The handle http://hdl.handle.net/1887/30105 holds various files of this Leiden University dissertation.

**Author:** Etemadi Idgahi (Etemaadi), Ramin  
**Title:** Quality-driven multi-objective optimization of software architecture design: method, tool, and application  
**Issue Date:** 2014-12-11
This chapter defines the foundations on which this dissertation is built and introduces the basic concepts used terminology.

This chapter is structured as follows. Section 2.1 defines the optimization problem and more specifically multi-objective optimization problem. It also gives the definition of Pareto dominance for ordering of solutions in multi-objective optimization solutions and the Pareto front as the result of the optimization process. Section 2.2 defines software architecture, software component and component-based software architecture. These are very critical aspects of our approach and in our framework. The notion of quality of software architecture is described in Section 2.3 which is a fundamental concept in our optimization approach. Finally, Section 2.4 defines software product line, feature and product which are used in Chapter 8.

2.1 Evolutionary Multi-objective Optimization (EMO)

According to Bäck [Bäc96], Evolutionary Algorithms are based on a model of natural, biological evolution, which was formulated for the first time by Charles Darwin. The "Darwinian theory of evolution" explains the adaptive change of species by the principle of natural selection, which favours those species for survival and further evolution that are best adapted to their environmental conditions. In addition to selection, the other important factor for evolution is the occurrence of small, apparently random and undirected variations between the phenotypes. These mutations prevail through selection, if they prove their worth in light of the current environment; otherwise, they perish.
2.1.1 Genotype and Phenotype

According to [Bäc96] and [Mit98], the Evolutionary Algorithms are based on *genes* as transfer units of heredity. Genes are occasionally changed by mutations. The genotype of an individual using bit strings is simply the configuration of bits in that individual’s chromosome. Selection acts on the individual (the individual is the unit of selection), which express in its phenotype the complex interaction within its *genotype*; i.e., its total genetic information, as well as the interaction of the genotype with the environment in determining the *phenotype*. The evolving unit is the *population* which consists of a common gene pool included in the genotypes of the individuals.

One of the distinctive features of the Genetic Algorithm (GA) approach is to allow the separation of the representation of the problem from the actual variables in which it was originally formulated. In line with biological usage of the terms, it has become customary to distinguish the ‘genotype’ - the encoded representation of the variables, from the ‘phenotype’ - the set of variables themselves.

2.1.2 Optimization Problem

According to Kruisselbrink [Kru12], an optimization problem is a triple \((X, F, G)\), where:

- \(X\) is the search space, which is the non-empty set of all possible solutions.

- \(F = \{f_1, \ldots, f_k\}, k \in \mathbb{N}\), is a set of one or more objective functions that are to be minimized. Each objective function is a function of the form \(f : X \rightarrow \mathbb{R}\) that maps elements of the search space to a score value.

- \(G = g_1, \ldots, g_p, p \in \mathbb{N}\), is a set of constraint functions that need to be satisfied. Each constraint function is of the form \(g : X \rightarrow \mathbb{R}\) mapping elements of the search space to a constraint value. For a certain input \(x \in X\) a constraint \(g\) is said to be satisfied if and only if \(g(x) \geq 0\). Otherwise, if \(g(x) < 0\), then solution \(x\) violates the constraint and is therefore infeasible.

2.1.3 Multi-objective Optimization Problem

A multi-objective optimization problem is an optimization problem with more than one objective function. For instance, for the triple \((X, F, G)\), the set \(F\) consists of at least two objective functions [Kru12].

For multi-objective optimization problems, the definition of optimality is often based on the notion of Pareto dominance on the objective space. Pareto dominance introduces a partial order on the space of objective function values, being \(\mathbb{R}^k\) for a problem with \(k\) objectives. In the context of minimization, this order is defined as the following section.
2.1.4 Pareto Dominance

We follow the definition of Kruisselbrink [Kru12] who defines Pareto dominance as follows. For any two vectors $u$ and $v$:

- $u$ dominates $v$ (notation $u \prec_{\text{Pareto}} v$ or just $u \prec v$) iff:
  \begin{align}
  &\forall i \in \{1, \ldots, k\} : u_i \leq v_i \\
  &\text{and } \exists j \in \{1, \ldots, k\} : u_j < v_j
  \end{align}

- $u$ weakly dominates $v$ (notation $u \preceq v$) iff:
  \begin{align}
  &u \prec_{\text{Pareto}} v \lor u = v
  \end{align}

- $u$ strictly dominates $v$ iff:
  \begin{align}
  &\forall i \in \{1, \ldots, k\} : u_i < v_i
  \end{align}

- $u$ and $v$ are incomparable (notation $u \parallel v$) iff:
  \begin{align}
  &u \not\preceq v \land v \not\preceq u
  \end{align}

The partial order introduced by using the notion of Pareto dominance on the solution space can be used to define the goal of multi-objective optimization as to find Pareto optimizers:

2.1.5 Pareto Optimum

For a multi-objective optimization problem $(X, F, G)$, a point $x$ in $X$ is called "efficient", if it is feasible with respect to $G$ and there is no other point $x'$ in $X$ with $F(x') \prec_{\text{Pareto}} F(x)$. The point $F(x)$ in the objective space is then called a Pareto optimum.

2.1.6 Pareto Front

For a multi-objective optimization problem $(X, F, G)$, the set of all Pareto optima is called the "Pareto Front" and the set of all efficient points in $X$ is called the "efficient set".

With the definition for a multi-objective optimization problem, and the goal to find either one, or multiple, Pareto optima, the basic notions of optimization have been introduced.
2.1.7 Hypervolume

A variety of quality indicators is used in order to measure the quality of Pareto front approximations. Among them, the hypervolume indicator is of outstanding importance. It is a quality indicator that rewards the convergence towards the Pareto front as well as the representative distribution of points along the front. The hypervolume measure was originally proposed by Zitzler and Thiele [ZT98], who called it the size of dominated space. Let $\Lambda$ denote the Lebesgue measure, then the hypervolume measure ($\phi$ metric) is defined as:

$$\phi(B, y_{ref}) = \Lambda \left( \bigcup_{y \in B} \{y' | y \prec y' \prec y_{ref}\} \right), B \subseteq \mathbb{R}^m$$

(2.6)

Here, $y_{ref} \in \mathbb{R}^m$ denotes a reference point that should be dominated by all Pareto-optimal solutions.

2.2 Software Architecture and Software Components

2.2.1 Software Architecture

Numerous definitions for software architecture have been formulated, and the research community has not achieved final agreement on a common wording. In the following, we give two commonly accepted definitions for software architecture.

**Software Architecture (Definition 1)** A general definition, that is used in this dissertation, emphasizes design decisions:

"A software system’s architecture is the set of principal design decisions made about the system" [TMD10].

Interestingly, what is a principal design decision depends on the system goal. Examples names by Taylor et al. [TMD10] are the structure of the system, important decisions on functional behaviour, the interaction of components, and non-functional properties. This definition only mentions the core concept of design decision. It is independent of the question how these design decisions are formulated, and thus includes intangible software architectures that are not documented. Thus, this definition separates the software architecture from its representation.

**Software Architecture (Definition 2)** A commonly accepted definition of software architecture is given in ISO/IEC/IEEE standard 42010. A software system’s architecture is:
"[t]he fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" [ISO11].

Because it concerns design at the software component level (as opposed to the design of those components themselves), software architecture is a pivotal vehicle to address and guarantee non-functional requirements. Since the interest in software architecture research has increased, several important concepts were introduced [Hei12].

For example, the influential 4 + 1-view model [Kru95] (as can be seen in Figure 2.1) expounded that, for representational clarity and the purpose of completeness, a software architecture is to be described according to predefined views. These views are defined so that they each accommodate the different issues that stakeholders have. The feedback that these stakeholders are then able to give is thought to benefit the fitness and other general design aspects of the software architecture.

2.2.2 Software Component

The term ‘software component’ is used with a somewhat different meaning in different fields of software engineering. In this section we first explain the view of components as result from decomposing the system. Next, we discuss software components as unit of composition.

An important subset of design decisions refer to the structure of the system, i.e., its decomposition into building blocks. To manage the complexity of software systems, architects apply the principles of encapsulation, abstraction and modularity to structure the system [TMD10]. The resulting building blocks are called software component: "A software component is an architectural entity that (1) encapsulates a subset of the system’s functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context" [TMD10].
Researchers have strived to define a notion of software component which has as main objective to enable the composition of systems from independently developed components. Szyperski et al. [Szy98] have identified the following characteristics of a software component that can be independently developed and reused, stressing the composability and reuse by third parties:

**Software Component** A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Szy98].

According to Szyperski extended definition, a software component is:

- A subject for multiple use. A software component should be designed and implemented such that its functionality can be reused in many different systems.

- An externally stateless entity. A component should not expose its execution state to a system and can be bound, started and stopped at any moment of a system lifecycle.

- Composable with other components. A component provides well-specified interfaces, by which it can be bound to other components.

- An encapsulated entity, i.e. a component internal implementation cannot be explored through its interfaces.

- A unit of independent deployment. All component dependencies on external resources are clearly specified and it can be substituted by some other component.

Because component’s aim to encapsulate their internal implementations from the outside world, they expose their functionality and connectivity specification via interfaces. A component interface is a set of named operations with specified signatures, through which it can interact with other components. As a special case, a component offers access to its functionality via its interfaces.

A component may have two types of interface: *provided* and *required* interfaces. Whereas a provided interface specifies the functionality that a component offers to the environment, a required interface specifies a component’s requirements to the environment that have to be satisfied for its proper operation. More specifically, required interfaces are ports through which a component can invoke operations provided by other component interfaces. At component deployment, a required interface can be bound to a provided interface of another component.
**Component Model**  A component model determines what is and what is not a component. Heineman and Councill [HC01] define a component model as follows:

"A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment" [HC01].

This definition points out that a component model covers multiple facets of the development process, dealing with (i) rules for the construction of individual components, and (ii) rules for the assembly of these components into a system.

### 2.2.3 Component-Based Software Architecture

A software architecture that is structured based on software components and connectors is called a component-based software architecture in the following.

**Component-Based Software Architecture**  A component-based software architecture is a software architecture whose principal design decisions regarding the structure of the systems are made by structuring the system as a set of software components. The system is thus described as a composition of components [Koz11].

To express (component-based) software architectures, architects have to describe the architecture in some type of artefact. These artefacts can be ranging from natural language descriptions over UML models [OMGa] to formal architectural description languages such as the Palladio Component Model [BKR09].

**Architecture Model**  An architecture model is an artefact that captures some or all of the design decisions that compromise a system’s architecture [TMD10].

**Component-Based Architecture Model**  A component-based architecture model is a formal architecture model that uses software components as the main entity to describe the design decisions: (1) software component are explicit model entities which encapsulate internal decisions and provide information on interfaces and dependencies, (2) the model of a component can be reused in any CBA model, (3) structural design decisions are expressed as a composition and assembly of software components, only making use of the provided interfaces and context dependencies of the component models, and (4) other design decisions are described in relation to the composition or to the components (e.g. by annotating components, connectors, or assemblies) [Koz11].
2.3 Quality of Software Architecture

Developing high quality software products is a goal in many development projects. However, quality is a highly subjective term and depends on the goals and perceptions of stakeholders. The software architecture of a system is critical for a system to meet its quality-objectives. Thus, quality should be considered when designing software architecture.

To better reason about software product quality, software quality models have been suggested to describe and measure software quality, e.g. ISO/IEC 9126-1:2001 [ISO01]. (See [BKLW95] for more information)

Software quality attributes (also called quality characteristics or quality properties) are characteristics which provide the basis for evaluating quality of software systems. Examples of software quality attributes of software systems are performance, safety, reliability, maintainability, usability, and cost. Software quality attributes are one of the influencing factors to take into account when designing a software architecture [BCK03]. For some software quality attributes, quantitative quality metrics are available to assess the level of quality achieved by a software system.

Often, software quality objectives are found to be in conflict during the architectural design activity: For example, security and reliability often negatively influence each other: While a system is secure if it offers few places that keep sensitive data, such an organization may lead to single points of failure and decreased reliability. Furthermore, almost all software quality attributes conflict with performance [Koz11]. The art of architecting is to find suitable trade-off’s between multiple mutually conflicting quality objectives.

2.4 Software Product Line (SPL)

The Software Product Line (SPL) approach is aimed at the development of a set of products within a well-defined domain by leveraging their commonalities and managing their variabilities in a systematic way in order to obtain large-scale reuse [vdLSR07].

2.4.1 Software Product Line

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Software product lines engineering (SPLE) is a software engineering paradigm institutionalizing reuse throughout the software life-cycle. Linden et al. [vdLSR07] define SPLE as a systematic approach for software reuse applied to a family of specific products within a well-defined domain.
For example, a TV set, as a consumer electronic device, uses software. In the modern TV production processes, SPL has been heavily employed. Various types of TV sets have lots of commonality in core functionalities but they might be vary in terms of internet connectivity, or capability of installing applications, or quality of display panel (e.g. HD, Full HD or 4K).

2.4.2 Feature

According to FODA (Feature-Oriented Domain Analysis) [KCH+90], "A feature is a prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems". Feature models allow visualization, reasoning, and configuration of large and complex software product lines.

2.4.3 Product

Each product is formed by taking suitable features from the base of common assets, tailoring them as necessary through variation mechanisms, and then assembling the collection according to the rules of a common product line wide architecture [GH11]. Ideally, building a new product becomes more a matter of assembly or generation than one of creation: the predominant activity is integration rather than programming.