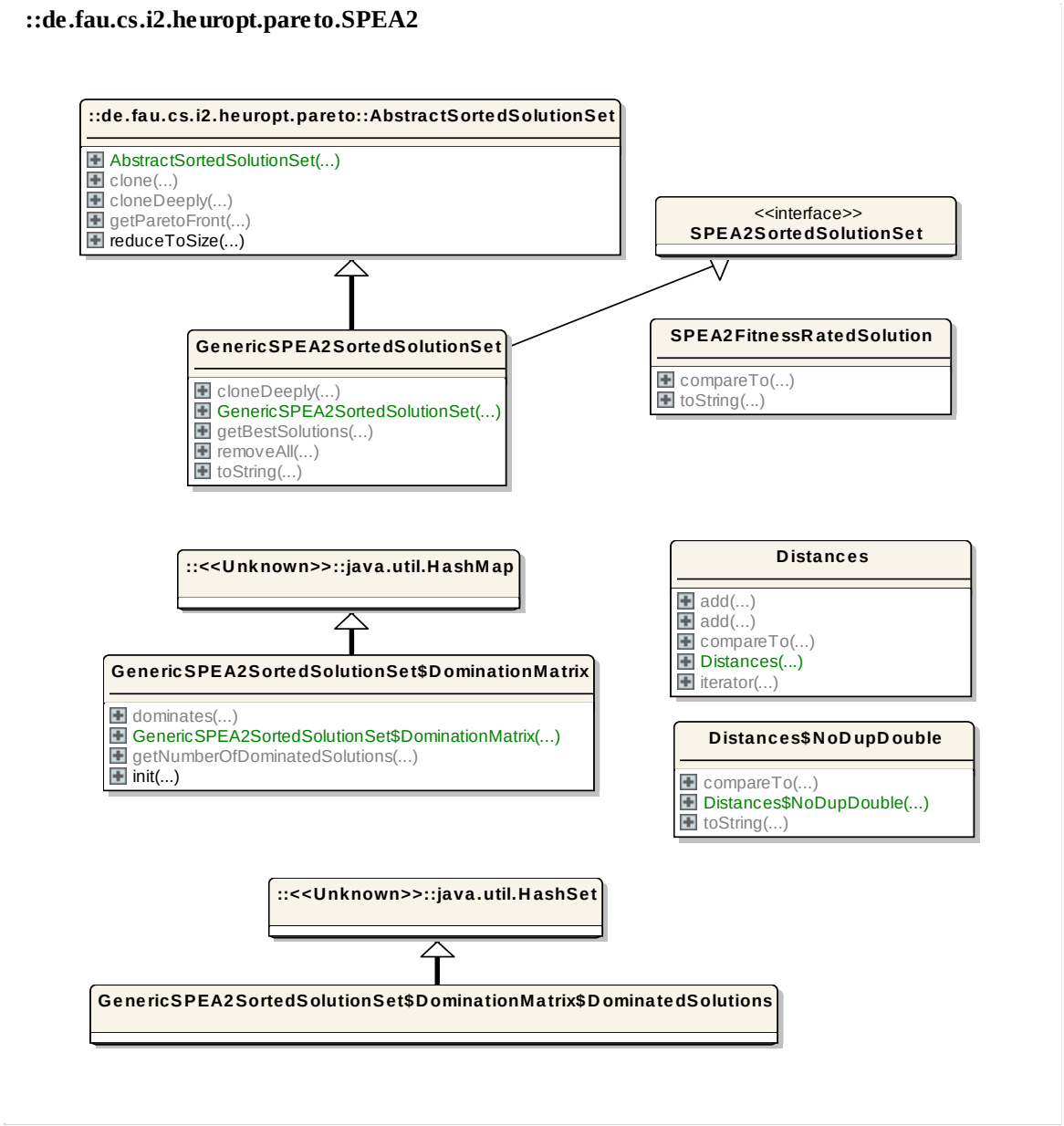


Figure A.10.: Package: de.fau.cs.i2.heuropt.pareto.SPEA2

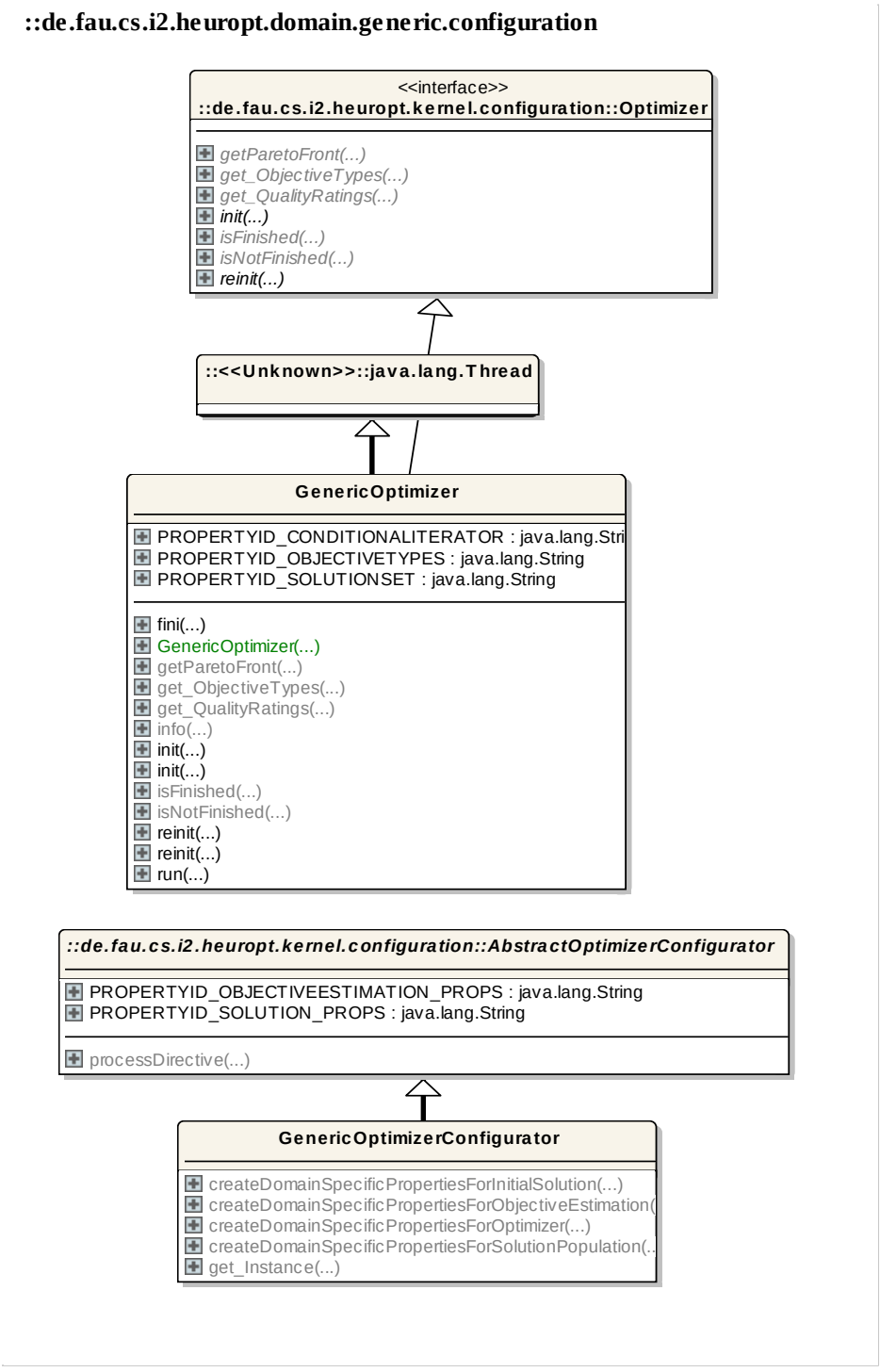


Figure A.11.: Package: de.fau.cs.i2.heuropt.domain.generic.configuration

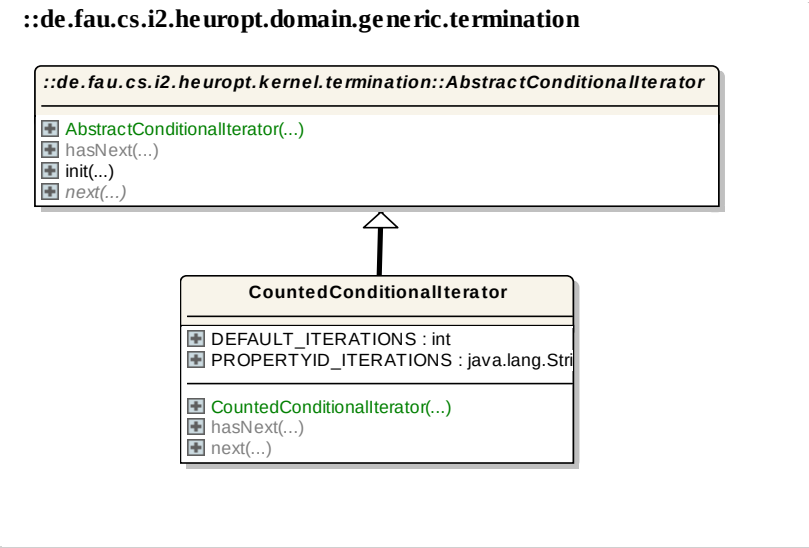


Figure A.12.: Package: de.fau.cs.i2.heuropt.domain.generic.termination

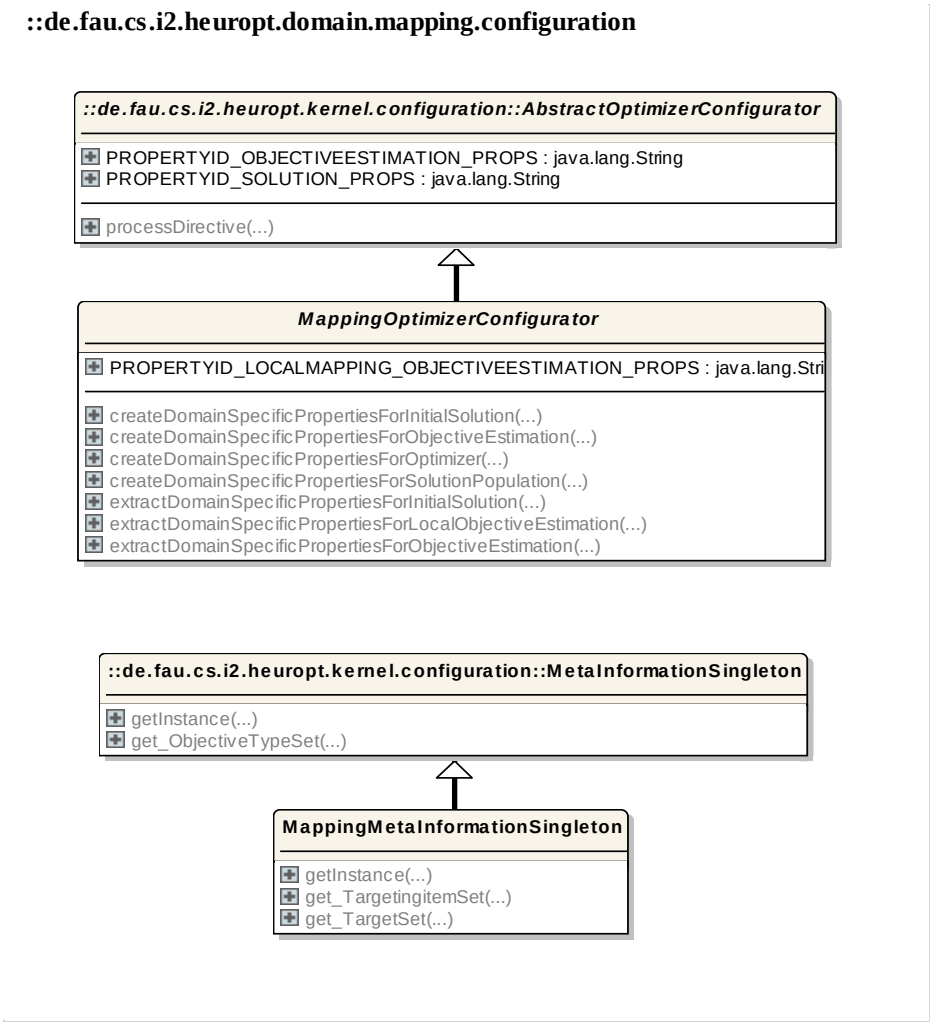


Figure A.13.: Package: de.fau.cs.i2.heuropt.domain.mapping.configuration

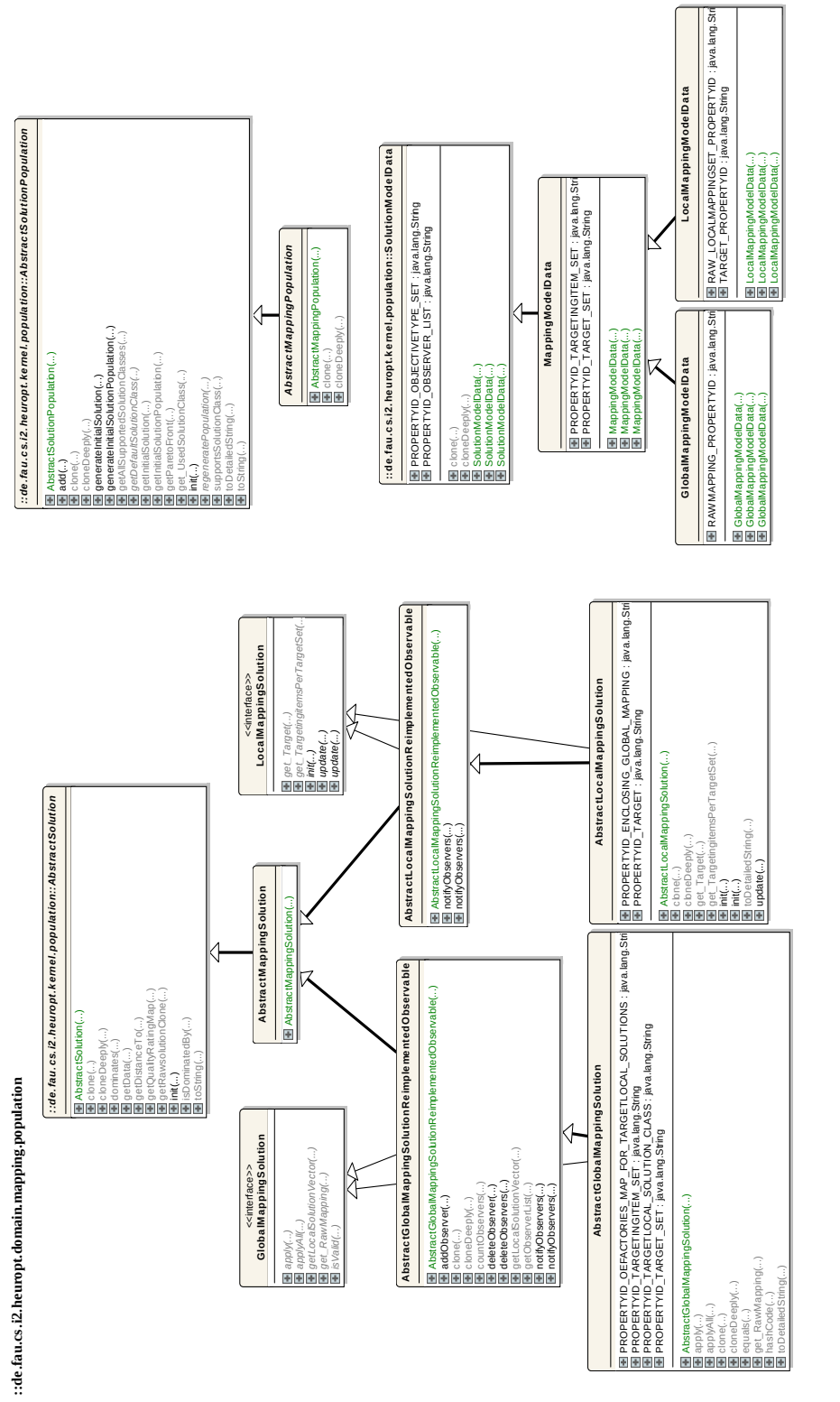


Figure A.14.: Package: de.fau.cs.i2.heuropt.domain.mapping.population

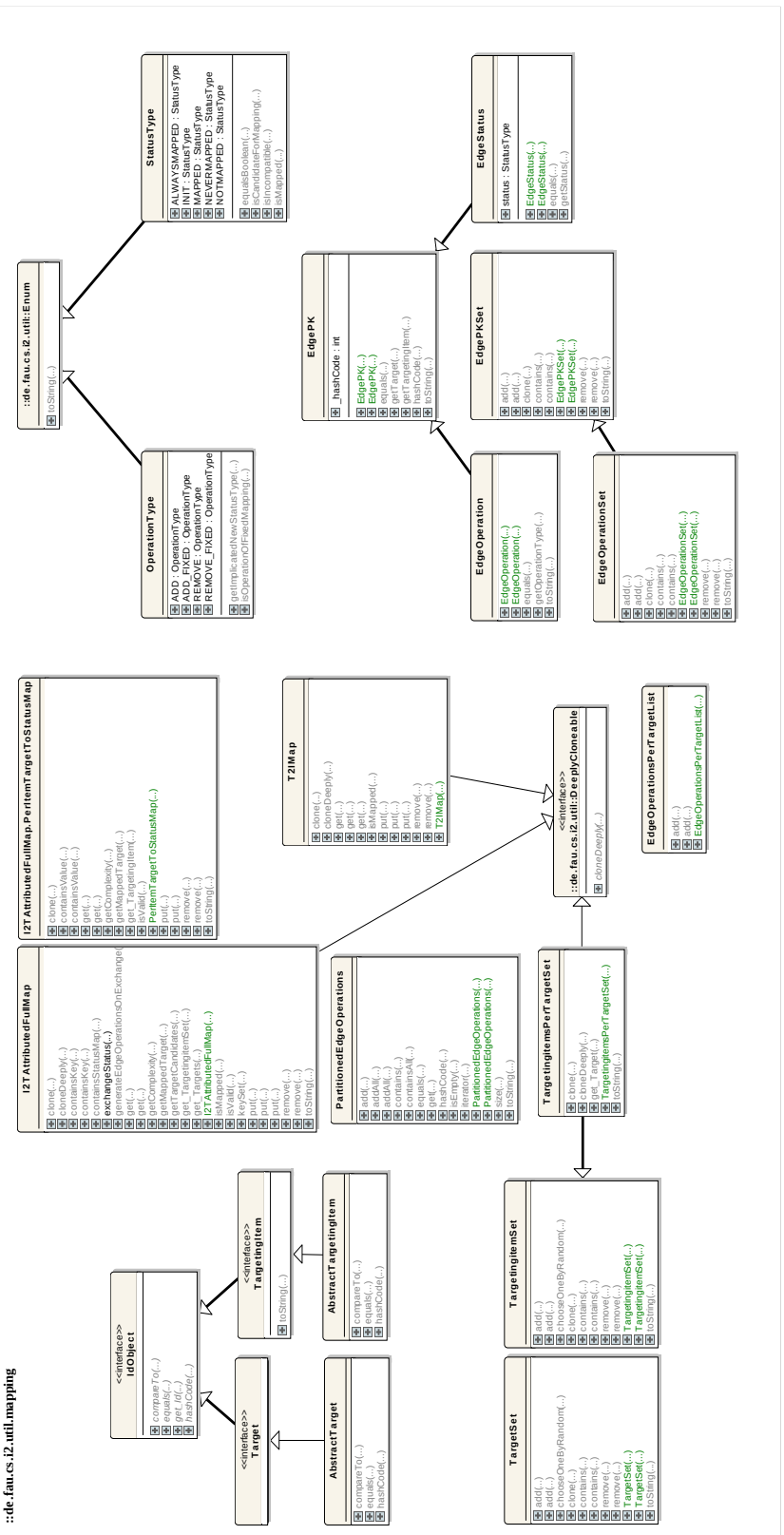


Figure A.15.: Package: de.fau.cs.i2.util.mapping

**::de.fau.cs.i2.heuropt.domain.mapping.objective**

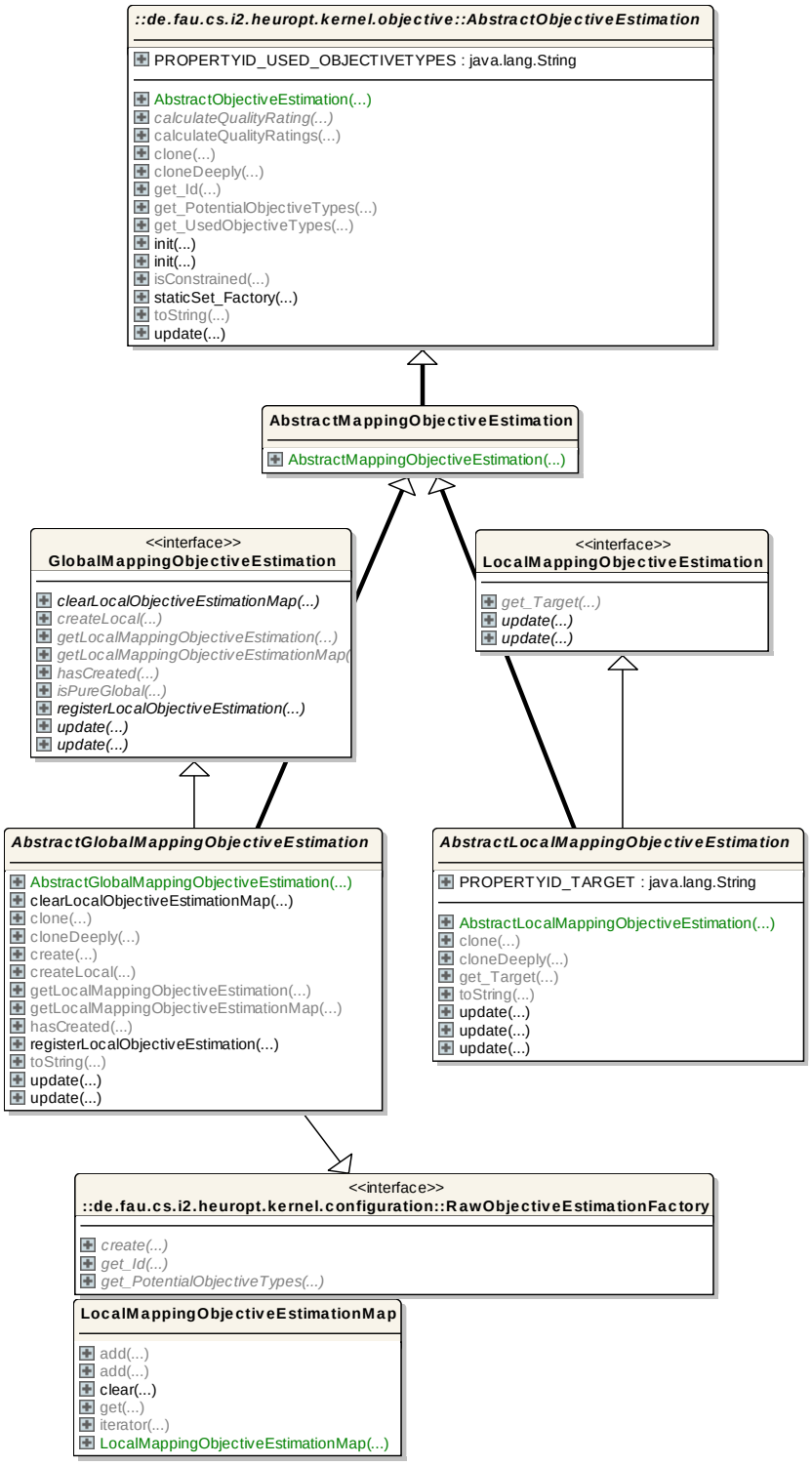


Figure A.16.: Package: de.fau.cs.i2.heuropt.domain.mapping.objective

## A.2. Template Listings

```
package de.fau.cs.i2.heuropt.template;

import java.util.HashMap;
import java.util.Iterator;

import de.fau.cs.i2.heuropt.kernel.population.AbstractSolutionPopulation;
import de.fau.cs.i2.heuropt.kernel.population.Solution;
import de.fau.cs.i2.heuropt.kernel.population.SolutionPopulation;
import de.fau.cs.i2.heuropt.pareto.SolutionSet;

public class TemplatePopulation extends AbstractSolutionPopulation {

    public final static Class DEFAULT_SOLUTION_CLASS = TemplateSolution.class;

    public Object clone() {
        return super.clone();
    }

    public Object cloneDeeply() {
        return super.cloneDeeply();
    }

    public Class getDefaultSolutionClass() {
        return DEFAULT_SOLUTION_CLASS;
    }

    protected void initDomainSpecific(HashMap props) {
        return;
    }

    protected Solution makeRawsolutionToTemplate(Solution rawSolutionClone) {
        Solution templateSolution = rawSolutionClone;
        // ASSERT: rawSolutionClone has has been structurally set up
        return templateSolution;
    }

    protected Solution makeTemplateToValidInitialSolution(
            Solution templateSolutionClone) {
        Solution validSolution = templateSolutionClone;
        // ASSERT: makeRawsolutionToTemplate has been invoked before
        return validSolution;
    }

    public SolutionPopulation regeneratePopulation() {
        SolutionSet sc = (SolutionSet) this.lnkSortedSolutionSet.cloneDeeply();

        // ... create next population ...
        for (Iterator i = sc.iterator(); i.hasNext();) {
            TemplateSolution solution = (TemplateSolution) i.next();
            solution.operation();
        }

        return this;
    }
}
```

Listing A.1: Template code listing for a population class

```
package de.fau.cs.i2.heuropt.template;

import java.util.HashMap;

import de.fau.cs.i2.heuropt.kernel.population.AbstractSolution;
import de.fau.cs.i2.util.ChangeObject;
import de.fau.cs.i2.util.mapping.I2TAttributedFullMap;
import de.fau.cs.i2.util.mapping.TargetSet;
import de.fau.cs.i2.util.mapping.TargetingitemSet;

public class TemplateSolution extends AbstractSolution {

    protected void applyOnSolutionDataStructure(ChangeObject change) throws Exception
         {
        /* apply the change on the solution's data structure
         * (is invoked by the inherited apply[All](..) methods) */
        // ...
    }

    public Object clone() {
        return super.clone();
    }

    public Object cloneDeeply() {
        return super.cloneDeeply();
    }

    protected void initDomainSpecific(HashMap props) throws Exception {
        // initialize the solution data structures
        // ...
        return;
    }

    public void operation() {

        /* perform an evolving operation on the solution */
        // ...

        /* at the end: update the solution data structure with ChangeObjects:
         * (the "ToDo[Set]" class has to be defined, according to the used
         *  data model structure) */
//        ChangeObject change = new ToDo(..);
//        this.apply(change);
        /* or use a set */
//        ChangeObjectSet changeSet = new ToDoSet(..);
//        changeSet.add(..);
//        this.applyAll(changeSet);
    }
}
```

Listing A.2: Template code listing for a solution class

# B. MOOVE

## B.1. Data Model (Preliminary Extensions)

The available data model contains some additional relations, that are currently not used by the MOOVE project: feature trees, resource-to-resource relationships and storage of the framework configuration.

### Feature Trees

Feature trees have been outlined in section 4.5, in the context of vehicle features that can be ordered by customers. Please refer to *feature-oriented domain analysis* (FODA) [KCH+90], and the book form Czarnecki and Eisenecker [CE00] for more information.

Figure B.1 describes a data model for feature trees. The feature_hierachy relation mirrors the features from the feature relation. The father-child relation is established by the father_ID attribute. The attributes is_optional and is_abstract are directly related to the concept of feature trees, as are the feature groups: A feature is part of a feature group in relation to its father. The number attribute is a father-local index, allowing to order his subordinate groups. The logic_operation is either "AND", "OR" or "XOR".

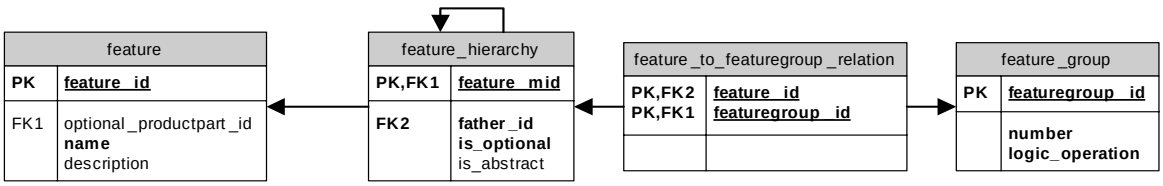


Figure B.1.: Model for feature trees

### Resource-to-resource Relationships

Figure B.2 provides the relations for resource-to-resource dependencies. The resource_to_resource_relation includes the ECU_ID, because they may be dependend on the ECU. Usually, the relations will be provided with the environment-ECU (ID = 0).

The semantics of a resource-to-resource relationship is provided by its r2r_relation_type. Two mandatory attributes are is_directed and is_weighted. If the resource-to-resource tuple is not directed, then the evaluation has additionally to consider the

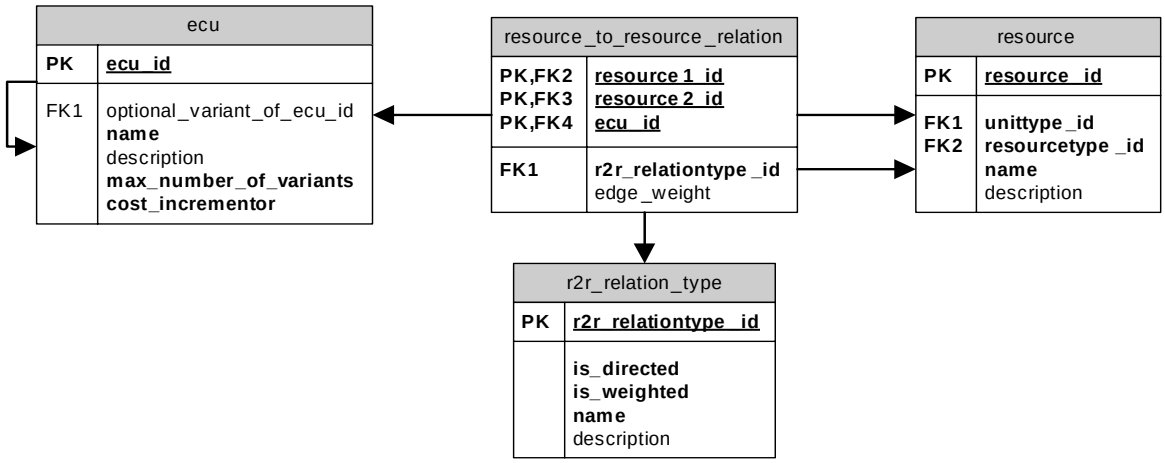


Figure B.2.: Resource-to-resource relationships

reverse tuple. If the type is weighted, the the edge_weight attribute in the resource_to_resource_relation can provide a weight, which defaults to 0.

These relations have particularly been introduced in order to integrate OS-resources and non-functional properties [LSPS04]: The fine evaluation of resource consumptions for the ECU's OS and middleware layers itself requires a differentiated modelling of OS-resources like different available task scheduling algorithms, as well as their influence on memory consumption or CPU. In fact, an enhanced feature trees would be required, as have been introduced in section 4.5 for the features of a vehicle that can be ordered by a customer.

## Framework Configuration

In order to allow some reproduction of the results, the framework configuration should be stored for a mapping solution, as outlined in figure B.3. The framework_runtime contains meta information like the date of execution or the user_name of the person in charge.

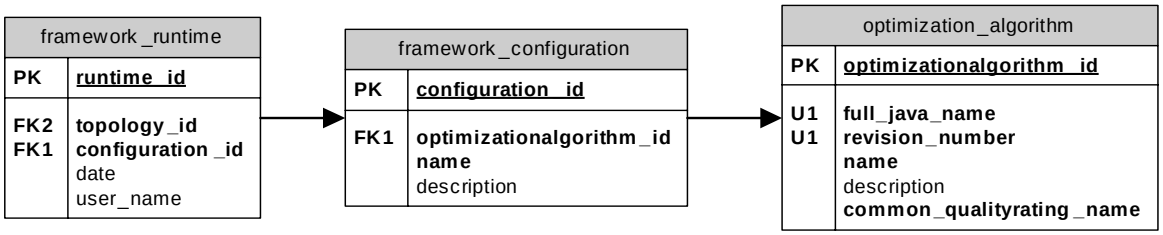


Figure B.3.: Storing the framework configuration

The reference to the topology is contained, because optimization might be executed for them individually. The framework_configuration is very preliminary and just contains a description and a reference to the applied optimization_algorithm. The algorithm is also indentified by a full_java_name and revision_number, as have been the objective estimation implementations in section 4.3. The common_qualityrating_name would, for example, be "fitness" for an evolutionary algorithm.

## B.2. UML Class Diagrams

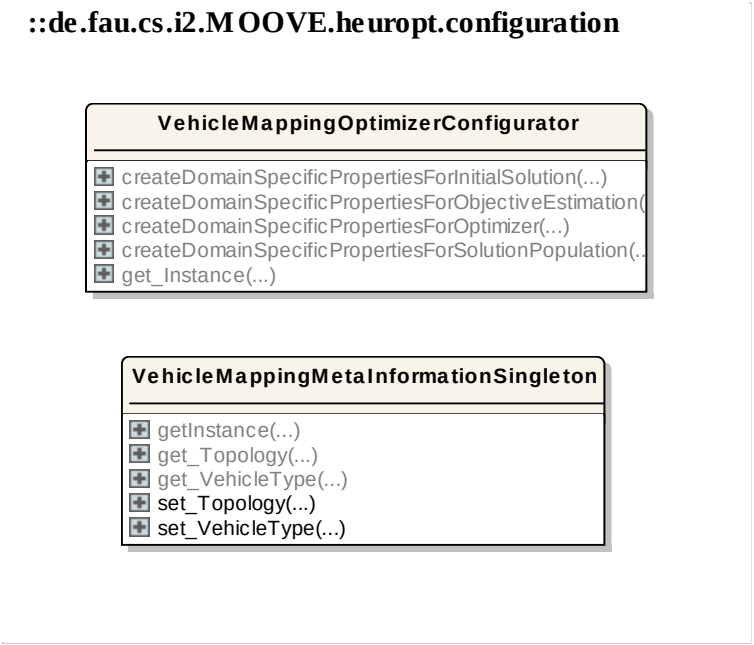


Figure B.4.: Package: de.fau.cs.i2.MOOVE.heuropt.configuration

**::de.fau.cs.i2.MOOVE.heuropt.population**


Figure B.5.: Package: de.fau.cs.i2.MOOVE.heuropt.population

## B.3. Template Listings

```
package de.fau.cs.i2.MOOVE.heuropt.template;

import java.util.HashMap;

import de.fau.cs.i2.MOOVE.heuropt.population.VehicleMappingPopulation;
import de.fau.cs.i2.heuropt.kernel.population.SolutionPopulation;
import de.fau.cs.i2.heuropt.pareto.SolutionSet;

public class TemplateVehiclePopulation extends VehicleMappingPopulation {

    public final static Class DEFAULT_SOLUTION_CLASS = TemplateVehicleSolution.class;

    public Class getDefaultSolutionClass() {
        return DEFAULT_SOLUTION_CLASS;
    }

    protected void praeinit_handleSolutionSet(HashMap props) {
        // lnkSortedSolutionSet = new GenericSPEA2SortedSolutionSet();
        // lnkSortedSolutionSet = new GenericParetoSortedSolutionSet();
        super.praeinit_handleSolutionSet(props);
    }

    protected void initSubDomainSpecific(HashMap props) {
        super.initSubDomainSpecific(props);
        // Need more initialization data?
    }

    public SolutionPopulation regeneratePopulation() {
        SolutionSet sc = (SolutionSet) this.lnkSortedSolutionSet.cloneDeeply();

        // ... create next population ...

        return this;
    }
}
```

Listing B.1: MOOVE specific template code listing for a population class

```
package de.fau.cs.i2.MOOVE.heuropt.template;

import de.fau.cs.i2.MOOVE.heuropt.population.VehicleGlobalMappingSolution;
import de.fau.cs.i2.util.mapping.I2TAttributedFullMap;
import de.fau.cs.i2.util.mapping.TargetSet;
import de.fau.cs.i2.util.mapping.TargetingitemSet;

public class TemplateVehicleSolution extends VehicleGlobalMappingSolution {

    public void operation() {
        I2TAttributedFullMap rawMapping = this.get_RawMapping();
        TargetingitemSet targetingItemSet = rawMapping.get_TargetingitemSet();
        TargetSet targetSet = rawMapping.get_Targets();

        /* decide which TargetingItem to Target mapping(-edges) are to be changed */
        // ...

        /* at the end: update the Mapping with EdgeOperations: */
//        EdgeOperation edgeOP = new EdgeOperation(.., .., OperationType.ADD);
//        this.apply(edgeOP);
//        EdgeOperationSet edgeOP = new EdgeOperationSet();
//        this.applyAll(edgeOP_set);
    }

}
```

Listing B.2: MOOVE specific template code listing for a solution class

# Bibliography


[AEE02] AEE Konsortium. Architecture Électronique Embarquée, 2002. URL `http://aee.inria.fr`, last visited at 2005-04-13.

[Apa05a] Apache Software Foundation. Torque – a persistence layer, 2005. URL `http://db.apache.org/torque/`, last visited at 2005-02-05.

[Apa05b] Apache Software Foundation. Xerces Java parser – the Apache XML project, 2005. URL `http://xml.apache.org/xerces-j/`, last visited at 2005-14-26.

[AUT04] AUTOSAR. Automotive Open System Architecture, October 2004. URL `http://www.autosar.de/`, last visited at 2005-05-12.

[Azz04] Azzuri. Clay database modeling in Eclipse, 2004. URL `http://www.azzurri.jp/en/software/clay/index.jsp`, last visited at 2005-05-19.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems*. Oxford University Press, New York, USA, 1999.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons, New York, USA, 1996.

[Bor05] Borland. Together v6.2, 2005. URL `http://www.borland.com/together/`, last visited at 2005-04-20.

[BS98] James E. Beck and Daniel P. Siewiorek. Automatic configuration of embedded multicomputer systems. In *IEEE Transactions on computer-aided design of integrated circuits and systems*, volume 17, issue 2, pages 84–95, February 1998. URL `http://ieeexplore.ieee.org/iel4/43/14978/00681259.pdf`, downloaded at 2005-05-28.

[Bur60] Bureau International des Poids es Mesures. Système international d'unités, 1960. URL `http://www.bipm.org/en/si/`, last visited at 2005-05-20.

[CDM91] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the European Conference on Artificial Life*, pages 134–142, Paris, France, 1991. URL `http://dsp.jpl.nasa.gov/members/payman/swarm/colorni92-ecal.pdf`, downloaded at 2005-02-05.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, England, 2000.

[Coe00] Carlos A. Coello. An updated survey of ga-based multiobjective optimization techniques. *ACM Computing Surveys*, 32(2):109–143, 2000. URL `http://doi.acm.org/10.1145/358923.358929`, downloaded at 2005-05-12.

[Deb01] Kalyanmoy Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, Chichester, England, 2001.

[Ecl05] Eclipse Foundation. Eclipse: Extensible Java IDE, 2005. URL `http://download.eclipse.org/downloads/index.php`, last visited at 2005-05-15.

[Fle04] FlexRay Consortium. Flexray: The communication system for advanced automotive control applications, 2004. URL `http://www.flexray-group.com/`, last visited at 2005-02-16.

[FRHW00] Ulrich Freund, Thomas Riegraf, Markus Hemprich, and Kai Werther. Interface based design of distributed embedded automotive software – the TITUS approach. *VDI-Berichte 1547, VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany*, pages 105–123, October 2000.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, England, 1994.

[HIS03] HIS – Hersteller Initiative Software. Vector CAN-driver specification v1.0, August 2003. URL `http://www.automotive-his.de/download/HIS_CAN_Specification_V1_0.pdf`, downloaded at 2004-11-03.

[HIS04] HIS – Hersteller Initiative Software. API IO driver v2.1.3, April 2004. URL `http://www.automotive-his.de/download/API_IODriver_2_1_3.pdf`, downloaded at 2005-04-13.

[HKK04] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. Reuse of software in distributed embedded automotive systems. In

*Proceedings of the 4th ACM International Conference on Embedded Software, EMSOFT, Pisa, Italy*, pages 203–210, September 2004. URL `http://www.cc.gatech.edu/classes/AY2005/cs4220_spring/papers/10A-hardung-reuse.pdf` , downloaded at 2005-02-01.

[HKKK05] Bernd Hardung, Thomas Kollert, Gabriella Kókai, and Andreas Krüger. Optimization of variants of electronic control units under consideration of customer orders. To appear in *VDI-Berichte, VDI Congress Electronic Systems for Vehicles, Baden-Baden*, October 2005.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.

[HWK+03] Bernd Hardung, Matthias Wernicke, Andreas Krüger, Gerhard Wagner, and Florian Wohlgemuth. Development Process for Networked Electronic Systems. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany*, pages 77–97, September 2003.

[ITE04] ITEA. EAST-EEA: Embedded architecture and software technology – embedded electronic architecture; ITEA-Project-No. 00009, 2004. URL `http://www.east-eea.net/` , last visited at 2005-02-16.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990. URL `http://www-ivs.cs.uni-magdeburg.de/~lauterba/doc/FODA.pdf` , downloaded at 2005-03-24.

[KE95] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks (ICNN'95)*, volume IV, pages 1942–1948, Perth, Australia, 1995. URL `http://www.engr.iupui.edu/~shi/Coference/psopap4.html` , downloaded at 2005-05-02.

[KFAK05] Stefan Kubica, Wolfgang Frieß, Christian Allmann, and Thorsten Kölzow. Domain-crossing software product lines in embedded, automotive systems. To appear in *Proceedings of the International Embedded Systems Symposium (IESS'05)*, August 2005.

[LIN00] LIN Consortium. LIN: Local Interconnect Network, 2000. URL `http://www.lin-subbus.org/` , last visited at 2005-02-16.

[LSPS04] Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. On the design and development of a customizable embedded operating system. In *Proceedings of the International Workshop on Dependable Embedded*

*Systems, 23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, October 2004. URL http://www4.informatik.uni-erlangen.de/~lohmann/download/WDES04_Lohmann.pdf , downloaded at 2005-05-24.

[Mic03] Microsoft. Visio, 2003. URL http://office.microsoft.com/de-de/FX010857981031.aspx , last visited at 2005-05-19.

[MM99] C. E. Mariano and E. Morales. MOAQ: an ant-Q algorithm for multiple objective optimization problems. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceeding of the Genetic and Evolutionary Computation Conference (GECCO-99)*, volume 1, pages 894–901, Orlando, Florida, USA, 1999. Morgan Kaufmann. URL http://w3.mor.itesm.mx/~emorales/Papers/gecco99.ps , downloaded at 2005-03-14.

[MOS05] MOST Cooperation. MOST: Media oriented system transport, 2005. URL http://www.mostcooperation.com/ , last visited at 2005-02-16.

[Moy98] J. Moy. RFC 2328 – OSPF version 2, April 1998. URL http://www.faqs.org/rfcs/rfc2328.html , last visited at 2005-05-29.

[OY05] Özgür Yeniay. Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and Computational Applications*, 10(1):45–56, 2005. URL http://www.asr.org.tr/pdf/vol10no1p45.pdf , downloaded at 2005-02-15.

[Rie03] Matthias Riebisch. Towards a more precise definition of feature models. In M. Riebisch, J. O. Coplien, and D. Streitferdt, editors, *Modelling Variability for Object-Oriented Product Lines*, pages 64–76, Norderstedt, Germany, 2003. URL http://www.theoinf.tu-ilmenau.de/~riebisch/publ/06-riebisch.pdf , downloaded at 2005-05-06.

[Rob91] Robert Bosch GmbH. CAN: Controller Area Network, 1991. URL http://www.can.bosch.com/ , last visited at 2005-02-16.

[SAV+00] P. S. Shelokar, S. Adhikari, R. Vakil, V. K. Jayaraman, and B. D. Kulkarni. Multiobjective ant algorithm: combination of strength Pareto fitness assignment and thermodynamic clustering. In *Foundations of Computing and Decision Sciences*, volume 25, No. 4. Institute of Computing Science, Poznań University of Technology, Poland, 2000. Special issue on "Evolutionary and Local Search Heuristics in Multiple Objective Optimization", Part I. URL http://www.cs.put.poznan.pl/fcds/2000.htm , last visited at 2005-05-20.

[Sie04] Siemens AG. OSEK/VDX: Open systems and the corresponding interfaces for automotive electronics, 2004. URL `http://www.osek-vdx.org/`, last visited at 2005-02-16.

[Sou04a] SourceForge.net. SQL2JAVA – SQL to Java code generator, August 2004. URL `http://sql2java.sourceforge.net/`, last visited at 2005-05-19.

[Sou04b] SourceForge.net. XORM – extensible object-relational mapping, May 2004. URL `http://xorm.sourceforge.net/`, last visited at 2005-05-19.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley & Sons, New York, USA, 2000. Reprinted with corrections April 2001.

[TEF+03] Thomas Thurner, Joachim Eisenmann, Ulrich Freund, Roland Geiger, Michael Haneberg, Ulrich Virnich, and Stefan Voget. The EAST-EEA project – a middleware based software architecture for networked electronic control units in vehicles. *VDI-Berichte 1789, VDI Congress Electronic Systems for Vehicles, Baden-Baden, Germany*, pages 545–563, September 2003.

[TH02] Steffen Thiel and Andreas Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, 2002. URL `http://www.cin.ufpe.br/~in1045/papers/art28.pdf`, downloaded at 2005-05-20.

[Vec04] Vector CANtech. DaVinci tool suite: Functional software design for distributed automotive systems, 2004. URL `http://www.vector-cantech.com/products/davinci.html`, last visited at 2005-02-16.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Eidgenössische Technische Hochschule Zürich (ETH), Zurich, Switzerland, 2001. URL `http://e-collection.ethbib.ethz.ch/show?type=incoll&nr=324&part=text`, downloaded at 2005-03-13.