Multi-Quality Auto-Tuning by Contract Negotiation

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Sebastian Götz
geboren am 19.05.1984 in Dresden

Betreuender Hochschullehrer:
Prof. Dr. rer. nat. habil. Uwe Aßmann
(Technische Universität Dresden)

Tag der Verteidigung: Dresden, den 17. Juli 2013

Dresden im April 2013

Confirmation

I confirm that I independently prepared this thesis with the title *Multi-Quality Auto-Tuning by Contract Negotiation* and that I used only the references and auxiliary means indicated in the thesis.

Dresden, June 21, 2013

Dipl.-Inf. Sebastian Götz

Abstract

"The only constant is change."

— Heraclitus, *500 B.C.*

A characteristic challenge of software development is the management of omnipresent change. Classically, this constant change is driven by customers changing their requirements. The wish to optimally leverage available resources opens another source of change: the software systems environment. Software is tailored to specific platforms (e.g., hardware architectures) resulting in many variants of the same software optimized for different environments. If the environment changes, a different variant is to be used, i.e., the system has to reconfigure to the variant optimized for the arisen situation. The automation of such adjustments is subject to the research community of self-adaptive systems. The basic principle is a control loop, as known from control theory. The system (and environment) is continuously monitored, the collected data is analyzed and decisions for or against a reconfiguration are computed and realized. Central problems in this field, which are addressed in this thesis, are the management of interdependencies between non-functional properties of the system, the handling of multiple criteria subject to decision making and the scalability.

In this thesis, a novel approach to self-adaptive software–Multi-Quality Auto-Tuning (MQuAT)–is presented, which provides design and operation principles for software systems which automatically provide the best possible utility to the user while producing the least possible cost. For this purpose, a component model has been developed, enabling the software developer to design and implement self-optimizing software systems in a model-driven way. This component model allows for the specification of the structure as well as the behavior of the system and is capable of covering the runtime state of the system. The notion of quality contracts is utilized to cover the non-functional behavior and, especially, the dependencies between non-functional properties of the system. At runtime the component model covers the runtime state of the system. This runtime model is used in combination with the contracts to generate optimization problems in different formalisms (Integer Linear Programming (ILP), Pseudo-Boolean Optimization (PBO), Ant Colony Optimization (ACO) and Multi-Objective Integer Linear Programming (MOILP)). Standard solvers are applied to derive solutions to these problems, which represent reconfiguration decisions, if the identified configuration differs from the

current. Each approach is empirically evaluated in terms of its scalability showing the feasibility of all approaches, except for ACO, the superiority of ILP over PBO and the limits of all approaches: 100 component types for ILP, 30 for PBO, 10 for ACO and 30 for 2-objective MOILP. In presence of more than two objective functions the MOILP approach is shown to be infeasible.

Acknowledgment

First of all, I'd like to thank my girlfriend, Jennifer Varga, for her patience, *pedagogic* advice and constant support during all phases of my Ph.D. Furthermore, I'd like to thank my parents, Steffi and Matthias Götz, for their support, too. Without their help the last three years would have been much more demanding.

The same holds for my friends. In particular Christin Müller, Melanie Ragotzki and Alice Hentrich. Although our "meetings" could hardly be considered scientific, without them it would have been much harder to regain my strength whenever a paper got rejected or I required distraction to break a writer's block.

Many colleagues, which became friends, provided valuable discussions and some made the big effort to cross-check the final draft of the thesis. Amongst them I'd like to thank Sebastian Richly, Claas Wilke, Christian Piechnick, Sven Karol and Christoff Bürger. I'm particularly thankful to Claas Wilke, for his efforts on the initial versions of the contract language and component model.

I'm especially indebted to Sebastian Richly and Ilie Savga for introducing me into the field of research and providing me the opportunity to participate in paper writing already as an undergraduate.

Finally, but not lastly, I'd like to thank my supervisor, Uwe Aßmann, for his support and, especially, the many discussions, which did not always reveal helpful immediately, but in the end were invaluable.

Sebastian Götz
April 2013

Publications

This thesis is partially based on the following peer-reviewed publications:

- Sebastian Götz, Claas Wilke, Matthias Schmidt, Sebastian Cech and Uwe Aßmann: *Towards Energy Auto Tuning.* In: Proceedings of First Annual International Conference on Green Information Technology, GREEN IT 2010, GSTF (2010) 122–129.
- Sebastian Götz, Claas Wilke, Sebastian Cech and Uwe Aßmann: *Runtime Variability Management for Energy-efficient Software by Contract Negotiation.* In Proceedings of 6th International Workshop on Models@run.time, ACM/IEEE (2011) 61–72.
- Sebastian Götz, Claas Wilke, Sebastian Cech and Uwe Aßmann: *Architecture and Mechanisms of Energy Auto Tuning.* In Sustainable ICTs and Management Systems for Green Computing, IGI Global (2012) 45–73.
- Sebastian Götz, Claas Wilke, Sebastian Richly and Uwe Aßmann: *Approximating Quality Contracts for Energy Auto-Tuning Software.* In Proceedings of First International Workshop on Green and Sustainable Software, IEEE (2012) 8–14.
- Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin and Mario Trapp: *Uses and Purposes of M@RT Systems.* To be published in State-of-the-Art Survey Volume on Models@run.time. Springer LNCS. 2013.
- Sebastian Götz, Claas Wilke, Sebastian Richly, Georg Püschel and Uwe Aßmann: *Model-driven Self-Optimization using Integer Linear Programming and Pseudo-Boolean Optimization.* To be published in Proceedings of Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE'13), IARIA XPS Press (2013).

The following peer-reviewed publications cover work that is closely related to the content of the thesis, but not contained herein:

- Georg Püschel, Sebastian Götz, Claas Wilke and Uwe Aßmann: *Towards Systematic Model-based Testing of Self-adaptive Systems.* To be published in Proceedings of The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE'13), IARIA XPS Press (2013).

– Claas Wilke, Sebastian Richly, Sebastian Götz and Uwe Aßmann: *Comparing Mobile Applications' Power Consumption.* To be published in Proceedings of the 28th Symposium On Applied Computing (SAC'13), ACM (2013).

– Claas Wilke, Sebastian Götz and Sebastian Richly: *JouleUnit: A Generic Framework for Software Energy Profiling and Testing.* In Proceedings of the 1st Workshop "Green In Software Engineering Green By Software Engineering" (GIBSE'13), ACM (2013) 9–14.

– Claas Wilke, Sebastian Richly, Georg Püschel, Christian Piechnick, Sebastian Götz and Uwe Aßmann: *Energy Labels for Mobile Applications.* In Proceedings of 42nd GI Jahrestagung INFORMATIK'12, GI (2012) 412–426.

– Julia Schroeter, Sebastian Cech, Sebastian Götz, Claas Wilke and Uwe Aßmann: *Towards Modeling a Variable Architecture for Multi-Tenant SaaS-Applications.* In Proceedings of Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12), ACM (2012) 111–120.

– Christian Piechnick, Sebastian Richly, Sebastian Götz, Claas Wilke and Uwe Aßmann: *Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation - Smart Application Grids.* (Best Paper Award) In Proceedings of the Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE'12), IARIA XPS Press (2012) 93–102.

– Friedrich Gräter, Sebastian Götz and Julian Stecklina: *Predicate-C - An Efficient and Generic Runtime System for Predicate Dispatch.* In Proceedings of the 6th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'11), ACM (2011) 2.

– Sebastian Götz, Max Leuthäuser, Jan Reimann, Julia Schroeter, Christian Wende, Claas Wilke, Uwe Aßmann: *A Role-based Language for Collaborative Robot Applications.* In Leveraging Applications of Formal Methods, Verification, and Validation (CICS), Springer (2011) 1–15.

– Claas Wilke, Sebastian Götz, Jan Reimann and Uwe Aßmann: *Vision Paper: Towards Model-Based Energy Testing.* In Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11), Springer (2011) 480–489.

Contents

1. Introduction **1**
1.1. Motivation, Challenges and Problems . . . 3
1.2. Overview of the Approach . . . 6
1.3. Contributions . . . 11
1.4. Organization of the Thesis . . . 13

I. Foundations and Related Work **15**

2. Background **17**
2.1. Self-adaptive Software . . . 17
2.1.1. Architecture of Self-Adaptive Software: The Feedback Loop . . . 19
2.1.2. The Incarnations of Self-Adaptive Software: Self-* Properties . . . 20
2.1.3. The Challenge of Multi-Objective Optimization in SAS . . . 21
2.2. Auto-Tuning Approaches . . . 22
2.2.1. Static Auto-Tuning . . . 22
2.2.2. Dynamic Auto-Tuning . . . 23
2.2.3. Comparison to Self-Adaptive Software . . . 24
2.3. Combinatorial Optimization . . . 25
2.3.1. Single-Objective Optimization . . . 25
2.3.2. Meta-heuristic Optimization . . . 29
2.3.3. Multi-Objective Optimization . . . 32

3. A Taxonomy of Adaptation Reasoning **35**
3.1. Facets of Adaptation Reasoning . . . 35
3.2. Related Research Projects . . . 38
3.3. Summary . . . 52

II. A Development Methodology for Self-Optimizing Software Systems 53

4. A Multi-Quality-aware Software Architecture 55
4.1. The Cool Component Model . . . 57
4.1.1. Structural Models: System Architecture Description . . . 57
4.1.2. Variant Models: Abstract Runtime System Representation . . . 60
4.1.3. Behavior Models: Quality-specific Operational Semantics . . . 62
4.1.4. Shared Platform Concepts . . . 65
4.1.5. Special Purpose Packages . . . 69
4.2. The Quality Contract Language . . . 73
4.3. Architectural Aspects of Multi-Objective Optimization . . . 78
4.4. Summary . . . 80

5. Refinement of Platform-Specific Quality Contracts 81
5.1. The Process of Contract Generation . . . 83
5.2. Threats to Validity . . . 89
5.3. Summary . . . 92

6. From Contract Checking to Economic Multi-QoS Contract Negotiation 93
6.1. Contract Checking . . . 95
6.2. Economic Multi-Quality Contract Negotiation . . . 102
6.3. Summary . . . 104

III. A Runtime Environment for Self-Optimizing Software Systems 105

7. Exact Approaches for Multi-Quality Auto Tuning 107
7.1. Contract Negotiation by Integer Linear Programming . . . 108
7.1.1. The Rational of Decision Variables . . . 109
7.1.2. Generation of Objective Functions . . . 111
7.1.3. Constraint Generation . . . 114
7.1.4. ILP Generation by Example . . . 116
7.2. Contract Negotiation by Pseudo-Boolean Optimization . . . 119
7.2.1. Reformulation of the Configuration Problem in PBO . . . 120
7.2.2. PBO Generation by Example . . . 124
7.3. Scalability Evaluation . . . 124
7.3.1. Generation of Test Systems for Empirical Evaluation . . . 124
7.3.2. Measurements for Selected Types of Generated Systems . . . 128
7.4. Summary . . . 136

8. An Approximate Approach for Multi-Quality Auto-Tuning **139**
8.1. Contract Negotiation by Ant Colony Optimization 140
8.1.1. Generating the Optimization Problem for Ant Colonies 140
8.1.2. ACO Generation by Example 145
8.2. Scalability Evaluation 145
8.3. Summary 149

9. An A Posteriori Multi-Objective Optimization Approach for Multi-Quality Auto-Tuning **151**
9.1. Contract Negotiation by Multi-Objective Integer Linear Programming . . 152
9.1.1. Solving Multi-Objective Integer Linear Programs 152
9.1.2. Klein and Hannan by Example 154
9.1.3. The Confidential Sort Example 156
9.2. Scalability Evaluation 158
9.3. Summary 163

10. Conclusion and Future Work **165**
10.1. General Conclusion 165
10.2. Limitations and Future Work 168

Appendix **169**
A1. Concrete Syntax of the Quality Contract Language 169

1 Introduction

The future of software systems is predicted to be characterized by ubiquitous, interconnected software components, which run on and use several heterogeneous resources, are subject to frequent changes and optimize themselves w.r.t. their non-functional behavior [56]. Finkelstein and Kramer identified *Change*, *Non-functional Properties*, *Non-classical Life-cycles* (evolving systems) and *Configurability* amongst other as key challenges of future software engineering [56, p. 7]. Although these challenges have been identified already in 2000, more recent studies show that these challenges still exist. In 2007, Kramer and Magee outline the development of self-managed systems as a key challenge in software engineering [79]. In 2009, Cheng et al. provided a roadmap expressing the demand for self-adaptive systems.

Thus, the immanent problem of future software engineering is the shift from isolated (client-server) software running on stationary devices to software running on vast amounts of interconnected devices providing new ways of interaction with the real world. Such *ubiquitous* software demarcates from classical client-server software in the requirement to be able to adjust itself to changing contexts. "Context is any information that can be used to characterize the situation of an entity" [46, p.5]. According to Zimmermann et al. [132], the situation covers five aspects: individuality (the entity's self), activity (the task or goal of the entity), location, time and relations with other entities. That is, future software needs to be able to adapt itself w.r.t. to its own state, its goals, its location, time and relationships. Such software is called *self-adaptive software (SAS)*.

SAS is always embedded into a context and usually serves users with differing needs or expectations. In terms of Zimmermann et al. [132], the software and its users denote entities, which adapt to each other by forming a dynamic relationship. Moreover,

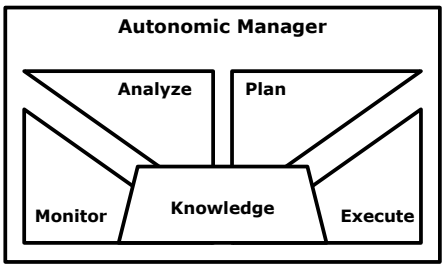


Figure 1.1.: MAPE-K Loop [98]: Monitor - Analyze - Plan - Execute - Knowledge.

the software is in relationship with its execution platform (i.e., the utilized hardware, operating system and middleware). The goals of SAS correlate with the self-* properties [112] of SAS: self-configuring, self-healing, self-protecting and self-optimizing. The self-configuring property of software denotes the bare ability of software to reconfigure (without any specified intent). In contrast, self-healing, -protecting and -optimizing software utilizes this ability to reach a specified goal: the system shall be able to recover from failures (self-healing), to resist security threats (self-protecting) or serve its users in the best-possible way (self-optimizing). Self-healing is an objective for systems, which have to ensure operability, but do not necessarily have to optimally utilize available resources. Future software systems, instead, should optimally leverage the available resources to provide the best possible user satisfaction. Moreover, user's typically have multiple, potentially competing objectives. For example, users of an audio file optimization system are likely to ask for as fast processing as possible, but for as best file compression and audio quality as possible, too. These three objectives contradict with each other. For example, better compression requires more time and potentially decreases the audio quality. Hence, the focus of this thesis is on multi-objective self-optimization.

The field of self-optimization in software engineering divides by the time when optimization is performed: at design time, compilation time, deployment time or at runtime. Software optimization at design time reaches from architecture optimization (e.g., ArchOpterix [3] and PerOpteryx [78]) to compiler optimizations and auto-tuning techniques (cf. Sect. 2.2). As future software systems are expected to be challenged by unanticipated changes at runtime [56], in this thesis, the focus is on self-optimization *at runtime*.

The basic principle of all SAS is a feedback loop, like the MAPE-K loop introduced by Oreizy et al. [98] depicted in Figure 1.1, which is comprised of four phases: a monitor phase, where the system is observed, an analyze phase, where the monitored data is evaluated, a plan phase, where appropriate adaptation decisions are planned and an execution phase, where the planned adaptation decisions are performed.

In this thesis, a novel approach to runtime, multi-objective self-optimizing software–Multi-Quality Auto-Tuning (MQuAT)–is presented, which provides design and operation principles for software systems which automatically provide the best possible utility to

the user while producing the least possible cost. It addresses the first three phases of the feedback loop, i.e., monitor, analyze and plan.

In this chapter, first the problems and challenges addressed in this thesis are discussed in Section 1.1, followed by an overview of the approach in Section 1.2. Finally, Section 1.3 summarizes the contributions given in this thesis and Section 1.4 outlines the organization of the overall thesis.

1.1. Motivation, Challenges and Problems

A classifying characteristic of self-optimizing software systems is their specification of optimality. For example, software can be optimized for maximum performance, maximum reliability, minimum energy consumption or minimum price. For each of these goals, a single non-functional property (NFP), such as performance or reliability, is optimized. Moreover, optimization can consider multiple qualities in combination, e.g., maximizing performance whilst minimizing energy consumption. Here, an important distinction of runtime software optimization approaches is easy to see: either the aim is to optimize for maximum (user) utility or for efficiency by interpreting qualities as utilities and/or costs, i.e., the best possible utility for the least possible cost. For example, the optimality of a configuration of an image analysis application, comprised of a set of software components having multiple implementations each, depends on whether the application is intended to process the images as fast as possible, as accurate as possible or as fast *and* accurate as possible. Each combination of the optimization objectives potentially leads to another optimal configuration, which denotes the selection of implementations and their mapping to execution environments.

The general objective of self-optimizing software systems is to determine and reconfigure to the optimal system configuration. Hence, techniques from combinatorial optimization are required, which in general seek an optimal object in a finite set of objects [96]. Thus, self-optimizing software systems need to consider multiple qualities in combination and have to interpret qualities as costs and utilities in order to optimize their trade off (i.e., the efficiency of the system). The following challenges arise from this problem:

1. The optimization problem needs to be formulated. But, there is no generic formulation of such optimization problems. In literature only problem-specific formulations exist (e.g., for vehicle networks [58]), which prescribe a fixed structure of the system elements to be considered. For example, in [4], a software component is restricted to have a memory requirement, a communication frequency and an event size only. If additional properties of software components are to be considered, the approach has to be extended or even revised. Hence, an approach capable of generating optimization problem specifications based on structural and runtime models of the system is required.

2. NFPs are not comparable per se. The characteristics of NFPs need to be considered in order to correctly compare NFPs or to interpret the interrelation of NFPs. Notably, NFPs can impact each other, and exist on different levels of abstraction. For example, on a high level of abstraction, NFPs like energy consumption or safety levels exist, whereas on a lower level NFPs like CPU time, disk read throughput and response time exist. Moreover, a low remaining battery capacity can endanger safety in electronic vehicles. Thus, interdependencies between NFPs have to be considered.

3. The computational complexity of combinatorial optimization techniques is very high (i.e., the optimization problems are known to be NP-hard) [96]. This poses the challenge to derive a correct reconfiguration plan *in time* or, more general, within an available *budget*, which might comprise further resources in addition to time. Thus, the challenge is to assess the scalability and predictability of the applied optimization techniques.

In addition to solutions for these problems and challenges at runtime, an appropriate design methodology is required. The central challenge for such a methodology is the need for *concise* specifications of the configuration space in order to cope with the combinatorial explosion of possible system configurations. A proper design methodology is the key to effective and efficient solutions for the runtime challenges as the design delimits the opportunities to reconfigure the system under development.

This thesis addresses three major problem areas in the field of self-optimizing software systems. First, a development method, which allows for the assessment of system configurations in terms of utilities and costs. Second, the application of optimization techniques at runtime and third, the scalability evaluation of these optimization techniques. Each area is described in further detail in the following:

Development Method Enabling Configuration Assessment in Terms of NFPs (cf. Chapters 4, 5 and 6). In this thesis, the first software architecture for self-optimizing software systems, which covers the dependencies and interactions of its NFPs, is presented. Chapter 4 introduces this architecture, which offers structural, behavioral and runtime views of the system.

In addition, a Quality of Service (QoS) contract language is presented, which allows to specify quality interaction and enables the interpretation of NFPs as costs and utilities. But, NFPs highly depend on the hardware being used and the way users interact with the software (i.e., NFPs depend on context in general). This prohibits the developer to give a complete specification of quality interactions and their interpretation as efficiency (utility/cost) values of the possible configurations of an SAS using QoS contracts at design time. Hence, in Chapter 5, this problem is addressed by an approach which measures and analyzes the NFPs at runtime and approximates the resource- and user-specific NFPs by multiple linear regression. The assessment of system configurations

based on the multi-quality-aware software architecture and the runtime QoS-contract refinement approach is part of Chapter 6.

Application of Optimization Techniques at Runtime (cf. Chapters 7, 8 and 9). A well known problem of SAS in general is the infeasibility to enumerate all system configurations in order to identify the best amongst them. This is due to the combinatorial explosion of possible configurations. Hence, the system's structural and runtime view need to allow optimization techniques to iteratively search for an optimal solution, in a way that potentially not every configuration needs to be investigated. In the presented approach, this is achieved by the specification of quality interactions and dependencies in QoS contracts.

A special problem related to runtime optimization is the computational complexity of both exact and approximate techniques in combinatorial optimization, because both are known to be NP-hard. This complexity cannot be avoided, but imposes a trade off between accuracy and complexity. To reduce complexity, heuristics are applied instead. But, heuristics cannot guarantee an optimal solution. Nevertheless, they are likely to achieve near-optimal solutions. A problem for both kinds of optimization techniques is the need to consider non-linear system behavior, which further raises the computational complexity. Especially performance and energy optimization, which consider the schedule of operations, require to reason about non-linear behavior. For example, energy consumption by utilizing a CPU is not necessarily proportional to the execution time of a process, as the CPU might switch its performance state while the process executes.

Finally, the combined optimization of multiple qualities requires techniques from multi-objective optimization (MOO), which raises the computational complexity of optimization even further. Approaches to MOO divide into two classes: *a priori* and *a posteriori* approaches. The demarcating characteristic is the way multiple objectives are handled. In a priori approaches, the objectives are merged in advance to the search for the optimum. In contrast, in a posteriori approaches, the objectives functions are merged after the search for an optimum. In terms of computational complexity, a posteriori approaches are worse than a priori approaches.

In this thesis, exact MOO approaches capable of interpreting qualities as costs or utilities are presented in Chapter 7 (an apriori approach) and 9 (an a posteriori approach). An approximate (meta-heuristic) approach is shown in Chapter 8.

Optimization Evaluation and Scalability (cf. Chapter 7, 8 and 9) The quality of the solutions of optimizers needs to be evaluated, but in case of approximate optimization techniques, this is not possible per se. For example, if the potential gain in efficiency due to reconfiguration is less than the accuracy of the approximate approach, the reconfiguration can potentially even degrade the system's efficiency.

Moreover, reconfiguration itself influences the QoS of the system subject to optimiza-

tion (e.g., reconfiguration consumes time, energy and could endanger safety). Considering the QoS effects of reconfiguration (and the preceding steps), a budget can be identified, which must not be exceeded in order to ensure the effectiveness of self-optimization. This demands for scalable runtime optimization. For this reason, the scalability of each optimization approach presented in this thesis is analyzed at the end of Chapter 7, 8 and 9, respectively.

1.2. Overview of the Approach

The approach presented in this thesis is called Multi-Quality Auto Tuning (MQuAT) and comprises a design time and a runtime part. The first part covers a new development method for self-optimizing systems. The second part concerns operation principles, namely, novel techniques to runtime self-optimization. Both parts are outlined in the following.

Design Principles for Self-Optimizing Software Systems

The development method proposed by MQuAT extends on Component-Based Software Development (CBSD) [117]. That is, software subject to self-optimization is proposed to be built from components with explicitly defined boundaries. Moreover, components are meant to comprise multiple implementations each providing the same functionality but differing in their non-functional behavior. This design principle is crucial as it is the key enabler for runtime optimization, because only if differently behaving configurations exist a reconfiguration to a better or optimal configuration is possible. For each component the NFPs of interest are to be specified. This enables the assessment and, in consequence, comparability of implementations of the same component.

These basic principles are covered by the Cool Component Model (CCM) [64]: a meta-architecture of self-optimizing software systems.

A specialty of MQuAT is the application of QoS contracts to cover the non-functional behavior of implementations as well as the interrelations between NFPs of different components. Contracts naturally describe the connection between provisions and requirements. In MQuAT, contracts are used to specify the provision of NFPs by an implementation if a set of specified requirements is hold. Notably, requirements can be specified against resources as well as other software components. The consequence of using QoS contracts, which quantitatively describe the dependencies between implementations and resources as well as other implementations is leveraged by the operation principles, namely, more efficient planning/decision approaches.

In this thesis, a special kind of QoS contracts has been developed, based on prior work by Röttger et al. [111]. In addition, a language to specify these contracts, called Quality Contract Language (QCL) [61], has been developed. The concept of QoS contracts and QCL are described in detail in Section 4.2.

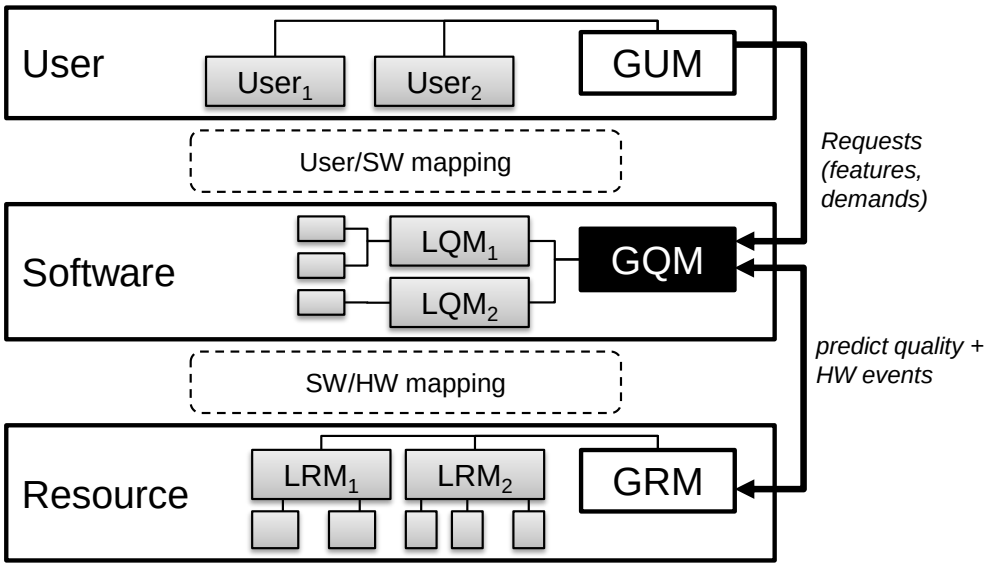


Figure 1.2.: The Layers of Multi-Quality Auto-Tuning Systems.

Operation Principles for Self-Optimizing Software Systems

The core runtime concept of the self-optimization approach proposed in MQuAT is the THE Auto-Tuning Runtime Environment (THEATRE) [64, 61]. This runtime environment can be divided into three layers: a user, a software, and a resource layer as depicted in Figure 1.2.

The top-most layer covers the users of the system, which invoke features and describe their demands and (potentially multiple) objectives in terms of NFPs w.r.t. the invoked feature. The Global User Manager (GUM) is used to coordinate the requests and to manage the mapping between users and software component implementations (i.e., which components are used by which users). For this purpose, a textual language able to express the feature invocation and the demands of users has been developed. User requests are minimal requirements to the software system, which have to be fulfilled in order to satisfy the user. The general objective of the runtime system is to serve the user as efficient as possible in terms of the user's objectives.

The middle layer comprises software components which are controlled by a Local Quality Manager (LQM). For each component container a separate instance of an LQM exists. The task of an LQM is

1. to "know" which components are deployed on its container, i.e., to provide an appropriate variant model,

2. to deploy and undeploy component implementations on the container and

3. to shut down or startup the container itself.

In addition to the LQMs, a single Global Quality Manager (GQM) exists in the runtime environment. Its task is to react on user requests and on changes to the system infrastructure (i.e., the addition of a new server or the invocation of a provided feature by a user). Whenever such an event occurs, it

1. collects information about the current state of all resources, the currently running software component implementations, and how they are mapped to the available resources,

2. calculates the current efficiency of the system and starts to search for a more efficient system configuration,

3. decides for or against a reconfiguration and derives the actual steps required to perform the reconfiguration and

4. delegates the tasks to perform the reconfiguration to the respective LQMs.

Thus, in short, the task of the GQM is to realize the analyze and plan phases of the feedback loop, whereas the LQM is responsible for the execution phase. The monitor step is performed by another manager, which is introduced in the next paragraph.

The bottom layer comprises physical (e.g., hard disk drive) as well as virtual resources (e.g., operating system). Each resource has its own Local Resource Manager (LRM). The entirety of all resources is monitored and controlled by the Global Resource Manager (GRM). The tasks of an LRM include in-depth knowledge about its resource and the ability to steer the resource by, for example, switching between power saving modes or shutting down/starting up the resource. For example, the values of monitored NFPs are provided by LRMs. The task of the GRM is to know which resources are connected to the system infrastructure, how they can be used and to delegate commands to all connected resources. For example, if the GQM decides for a reconfiguration which relieves a previously heavily utilized server from all its tasks, this server can be shut down to save energy. The GQM will send this command to the GRM, which in turn delegates the task to the respective LRM of the server. In [63] a detailed discussion about the interface and tasks of the GRM and LRM is given.

In summary, the envisioned runtime environment is comprised of multiple local quality and resource managers and one global resource and one global quality manager. The task of focus in this thesis is performed by the GQM, which realizes the analyze and plan steps of the feedback loop.

The runtime process of MQuAT as depicted in Figure 1.3 comprises five steps:

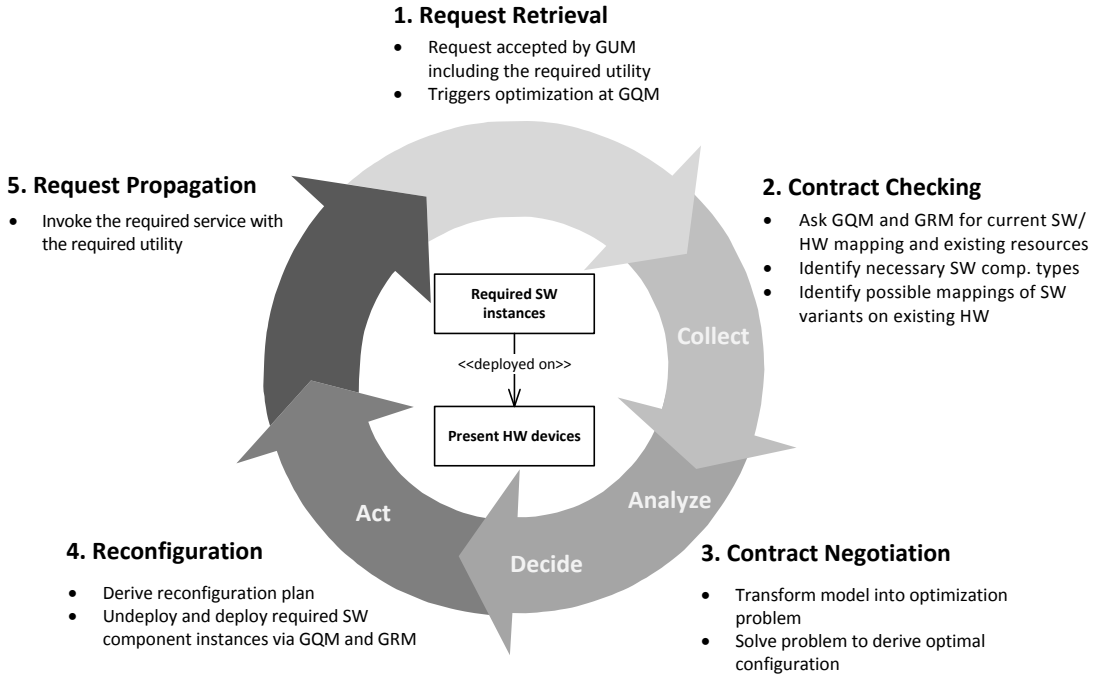


Figure 1.3.: Global Picture of MQuAT.

1. **Request Retrieval**: If a user requests a service, the service request is retrieved by the first central manager of the auto-tuning runtime environment (THEATRE), the GUM. Besides the service the user intents to invoke, the user can specify NFPs he requires for the service (e.g., minimum refresh rates).

2. **Contract Checking**: Subsequently, the GUM delegates the request to the GQM to check if the system has to be reconfigured to efficiently serve the user request (e.g., if new software component instances have to be deployed or parts of the system have to be reconfigured to provide the requested NFPs). During contract checking, the GQM searches for the required software components (which can have multiple implementation variants providing different NFPs) and requests the GRM for the currently present hardware devices and their available resources (e.g., free disk space, memory or CPU cycles). The GQM computes all possible mappings of existing software variants onto the existing hardware landscape. How the available software and hardware variants and their dependencies are modeled using CCM and QCL is described in Chapter 4.

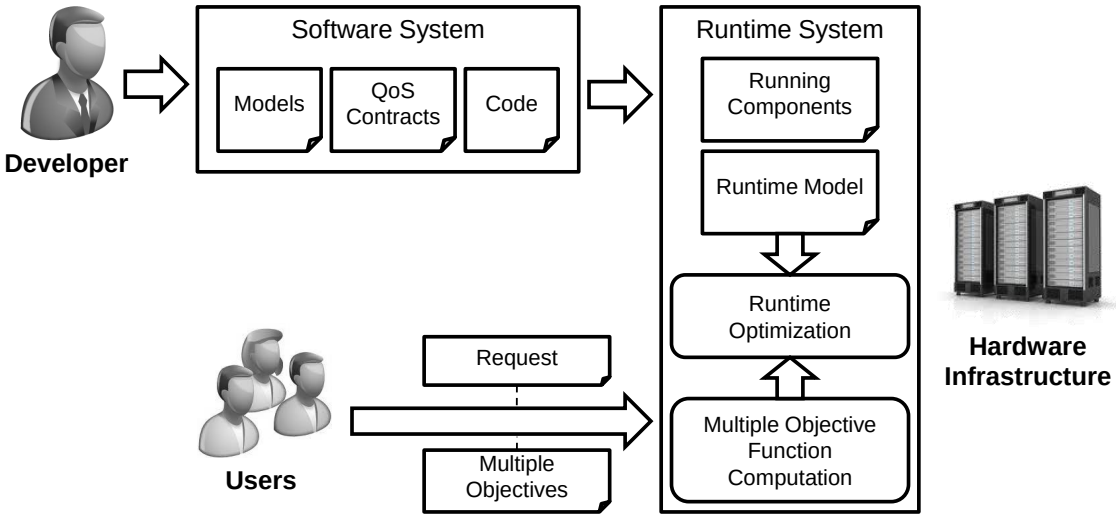


Figure 1.4.: Combined Design- and Runtime Perspective of the Approach.

3. **Contract Negotiation**: The requirements of the user request and the user's objectives together with the possible software-to-hardware mappings are negotiated (i.e., which system configuration is the best in terms of the users demands and objectives). The negotiation is realized by a transformation to a mathematical optimization problem that is passed to a standard solver to compute the optimal system configuration w.r.t. the user's demands in terms of an optimal trade off between utility and cost.

4. **Reconfiguration**: The result from the contract negotiation phase is used to compute a reconfiguration plan denoting which software component variants have to be undeployed, moved, and deployed to provide the requested service and utility. Afterwards, the reconfiguration plan is executed by the GQM.

5. **Request propagation**: The original user request is forwarded to the now optimal deployed software landscape providing the required service and utility.

Quality assessment denotes the mechanism used by THEATRE and its managers to collect information on the available software component implementations, hardware devices, their energy consumption and performance to name but a few. It runs in parallel to the feedback loop and allows to always register new software implementations and to plug or unplug hardware devices.

Summarizing, Figure 1.4 provides an overview of MQuAT combining the design time and runtime perspective. The developer creates models and quality contracts of the system in addition to the actual code. The users interact with the system at runtime

and formulate their objectives and requests. The runtime system comprises a runtime model, reflecting the current state of the system, in addition to the running components. Prior to optimization, the user's objectives are transformed to objective functions of an optimization program. Depending on the type of optimization technique used, either all objectives of the users are merged into a single objective function, or one objective function per objective is derived. The process of runtime optimization utilizes these objective functions and the runtime model of the system to generate formulations of the optimization problem for the respective technique. Finally, included by the runtime optimization step, the system reconfigures, if the computed optimal configuration differs from the current system configuration.

1.3. Contributions

This thesis contributes to the field of self-optimizing software systems in three of the four phases of self-adaptive systems: collection, analysis and decision making. The act phase is considered as state of the art, as it relies on component (re-)deployment, which has been used in various previous research projects (e.g., DiVA [57]). The central research question addressed in this thesis is *how to develop, assess and evaluate the runtime system of adaptive software to keep it automatically efficient.*
The main contributions are:

- a new taxonomy of self-optimizing software systems, which exceeds existing taxonomies by the inclusion of runtime optimization issues.
- a development methodology for self-optimizing software systems and
- four evaluated multi-quality, multi-objective runtime software optimization approaches.

In the following, a list of minor contributions per major contribution and chapter is given. Whereas the first major contribution is covered by Chapter 3, both other contributions comprise multiple minor contributions across multiple chapters. Chapter 6 is a bridge between the design time and runtime part of this thesis, by discussing how to leverage the architecture introduced in the first part of the thesis at runtime.

Development Methodology

- Chapter 4
  - A self- and context-aware, component-based software architecture for self-optimizing software systems.
  - An extended contract concept (and language) covering the complex interdependencies between NFPs, cost and utility.

- Chapter 5
  - A process to specify QoS contracts at design time and to approximate their concrete instantiations at runtime.
  - A statistical approach to assess the quality of QoS contracts.
- Chapter 6
  - An argumentation on how contract negotiation using QoS contracts advances over state of the art.
  - A new approach to merge multiple incomparable objectives by discrete event simulation using energy consumption as an example.

Runtime Optimization Approaches

- Chapter 7
  - An exact runtime optimization approach using Integer Linear Programming (ILP).
  - An exact runtime optimization approach using Pseudo-Boolean Optimization (PBO) [21].
  - A scalability analysis and comparison of both approaches.
- Chapter 8
  - An approximate runtime optimization approach using Ant Colony Optimization (ACO).
  - A scalability and accuracy analysis of ACO in relation to the previous approaches.
- Chapter 9
  - An exact, a posteriori, multi-objective runtime optimization approach using Multi-Objective Integer Linear Programming (MOILP).
  - A scalability analysis of this approach.

1.4. Organization of the Thesis

This thesis is organized in three parts as depicted in Figure 1.5. The first part covers the foundations of the presented approach in Chapter 2 and provides an overview as well as a demarcation from related work by introducing a novel taxonomy for self-optimizing systems in Chapter 3. The design methodology is covered by the second part. First, a multi-quality aware software architecture is presented in Chapter 4, followed by an approach elaborating on the process of QoS contract creation in Chapter 5. In Chapter 6, an approach to contract checking and the difference to multi-quality auto-tuning by contract negotiation are presented. The third part of this thesis focuses on runtime optimization techniques for self-optimizing systems. In Chapters 7, 8 and 9 two exact an approximate and an exact, a posteriori contract negotiation approach are shown. The thesis is closed by giving directives on how to proceed and a conclusion in Chapter 10.

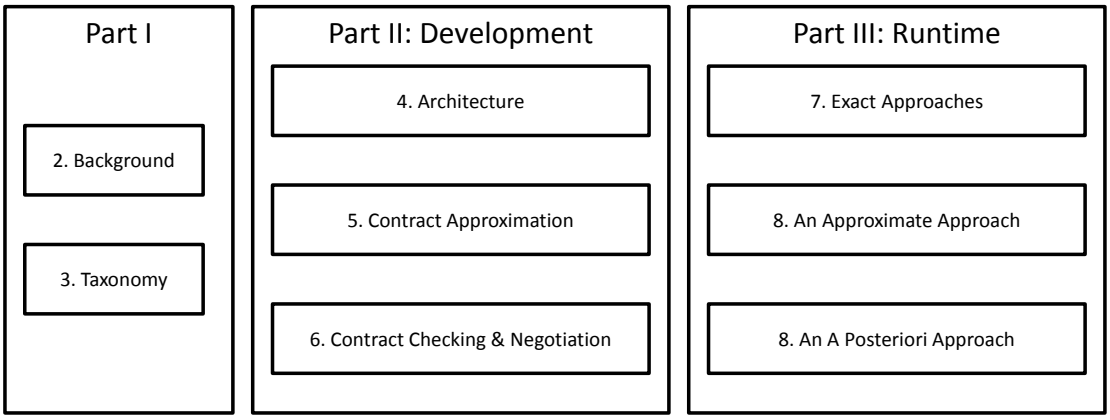


Figure 1.5.: Structural Overview of the Thesis.

Part I.

Foundations and Related Work

2
Background

The thesis at hand is influenced by several research fields, whereof the most important are SAS, auto-tuning and software optimization. Each field divides itself into many research communities focusing on special aspects of the respective field. The general theme underlying this thesis is given by the SAS community, because MQuAT is a special type of SAS and contributes to this community. Auto-tuning, which originates from the High-Performance Computing (HPC) community, is a closely related technique to realize a SAS, which was a main inspiration for this thesis. Finally, software optimization is a field of research which offers several general techniques, which have been investigated, tailored and adapted for MQuAT. This chapter serves the purpose to ease the understanding of the succeeding chapters by summarizing the most important aspects of the three related research fields w.r.t. this thesis. In the following, first SAS will be discussed in Section 2.1, followed by an overview of auto-tuning in Section 2.2. Finally, in Section 2.3, optimization techniques utilized in this thesis are introduced.

2.1. Self-adaptive Software

The concept of SAS reaches back to the mid 90's of the last century. Since then, various definitions for SAS have been provided. One of the first is by Robert Laddaga from 1997, who created and managed the self-adaptive software program at DARPA from 1996 till 1999.

> "Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible." [81]

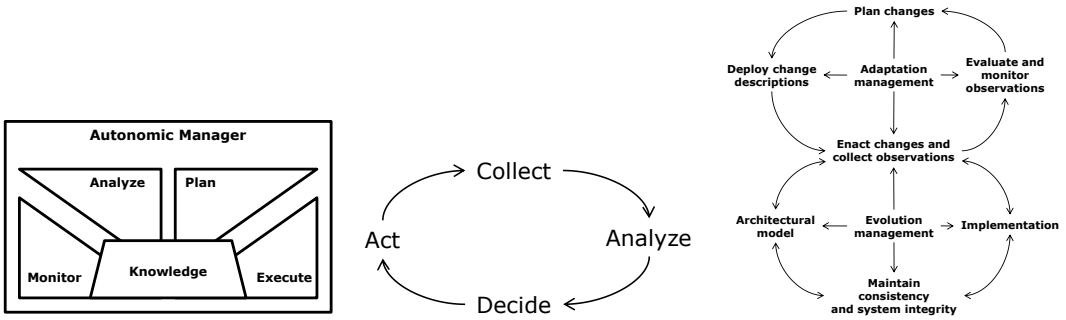
Thus, according to Laddaga, the key characteristics of SAS are self-awareness (i.e., to be able to evaluate the own behavior) and the ability to adjust itself. Another definition for SAS has been provided by Peyman Oreizy in his 1999 article on an architecture-based approach to self-adaptive software in the IEEE Intelligent Systems journal.

> "Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation." [98]

This definition goes further than Laddaga's in that the operating environment is considered in addition. Thus, context-awareness is another key characteristic of SAS. Notably, Oreizy further details how SAS adapt themselves: namely, "... in response to changes ..." and Laddaga implicitly names the continuous reevaluation of an SAS. Both indicate the existence of a feedback loop as a further key characteristic of SAS. That is a SAS continuously adjusts itself to the current context.

In [112], Salehie and Tahvildari provide a concise overview of the current state of the art and research challenges in self-adaptive software systems. They provide a taxonomy of SAS based on the evolution taxonomy by Buckley et al. [27].

The proposed taxonomy specifies four facets of SAS: (1) the object to adapt (e.g., methods and resource adjustments), (2) realization issues including a distinction between dynamic and static decision-making, but also the separation of open and closed systems, (3) temporal characteristics like reactive or proactive adaptation and (4) interaction concerns covering human involvement, trust and interoperability. The second, third and forth facet show a diversification in the research field of SAS. This includes approaches to SAS, which are able to handle the issues of the open world assumption in contrast to classical closed world approaches, approaches providing runtime reasoning capabilities instead of switching between precomputed variants of the systems in a predefined way, approaches enabling proactive adaptation instead of classical reactive behavior and approaches considering human interaction as well as security issues. The approach presented in this thesis can be categorized, according to the taxonomy, as a closed world, dynamic decision-making, reactive approach covering the trade off between multiple, potentially competing, quality concerns.



(a) MAPE-K loop (b) CADA loop (c) Oreizy's Loop

Figure 2.1.: Incarnations of The Feedback Loop for Self-Adaptive Systems.

2.1.1. Architecture of Self-Adaptive Software: The Feedback Loop

The general architecture of self-adaptive software is guided by the incorporation of a feedback loop [112]. Different concretizations of this feedback loop, depicted in Figure 2.1 have been proposed in literature.

Kephart and Chess proposed the MAPE-K Loop, depicted in Figure 2.1(a), comprising a monitor, an analyze, a plan and an execute step as well as a shared knowledge base across all steps [70]. In contrast to [112], Kephart and Chess enumerate research challenges for self-organizing systems (i.e., system constituents are unaware of the complete system) instead of self-adaptive systems (where a holistic view of the system exists).

Another concretization of the feedback loop has been proposed by Dobson et al. in [48]: the CADA loop (c.f. Figure 2.1(b)). According to Dobson et al. the feedback loop comprises four phases: the collection of information about the system (C), its analysis (A), decision-making based on the derived knowledge (D) and acting as realizing the plan decided for (A).

A more complex concretization of the feedback loop is provided by Oreizy et al. [98], depicted in Figure 2.1(c). Oreizy et al. differentiate between high-level adaptation management and low-level evolution management. The adaptation management encompasses to evaluate and monitor observations of the system, to plan changes, and to deploy the change descriptions decided for. These three steps are comparable to collect, analyze and decide of the CADA loop and to monitor, analyze and plan of the MAPE-K loop. Evolution management covers the realization of adaptation decisions (i.e., the act step of the CADA loop or the execute step of the MAPE-K loop respectively). It enacts changes and collects observations in preparation for or in consequence of adaptation management. The major issue of evolution management is identified as maintenance of consistency and system integrity.

Yet another concretization of the feedback loop is provided by Salehie and Tahvildari

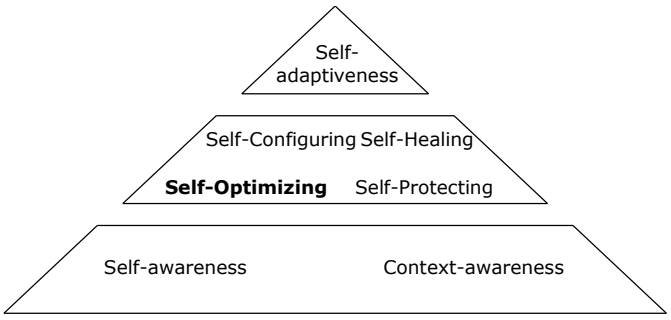


Figure 2.2.: Hierarchy of Self-* Properties. Redrawn from [112, p.5].

in [112], which covers four processes: monitoring, detecting, deciding and acting, which are very close to the elements of the CADA and MAPE-K loop. In addition, sensors and effectors are put into context of these steps, where sensors are utilized by the monitoring process and effectors by the acting process.

Notably, the description of Kephart and Chess on negotiation in autonomic systems highlights the central difference of contract negotiation as investigated in this thesis—global optimization techniques—in contrast to self-organizing systems, where the system elements negotiate with each other aiming for concession using auction-like techniques.

2.1.2. The Incarnations of Self-Adaptive Software: Self-* Properties

Salehie and Tahvildari introduce a three-layer hierarchy of self-* properties, which is outlined in the following to highlight the focus of this thesis in its context, namely self-optimization in the context of self-adaptiveness. It is depicted in Figure 2.2.

The top-most layer is the general level, which covers **self-adaptiveness** as a top-down, centralized type of self-adaptive software and **self-organization** as a bottom-up, decentralized approach to self-adaptive software, where the constituents are unaware of complete system. This thesis focuses on self-adaptiveness.

The middle layer is termed the major level. It comprises four well-known self-properties: self-configuring, self-healing, self-optimizing and self-protecting. Self-configuring denotes the ability of the software to perform adaptation in an automatic way (e.g., by migrating software entities). Self-healing encompasses the automatic identification, anticipation and recovery from disruptions. Self-optimizing, in contrast, denotes "managing performance and resource allocation [...] to satisfy the requirements of different users" [112, p.5]. The authors emphasize that "self-optimizing has a strong relationship with efficiency" [112, p.6]. Finally, self-protecting encompasses the detection, anticipated, prevention, mitigation and/or recovery from security threats. This thesis focuses in particular on self-optimization.

The lowest layer is called the primitive level and covers self-awareness, context-awareness, openness and anticipation as basic mechanisms required for self-adaptive systems.

2.1.3. The Challenge of Multi-Objective Optimization in SAS

One of the key challenges according to [112] is "building multi-property self-adaptive software" [112, p.31], by "finding approximately or partially optimal solutions for multi-objective decision-making problems" [112, p.32]. This thesis particularly focuses on this multi-objective optimization problem.

A first attempt to multi-objective decision-making was presented by Cheng et al. in 2006 [35]. They presented an approach based on utility theory, which is able to consider the trade off between multiple objectives for decision making. The central idea behind the approach was to use utility functions on quality attributes of different dimensions, where these utility functions map the attributes to a decimal value between 0 and 1. This enables the comparison of attributes in terms of utility. Cheng et al. proposed to use the weighted sum of utilities, where each dimensions gets a weight representing its importance to all stakeholders, for decision-making. Notably, this does not reflect the full capability of multi-objective optimization, because the approach merges all dimensions into one for decision-making (weighted sum) and does not provide a set of Pareto optimal solutions. Moreover, the weights of each dimension are not necessarily the same for each stakeholder. The approach by Cheng et al. requires a single weight per dimension for all stakeholders. Hence, part of the trade off negotiation is shifted to the stakeholders, which have to come up with a consensus of single weights. The same problem arises for utility functions, which need to reflect the utility of the whole instead of each stakeholder individually. Nevertheless, Cheng et al. can be considered the first to explicitly address the problem of multi-objective optimization in the area of self-adaptive software.

Notably, the approach to merge different dimensions of the decision problem using utility functions, stems from economics. In [86], Chris Lucas writes about multidimensional economics and addresses the problem that "current economic theory reduces all things to one dimension: that of monetary value [...]" [86] and proposes to use multi-dimensional values for comparisons. Poladian et al. show this problem especially in the context of software engineering [104]. Therefore, they investigate properties of costs. First the divisibility/granularity of costs is differenced in continuous (e.g., energy), dense discrete (e.g., currency) and sparse discrete (e.g., editing a document). Second, fungibility as the property of a cost to be convertible to another cost is differenced into complete fungibility (e.g., common currency), partial fungibility (e.g., bandwidth and CPU cycles if multiple algorithms utilize both resources) and no fungibility (e.g., calendar days and staff months). Third, the measurement scale, divided into nominal, ordinal, integer and ratio, is considered. Fourth, perishability as a property describing whether a resource will be lost if utilized or not (e.g., bandwidth versus energy). Fifth, the economies of scale specifying the impact of the cost is differenced into super linear, linear and sub

linear scale. Finally, Poladian describes the property of being rival for costs. This multitude of properties of costs highlights that costs cannot be easily compared to each other and, hence, a utility function merging different cost dimensions into a single one in general compares apples and oranges. Further approaches, which base on utility theory, but do not explicitly address the multi-objective optimization issue, have been provided by [22, 124, 105, 97, 103].

Interestingly, Cheng et al. already considered these critics to utility theory and followed the proposal of Lucas and Poladian et al. by supporting multidimensional adaptation attributes. Nevertheless, the proposed utility functions still merge all dimensions of the attributes into a single one and no solution to deal with the complexity of correct utility functions taking care of the properties of each cost dimension is provided.

2.2. Auto-Tuning Approaches

Auto-Tuning covers techniques from HPC, which automate the process of performance tuning for scientific applications (e.g., weather forecasts and genome expression analysis). Various approaches have been developed throughout the past decades [128, 18, 99, 106, 119, 90, 110, 122].

The motivation for auto-tuning in HPC is the problem that the frequency of new hardware increases, but the required time to manually tune high-performance code for this new hardware remains unchanged. Hence, approaches to automate the performance tuning for new hardware are needed.

The common way of performance tuning in HPC relies on source code transformations. Thus, the goal of auto-tuning approaches is to find those source code transformations, which improve performance. A basic prerequisite of most auto-tuning approaches is the existence of a kernel library. Such a library contains kernel (i.e., core) algorithms, which are used by scientific applications. Auto-tuning is applied to those kernel libraries instead of the applications themselves. This adheres to standard principles in HPC, where manually optimized kernel libraries are commonly used. The application of auto-tuning enhances these libraries with code transformations, which are adjust to comprised algorithms to the given hardware architecture.

In general, there is a distinction between static and dynamic approaches, which divides the approaches by the time of their decision-making process. This is either at compilation-time, denoting static auto-tuning, or at runtime, denoting dynamic auto-tuning. In the following representative approaches of both types are discussed.

2.2.1. Static Auto-Tuning

The Automatically Tuned Linear Algebra Software (ATLAS) project developed an auto-tuning paradigm called Automated Empirical Optimization of Software (AEOS), which relies on empirical timings to select the best implementation of an algorithm for a

given architecture [128]. The central issue addressed by AEOS and its reference implementation ATLAS is the high effort of manually tuning domain-kernel algorithms for performance. The domain-kernel denotes a selected subset of algorithms or procedures representing the main performance drivers of a particular domain (e.g., linear algebra). The time for new hardware architectures to replace current architectures is getting shorter and shorter, so the time needed for performance tuning—which is by nature hardware-specific—needs to be reduced, too. The principle idea is to investigate the timing behavior of multiple implementations of the same algorithm or procedure on the respective target architecture automatically. To get multiple implementations either a community is engaged or code generators are employed. On the target architecture the implementations are executed and their timing behavior is compared to determine the fastest amongst them. Notably, not every implementation is meant to be profiled in advance, which would be a brute force search. Instead, more sophisticated search algorithms can be employed. The ATLAS reference implementation, for example, first determines cache sizes, the available registers and operations (e.g., combined multiplier and adder) and only generates and profiles[1] those implementations of its provided methods, which potentially provide a good performance for the current architecture [128].

Further static auto-tuning approaches are PHiPAC [18], OptimQR [99] and SPIRAL [106].

2.2.2. Dynamic Auto-Tuning

The FFTW approach [59] represent a dynamic auto-tuning technique, as it postpones optimization to run-time instead of installation/compilation-time. It generates plans at compile-time, which are evaluated at runtime to identify optimal implementations (called *codelet* by Frigo and Johnson) in terms of execution time.

OSKI [123] is another approach to auto-tuning, which adheres to the AEOS paradigm, but postpones the evaluation and selection of the best implementation to runtime and, thus, can be considered a dynamic auto-tuning approach.

A recent approach to auto-tuning is Active Harmony [119]. It specifically exploits information available at runtime and realizes the principle ideas behind AEOS at runtime. Active Harmony focuses specifically on parameters of code sections, which cannot be realized as variables in the code. A typical example for this type of parameters are loop unrolling factors. The goal of Active Harmony is to tune the code of scientific applications on parallel machines at runtime to near-optimal performance. It utilizes runtime code-generation, runtime optimization (search-space pruning) and runtime code transformation or replacement. A notable feature of Active Harmony is that dependencies between parameters as well as constraints between parameters are considered and can be specified by the developer using a constraint language. Dependencies between

[1] The installation process performing these steps takes approximately 2 hours.

	Self-adaptive Systems	Auto-Tuning
Elements	components, features, classes	source code statements
Techniques	migration, selection	compiler optimization, loop unrolling
Principle	multi-variant code	

Table 2.1.: Comparison of Self-adaptive Systems and Auto-Tuning.

parameters are specified by means of mathematical expressions. For decision-making the approach relies on the Parallel Rank Order algorithm presented by Tabatabaee et al. [118] in 2005. A key challenge for dynamic auto-tuning identified by Tiwari and Hollingsworth [119] is the performance of the tuner itself. They address this challenge by an asynchronous connection between the runtime system and the tuning system. The code of the runtime system is updated in a push-based manner, which minimizes the influence of auto-tuning to a minimum. The general idea of dynamic auto-tuning, according to Tiwari and Hollingswirth [119], is the merger of classical compiler optimization and modern just-in-time compilation.

Yet other approaches to dynamic auto-tuning include MATE [90] by Morajko et al., Autopilot [110] by Ribler et al., and ADAPT [122] by Voss and Eigemann.

2.2.3. Comparison to Self-Adaptive Software

Auto-tuning approaches are closely related to SAS in that they realize feedback loops comparable to those of SAS as shown in Figure 2.1 in Sect. 2.1. For example, the CADA loop [48] is realized in the following way: (1) information about the available hardware is *collected*, (2) this information is *analyzed* w.r.t. its effect on the kernel algorithms, (3) a *decision* selecting code transformations improving (or optimizing) the performance of the kernel algorithms is made and (4) the code transformations are applied (*act*).

Thus, auto-tuning can be seen as a special kind of SAS, which operates on source code level with a restricted focus on scientific applications (i.e., HPC). Notably, approaches of the SAS community usually realize the feedback loop on higher levels of abstraction. Commonly, the elements of variation are components, features or classes, whereas auto-tuning works on source code statements. Auto-tuning approaches mainly apply techniques known from compiler optimization like loop unrolling with the goal to identify different variants of the code, which optimally utilize to the underlying hardware (e.g., by not exceeding the number of available registers or the size of available memory). Table 2.1 summarizes the commonality and differences between SAS and auto-tuning.

A considerable question is if such low-level techniques (in terms of abstraction) can be reused on higher levels. An apparent difference between low- and high-level techniques is the complexity of transformations. On a coarse-grain level of abstraction the transformations are usually restricted to the exchange of implementations (or components, features, aspects) and their migration from one resource to another. Low-level techniques seem

to be more complex in that they apply transformations on statements, resulting in new code (e.g., transforming a loop statement to a list of succeeding executions of the loop's body). But, the basic principle is the same: multi-variant code, i.e., in low-level techniques multiple variants of an algorithm are generated (by applying code transformations to an initial version of the algorithm), but the decision used to optimize the algorithm for a given hardware architecture is just the selection of the appropriate variant of the code. Thus, the improvement of auto-tuning w.r.t. SAS is due to the techniques used to generate variants of implementations, but the decision-making is similar to SAS.

An approach to generate variants of implementations is not part of this thesis.

2.3. Combinatorial Optimization

The optimization problems addressed in this thesis belong to the field of combinatorial optimization [96]. The question, which system configuration is best in a current situation, boils down to a search across all possible configurations, which are essentially combinations of decisions. In this thesis, a system configuration denotes a set of *selected* implementations and their *mapping* to resources. Notably, selection and mapping are well known in combinatorial optimization as knapsack problems. The challenge is to select only those implementations, which are required to satisfy the user, and to optimally assign (i.e., map) them to resources whilst minimizing the resulting cost. Hence, the solutions proposed in Chapter 7 are special types of knapsack problems.

In the following an overview of the applied combinatorial optimization techniques is given. First, approaches for single objectives are discussed. Then, approaches able to cover the complex interdependencies of multiple objectives are outlined.

2.3.1. Single-Objective Optimization

In the presence of a single objective function, the optimization goal is either to maximize or minimize this function. For example, a typical objective function is the length of path through a given graph. The shortest path problem denotes the minimization of this objective function for a given start and stop node in the graph.

In general, combinatorial optimization problems are comprised of a set of variables, one or more objective functions, and a set of constraints. The goal is to efficiently determine a variable assignment, which leads to the minimum (or maximum) value of the objective function. More formally, a combinatorial optimization problem comprises a set of decision variables $V = \{v | v \in D\}$, where D is the domain of the variables (e.g., $\mathbb{R}$, $\mathbb{N}$ or $\mathbb{B}$). The objective function is a function $f : V \rightarrow D$, which is either subject to minimization or maximization. Finally, a set of (in-)equality constraints delimits the valid variable assignments.

Depending on the search space characterization (i.e., the variables of the problem), different types of optimization approaches can be applied. In this thesis, the search

space is linear and comprised of real as well as boolean variables. For this type of search space Mixed Integer Linear Programming (MILP) is the standard approach. A typical representation of such problems in mathematics is shown below:

$$min: \quad c^T x (= c_1 x_1 + c_2 x_2 + ... + c_n x_n) \tag{2.1}$$

$$subject\ to: \quad Ax \leq b \tag{2.2}$$

A subtlety of MILP is the restriction to only build sums of variables. That is, the constraints of the optimization problem are bare summations of (weighted) variables and the objective function, too, only sums up the variables. Further operations like multiplication are prohibited. The following example shows a problem for linear programming (note the absence of the integer restriction):

Example 1: *In a factory products of type A and B can be produced. Product A can be sold for 2 EUR each, product B for 3 EUR. There is only one machine, which requires two hours to produce product A and one and a half hour to produce product B. We assume the machine to be available 24 hours a day. Furthermore, the demand for product A is limited by 20 items per day. The demand for B to 10 per day. What should optimally be produced to maximize profit ?*

This problem can be formulated as linear problem with two variables A and $B \in \mathbb{N}$*:*

$$max: 2A + 3B \tag{2.3}$$

$$2A + 1.5B <= 24 \tag{2.4}$$

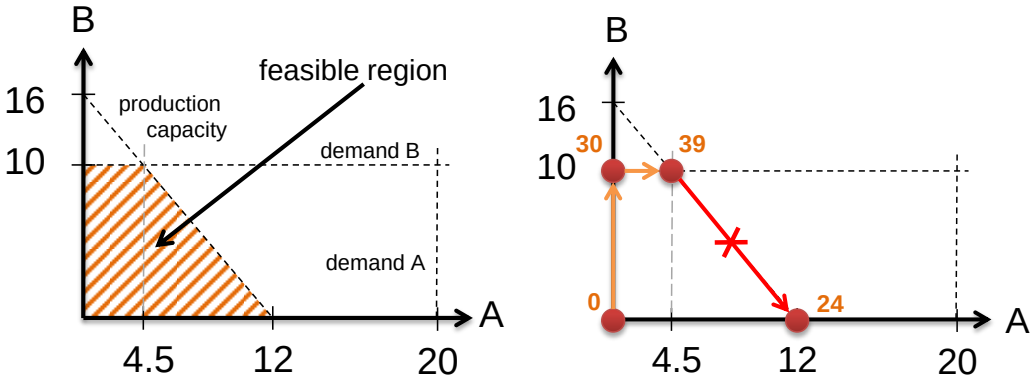
$$A <= 20 \tag{2.5}$$

$$B <= 10 \tag{2.6}$$

The answer is $A = 4.5$ *and* $B = 10$*, as this leads to the highest possible profit (39 EUR). In other words, as products of type B lead to higher profit (3 EUR per item) than those of type A, as many items of B should be produced as possible. Since the demand for items of B is limited to 10 the solution is* $B = 10$*. Producing 10 items of type B takes 15 hours. The remaining 9 hours can be used to produce items of type A. As these items require 2 hours to be produced at most 4.5 items can be produced and* $A = 4.5$*.*

The problem formulated mathematically above is shown graphically in Figure 2.3(a). The set of feasible (i.e., valid) solutions is denoted *feasible region*. Obviously only the boarders of this region need to be investigated to determine a solution. Several approaches exist to determine this solution.

A well-known approach is the *Simplex* algorithm, which has been introduced by Georg Dantzig in 1947 and presented to publicity in a report prepared for the US Air Force in



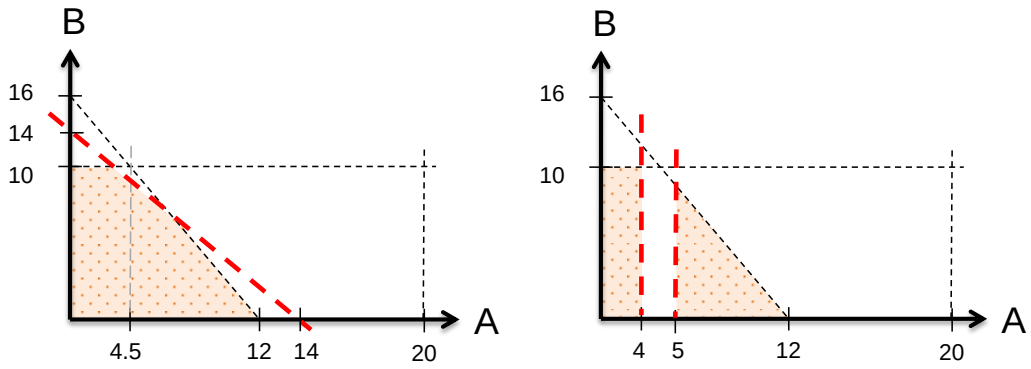
(a) Graphical Representation of the Linear Program

(b) Simplex Approach

Figure 2.3.: Graphical Representation of an Exemplary Linear Program and the Applied Simplex Approach.

1963 [39]. It is known to be of exponential complexity in general. The basic idea of the algorithm is to search for a solution by following the edges of an polyhedron defined by the optimization problem. This principle is shown in Figure 2.3(b) for the two-dimensional polyhedron comprised of the variables A and B. The algorithm consists of two phases: first a start solution has to be identified, then the search for the optimal solution is performed. Various approaches exist to determine a valid start solution. Nevertheless, as in this thesis the *Simplex* algorithm is used, we omit a deeper explanation of these approaches. In the example above, the zero solution is taken as a starting point (i.e., $A = 0, B = 0$ leading to a profit of zero. The second phase is performed by searching for directly connected edges in the polyhedron leading to higher profits. In the example these are either $A = 12, B = 0, profit = 24$ or $A = 0, B = 10, profit = 30$. As the second edge leads to higher profit the algorithm selects this edge and performs the search again. The next solution found is $A = 4.5, B = 10, profit = 39$, which is better than the current solution and, hence, the algorithm selects this edge as current optimal solution. Finally, the last found edge is $A = 12, B = 0, profit = 24$, which is worse than the current solution. In consequence, the algorithm terminates with $A = 4.5, B = 10, profit = 39$ as optimal solution. Please note that a solution found by the *Simplex* algorithm is a *global* optimum. This is because any local optimum of a polyhedron is also a global optimum by definition, because the polyhedrons of linear programs are convex [39].

The central benefit of the *Simplex* algorithm is its ability to handle changes to the optimization problem without the requirement to restart the search for a solution. Another approach is the interior point method, which is known to be of polynomial complexity. Nevertheless, the *Simplex* algorithm is known to perform better in practice [84].



(a) Cutting Plane Approach (b) Branch and Bound Approach

Figure 2.4.: Approaches to Solve Integer Linear Programs.

The example above illustrates a problem of linear programming without additional constraints: the decision variables can be floating point numbers, which is not necessarily intended. The solution $A = 4.5, B = 10$ has to be interpreted as *produce 9 items if type A and 20 of type B in two days* to gain maximum profit. The problem is that an integer solution is desired, as an item can be either produced or not. Half an item cannot be produced. If only one day is available, the found solution opens the question, whether adjusting downward A to $A = 4$ is really an optimal integer solution. Thus, an additional constraint for decision variables to be of integer type is required.

This integer requirement forms a new subtype of linear programming called ILP. If both unrestricted and integer variables exist in the problem, the term MILP is used. Notably, the integer restriction demands for new approaches to solve the optimization problem. These approaches usually first solve the problem without the integer restriction and apply problem transformations if the acquired solution does not adhere to the integer constraints. Two well-known approaches are the *cutting plane* approach and the *branch and bound* approach [96].

The cutting plane approach tries to *cut* out infeasible solutions by adding additional constraints to the problem. In the example from above a constraint which excludes the solution $A = 4.5, B = 10$ would be introduced (e.g., a linear function intersecting $A = 4, B = 10$ and $A = 5, B = 9$). This extension of the original solution is performed until solving the linear program yields a valid integer solution. Figure 2.4(a) depicts the first step of this approach.

The branch and bound approach splits the original linear program if a solution is found which violates the integer restriction. In the previous example the linear program would be split into two linear programs, which are delimited by the (vertical) constraints $A \leq 4$ and $A \geq 5$, respectively. Figure 2.4(b) depicts the first step of this approach.

Current solvers usually combine both approaches, which is termed *branch-and-cut* in literature [96]. Notably, the integer solution to the example above is indeed $A = 4, B = 10$, but using the approaches presented above the validity of this solution is ensured. Amongst a variety of solvers available, in this thesis *LP Solve* has been used [54], because it is freely available and offers many libraries for the integration into programs of various programming languages.

A further specialization of MILPs are so-called 0-1 ILPs. They require variables to be in $\mathbb{B}$, i.e., to be either 0 or 1. The restriction to use boolean variables only allows for specialized solving techniques. Namely, Pseudo-Boolean Optimization (PBO) can be used to solve these 0-1 ILPs. The name of PBO originates from the fact that, although all variables have to be boolean, numeric constants can be used in the optimization problem. A well-known approach for PBO is the application of the Davis-Putnam approach [42] (or DPLL [41]), which is a standard approach to solve satisfiability problems in propositional logics. Such an approach has been described, for example, in [11].

In this thesis, a reduction from a MILP to a PBO formulation of the software self-optimization problem is shown (cf. Section 7.2). Notably, although DPLL is known to be of polynomial complexity, it does not necessarily outperform the algorithms used to solve MILPs (which are known to be of exponential complexity). For evaluation, the standard solver called OPBDP [10] by Peter Barth has been used.

2.3.2. Meta-heuristic Optimization

Meta-heuristics are heuristics applicable to many problem domains [50]. Typical representatives are ant colony optimization and simulated annealing. Both are inspired from nature. ACO simulates ant colonies searching the shortest path from their anthill to food sources and in simulated annealing the annealing of metal atoms leading to a stable, near-optimal crystal structure is simulated. In this thesis, ACO has been chosen as a representative for meta-heuristic optimization approaches and, hence, will be explained in more detail in the following.

As outlined, the principle idea of ACO is inspired from nature: the behavior of ant colonies. An ant colony explores its surrounding for food and water. But, how does each individual ant know where to go and which paths have already been explored? For an efficient search for food and water, the ants need to communicate. This communication is realized by *pheromone trails*. The first ant randomly chooses a path to explore. If it succeeds in finding food or water it returns on the same path. While walking back, the ant distributed pheromones if it was successful. This way, the second ant can "smell" where the first ant went and consider this information for path finding. In consequence, the surrounding of the ant colony is marked with pheromones indicating near-optimal paths to food and water. The paths are only near-optimal, because the optimal path is not known in advance and, hence, it is impossible to decide whether searching longer could reveal an even better path.

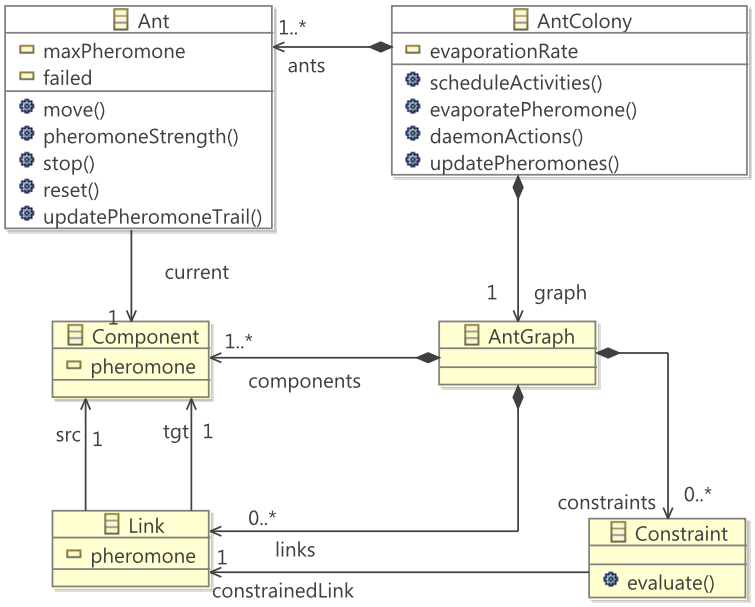


Figure 2.5.: Concepts of an Ant Colony Optimization Framework.

Concepts of Ant Colony Optimization

Figure 2.5 depicts the key concepts of an ACO framework: `AntColony`, `Ant`, `AntGraph`, `Component`, `Link` and `Constraint`.

The central concept is the `AntColony`, which comprises a set of ants and a graph representing the problem to be solved. The task of an `AntColony` is (1) to schedule the ants on the problem graph, (2) to update the pheromone trails after each iteration (i.e., a run of all ants through the graph) and (3) to perform some additional modifications to the graph if the respective problem demands for it. The additional modifications are operations on user-defined (i.e., problem-specific) information in the graph. A typical strategy to update the pheromone trails is the application of evaporation by a specified rate. That is after each iteration the current pheromone levels are decreased by the specified rate.

The `AntGraph` is a direct acyclic graph (DAG) comprised of `Components` (i.e., nodes), `Links` (i.e., edges) and `Constraints`. Each component and each link has a current pheromone level, which is set by the ants or the colony. The `Constraints` allow to put restrictions on `Links`, which are considered by ants during path finding.

An `Ant`'s task is to find a path through the problem graph. In analogy to nature, the ant performs this search step-wise. That is at each component in the graph the

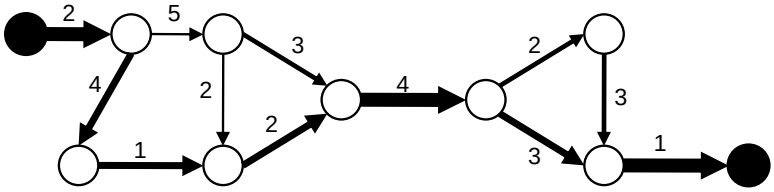


Figure 2.6.: The Shortest Path Problem in Ant Colony Optimization.

ant chooses a link to move to another component considering the constraints. Thus, the individual tasks to be performed by an ant are (1) to decide for and perform a move in the graph possibly using the pheromone trails and/or additional heuristic information, (2) to identify whether the stop criterion for the search is fulfilled, (3) to compute the current pheromone strength to be distributed and (4) to remember problem-specific information while moving through the graph and to reset this memory at each new iteration.

These general concepts need to be instantiated for each concrete problem. In the following such an instantiation is shown for an idealized problem: the shortest path problem. Then, an instantiation for contract negotiation is presented.

Example: Shortest Path Search

The shortest path problem [13] as formulated by Richard Bellmann in 1958 is a well-known problem in graph theory. It denotes the search for the shortest path between a source and a target node in a DAG. The length of the path is characterized by the sum of the weights of all edges used to move from the source to the target node.

To realize this problem in the ACO framework, the graph of the shortest path problem can be directly realized as `AntGraph`. The tasks of the `Ant`s and the `AntColony` need to be specialized. First, the `AntColony` has to set all ants at the beginning of an iteration on the start node. Second, the `Ant`s have to consider a special stop criterion: namely, if they reached the target node. Each ant takes a path through the graph and, if successfully finishes the path[2], marks the found path with pheromone. If the ant fails, the ant can decrease the pheromone on the path taken.

Figure 2.6 depicts a concrete shortest path problem and the pheromone trails created by ACO by several iterations. The numbers at the edges denote the length of the edges. Their thickness depicts the amount of pheromone. The shortest path can easily be extracted from the graph by always following the edge with the highest amount of pheromone.

But, it is important to note that the accuracy of the resulting pheromone distribution depends on several factors. First, the strategy of pheromone distribution by the ants.

[2] as defined by the stop criterion, which can cover a set of "stop" nodes, but also a number of nodes to be visited or even more complex criteria

For example, ants can be realized with a budget of pheromone to be distributed, a constantly decreasing pheromone level or a fixed pheromone level. The final pheromone distribution could be designed to be representable by a value between 0 and 1 or could have no upper limit. The colony can constantly evaporate the pheromone trails after each iteration, which is useful to avoid getting stuck in the search. Such locks occur if many ants randomly choose a suboptimal way. Evaporation helps to reduce this effect. Thus, the quality of the result of ACO strongly depends on the concrete decisions made w.r.t. pheromone distribution.

The application of ACO to contract negotiation is shown in Chapter 8.

2.3.3. Multi-Objective Optimization

The research field of MOO originates from the theory of economic equilibrium, welfare theories, game theory and general mathematics. An overview of state of the art in MOO is given by Marler and Arora in [88]. They provide a definition for MOO as follows:

> "The process of optimizing systematically and simultaneously a collection of objective functions [...]" [88, p. 369]

Thus, in contrast to single-objective optimization, the focus of MOO is on the *simultaneous* optimization of *multiple* objective functions. Comparable to single-objective optimization the optimization problem comprises a feasible design space represented by variables and a feasible criterion space, which represents valid assignments for these variables (i.e., valid solutions). Marler and Arora differentiate between feasibility and attainability in that feasibility implies validity of a solution and attainability only implies the possibility of a variable assignment, which is not necessarily a valid solution. Thus, the general MOO problem comprises a set of objective functions, a set of constraints (equalities and inequalities) and a set of decision variables.

A central issue of MOO is the relationship between the objective functions. These are usually modeled as *preferences* of the decision maker. A commonly used approach is the application of utility functions to express a decision-makers satisfaction. Notably, the utility function is applied to the solutions of the optimization problem and not, as shown in section 2.1, to the objective functions. The challenge of MOO is to derive the solutions without merging the objective functions.

A direct consequence of not merging objectives is the absence of a single optimal solution. The optimality of each solution is assessed for each objective function and, hence, a solution might be optimal in some objective functions, but suboptimal in others. To identify the set of optimal solutions across all objectives a new type of optimality is required: Pareto optimality [101]. A solution is termed (properly) Pareto optimal, if it is not possible to improve the solution in one objective without diminishing another objective. Such a solution is called to *dominate* other solutions. This property (dominance)

is usually utilized to search the solutions of the MOO problem. There are two types of Pareto optimality to be differentiated: weak Pareto optimality, where no other solution is better in all but one objective and proper Pareto optimality, where the increment of one objective is bounded by the decrement of other objectives (trade off). All Pareto optimal solutions for a given problem form a so called *Pareto front*.

Salukvadze proposes in [113] an alternative to Pareto optimality: the compromise solution using an utopian point (i.e., ideal point). The optimization problem changes from searching for dominance between solutions to minimizing the distance of solutions to the utopian point. This alternative approach to optimality in the presence of multiple objectives is only applicable if the utopian point can be estimated, but is meant to reduce the effort required to solve a MOO problem.

Compromise solutions and the utopia point originally have been introduced by Yu [130] and denote the determination of the least group regret for groups whose participants have individual utility functions. Notably, compromise solutions are shown to be Pareto optimal in [130]. Another characteristic of compromise solutions is uniqueness (i.e., there is a single optimal solution). But, the prerequisite is the existence of each participant's utility function. Without such functions the Pareto front results as a set of optimal solutions. To solve a MOO problem using the compromise solutions approach, Yu suggests the division of the problem into $n + 1$ subproblems for n (group) participants—n to identify each participant's maximum utility and another one to identify the compromise solution. Each problem can be solved using linear (or quadratic) programming.

In general, the methods to solve MOO problems divide into (1) a priori articulation of preferences, (2) a posteriori articulation of preferences and (3) methods without preference articulation [88]. A priori methods merge the objective functions before solving the optimization problem. A multitude of approaches to merge the objective functions exist. The most common a priori method is the weighted sum, where all objective functions are weighted and aggregated to form a single objective function. The weights can be interpreted as user preferences reflecting the relative importance of each objective for the user. Another approach is goal programming [82], where goals are defined for each objective function and the deviation of solutions from these goals is minimized. Under special circumstances, an approached coined *bounded objective function* is applicable. Here only one objective function is optimized, while all others are fixed as constraints of the optimization problem.

In this thesis, a priori methods are used for the optimization approaches presented in Chapter 7, because they only allow for a single objective function and a posteriori methods for the approaches presented in Chapters 8 and 9.

The exact single-objective approaches to optimization presented in this thesis use either the *weighted sum* method or the *bounded objective function* method, because both ILP and PBO only allow for a single objective function to be specified.

The key problem of the *weighted sum* method is the incomparability of many NFPs, which renders the approach inapplicable for the general case. Nevertheless, for selected

combinations of NFPs the comparability is given and, hence, the *weighted sum* method is applicable for these special cases. For example, the memory consumption of an algorithm and the central processing unit (CPU) time used by the algorithm can be compared by their effect on energy consumption. Both memory operations and CPU usage can be translated into their implied energy consumption, which in turn is comparable. Thus, for the *weighted sum* method to be applicable a common theme for all NFPs subject to optimization needs to exist.

The *bounded objective function* method is not a MOO method in a strict sense, because only a single objective is optimized, whereas the remaining are fixed as constraints. For incomparable NFPs, which do not have a common theme, this approach enables the application of exact approaches by asking the user to select a single NFP, which shall be optimized.

In contrast, the approximate optimization approach–namely, ACO and meta-heuristics in general– and the approach presented in Chapter 9 allow to specify multiple objective functions and, hence, a posteriori methods to MOO are applicable. That is, a Pareto front [101] of optimal trade offs is presented to the user to select from. By asking the user to rate the importance of each NFP implicitly (by selecting a single solution) the problem of incomparable NFPs is avoided (i.e., postponed to the user).

3
A Taxonomy of Adaptation Reasoning

Various approaches covering adaptation reasoning for self-adaptive systems have been investigated in the last decade. In analogy to the control loop of autonomous systems [48] comprised of the four steps–collect, analyze, decide and act–this chapter summarizes related approaches of the decide step and particularly exposes how MQuAT using contract negotiation advances over this state of the art. First, in section 3.1, a classification of related approaches is given. Then, section 3.2 provides a demarcation and detailed description of each investigated, related research project.

The contributions given in this chapter are the following:

- A new taxonomy of self-optimizing software systems, which exceeds existing taxonomies by the inclusion of runtime optimization issues.

3.1. Facets of Adaptation Reasoning

In [112] a taxonomy for self-adaptive systems in general is given. For a clear and concise demarcation of the approach presented in this thesis, this taxonomy is too broad, because many existing approaches can be categorized as *self-optimizing*. Hence, the following comparative criteria for adaptation reasoning approaches have been identified by an exhaustive literature analysis, to highlight the advances over state of the art in self-optimizing systems. Figure 3.1 depicts the criteria and shows how many related approaches apply for each bottom-most (leaf) category. A detailed discussion for each related approach is provided in the remainder of this chapter.

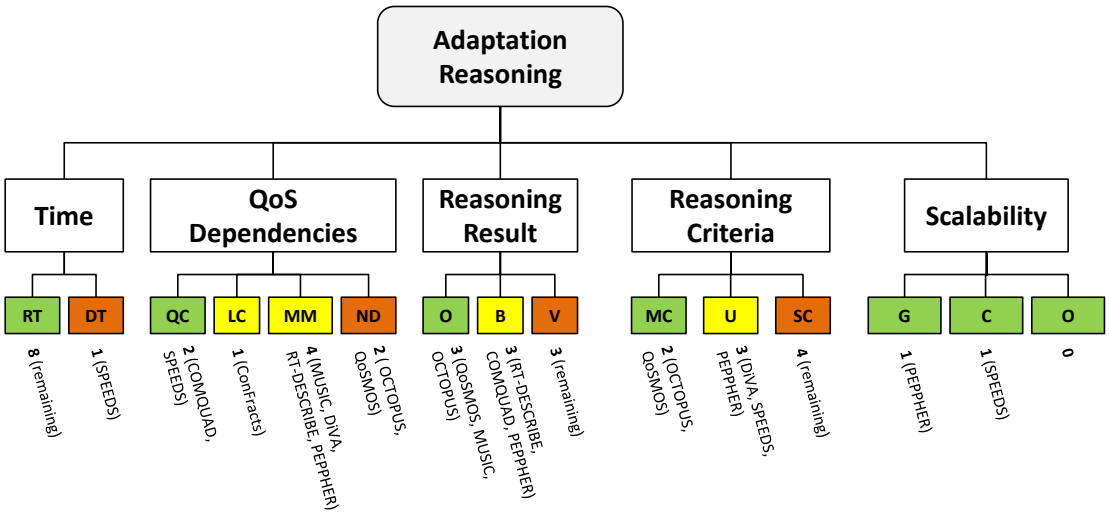


Figure 3.1.: Comparison Criteria for Related Work.

The taxonomy comprises five categories, which have been identified as key demarcating characteristics of the investigated approaches. The literature analysis used to derive this classification included 33 papers originating from nine research projects[1]. In the following each category is discussed in more detail.

Time. The question underlying this class is: *when* is adaptation reasoning performed? Although most of the investigated approaches are naturally runtime approaches (as adaptation reasoning usually relies on runtime information), there are approaches, which allow for design time adaptation reasoning to prepone at least some of the computations required for adaptation reasoning. Thus, this class divides into two categories:

- Runtime (RT) approaches.
- Design-time (DT) approaches.

QoS Dependency Specification. This class addresses the problem of if and how dependencies between QoS attributes of the system subject to self-optimization are covered. The specification (including their automatic detection) of such dependencies provides a deeper understanding of the system (than without this knowledge) and, consequently, allows for finer grained analyses. Most self-optimizing systems are naturally able to reflect on their non-functional behavior, but not every approach allows to evaluate the effect of a

[1]PEPPHER: 6, SPEEDS: 3, MADAM/MUSIC: 4, QOSMOS: 1, DiVA: 3, Octopus: 4, SALTY: 4, COMQUAD: 5, RT-Describe: 3

change in one NFP on other NFPs in addition. This additional step demarcates classical reflective systems from upcoming *predictive* systems, which utilize the knowledge about how the system can change over time to evaluate different action sequences w.r.t. their resulting system configuration. To cover QoS dependencies QoS contracts are suited best, because they explicitly describe these dependencies. In contrast, lower-level contracts—like syntactic, behavioral or synchronization contracts according to [17]—allow to derive QoS dependencies (but require this additional computation step). A further approach is to cover the dependencies in the system model (but not as contracts). This usually requires additional computation to derive the actual dependencies. Thus, the categories of this class are the following:

- Approaches using *QoS contracts* (QC).
- Approaches using *lower-level contracts* (LC).
- No contracts are utilized, but provisions and requirements are part of the system *meta-model* or are realized as a profile (MM).
- Approaches which cover *no dependencies* between system parts at all (ND).

Reasoning Result. The underlying question of this class is the intended result of adaptation reasoning. Depending on the application area of self-adaptive approaches, adaptation reasoning does not always need to derive an optimal system configuration. For self-healing approaches, for example, the resulting system configuration only needs to be valid w.r.t. the system constraints. In other approaches often the complexity of sophisticated adaptation reasoning contradicts with time requirements of the applications. That is the time required for adaptation reasoning exceeds the maximum time feasible for the application to wait for a reconfiguration decision. In such cases *better* solutions, which are not optimal (or near-optimal) are searched, starting from valid solutions. Hence, this class divides into three categories:

- An *optimal* configuration of the system is searched (O).
- A *better* solution, which is not necessarily the best is searched (B).
- A *valid* configuration of the system is searched (V).

Reasoning Criteria. A further demarcating characteristic of adaptation reasoning is how multiple criteria are considered by the respective approach. As pointed out by Salehi and Tahvildari [112], most approaches to self-adaptive systems consider only a single criterion (e.g., performance). More recent approaches, started to support multiple criteria by composing them to a single criterion using utility theory. Each separate criterion is assessed in terms of its utility, which is measured as a value between 0 and 1.

The weighted sum of the utilities results in a single, composed criterion, which is considered by adaptation reasoning. But, this investigation does not reflect the cost required to gain the respective utility. Approaches, able to negotiate this trade-off between cost and utility, consider the efficiency as composed criterion for adaptation reasoning. Finally, multi-objective optimization approaches exist, which investigate multiple criteria without merging them. Nevertheless, to make a decision based on adaptation reasoning, MOO results need to be merged, too. The benefit of MOO is that this merger of criteria is postponed after adaptation reasoning and, thus, the information loss due to merging criteria does not affect the reasoning approach. In summary, this class divides into four categories:

- A set of Pareto optimal configurations for *multiple criteria* will be found (MC).
- A single, but composed, criterion using *utility theory* is considered (U).
- A *single* criterion is considered (SC).

Reasoning Scalability. A central problem of self-adaptive systems is the ability to compute decisions in a given budget (e.g., time or energy). The decision itself can be assessed in terms of its utility. This is because the computation of a decision–which aims to improve certain cost or utility attributes–has a cost, too. To enable *in-budget* decision making, the adaptation reasoning approach needs to be scalable (i.e., it needs to scale with the available budget). Scalable approaches can leverage the granularity of information used for adaptation reasoning, the concerns considered for reasoning and the optimization approach used for reasoning. This class, hence, divides into three subclasses:

- The granularity of information used for optimization is scaled (G).
- The concerns considered by optimization are scaled (C).
- Multiple optimization techniques are utilized (O).

3.2. Related Research Projects

The origin of the investigated papers is given by the research project. In the following, the 9 investigated projects are compared to MQuAT using the taxonomy shown in Fig. 3.1. Table 3.1 depicts this comparison in a concise manner. The details of the demarcation to each project are presented in this section.

	Time	QoS Deps.	Result	Criteria	Scalability
MQuAT (General)	RT	QC	O	MC	G,C,O
PEPPHER (Scientific Computing)	RT	ND	B	SC	G
RT-DESCRIBE (Automotive)	RT	MM	V	U	-
QoSMOS (SOA)	RT	ND	O*	MC*	-
OCTOPUS (Embedded Systems)	RT	ND	O*	MC*	-
SPEEDS (Embedded Systems)	DT	QC	V	U	C
MUSIC (Mobile Devices)	RT	MM	B/O	U	-
DiVA (General)	RT	MM	V	U	-
SALTY (General)	RT	LC	V	U	-
COMQUAD (General)	RT	QC	B	U	-

Table 3.1.: Comparison with Related Work.

In the following, for each of the 9 research projects shown in the previous section a more detailed discussion and demarcation is given.

PEPPHER: Dynamic Auto-Tuning for Many-Core Architectures

The EU-funded (FP7) Performance Portability and Programmability for Heterogeneous Many-core Architectures (PEPPHER) project addresses the challenge of performance portability of *compute-intensive tasks* in heterogeneous multi-core systems. Benkner et al. provide an overview of the PEPPHER project in [15]. Performance portability is one of the key problems motivating auto-tuning (cf. Chapter 2.2). It denotes the problem that code, which has been optimized (w.r.t. performance) for a given platform, does not necessarily provide the same performance properties as other platforms, which have to be supported in addition. The large amount of available platforms strengthens this problem. In consequence, for each platform to be supported, multiple variants of the code need to be developed and optimized, which is a time-consuming and error-prone activity. Moreover, the optimality of the code depends upon runtime information, too. Most notably, the input size has to be considered as shown, for example, in [55]. PEPPHER aims at automating this multi-variant code generation and its selection at runtime. PEPPHER is not an auto-tuning project, but exposes the parameters to be used by auto-tuning techniques (i.e., PEPPHER consideres the runtime selection of variants and the determination of optimal parameters for the variants, but not the generation of implementation variants). The selection of the best variant at runtime considers resource and data availability (i.e., placement) as well as performance information. It is realized as a scheduling problem, which is solved by the PEPPHER runtime environment [8] using heuristics (e.g., Heterogeneous Earliest Finish Time [120]).

The PEPPHER component model provides means to annotate components and their variants with meta-data. Each PEPPHER component defines the interface it implements, deployment information, its required resources, a performance prediction function, call parameter context and resource availability (to be filled at runtime), tunable parameters and additional constraints on selectability. Notably, PEPPHER can generate performance prediction functions based on historical data (benchmarks).

In [114], Sandrieser et al. present details on the application of an explicit platform model to generate code variants optimized for a specific hardware platform. Their approach relies on the Extensible Markup Language (XML) [24]. The platform models are specified using an XML-based platform description language, which supports the definition of processing units, memory regions and interconnects. In addition, control relationships can be defined between processing units to specify the possibility of offloading a task from one processing unit to another. Memory regions and interconnects allow for analysis of the data-transfer implied by offloading tasks. The prototypical framework, called `cascabel`, takes an annotated program and a target platform description as input and generates an optimized output program by replacing annotated calls (i.e.,

functionality marked to be offloaded) by platform-specific code realizing the offloading.

In [40], Dastgeer et al. present a machine learning approach on top of [114] to adapt data-parallel computation tasks implemented using skeleton programming to a given platform. Skeleton programming [37, 108] is a technique, which requires the programmer to implement his code using a predefined set of parameterizable generic components. The adjustment to a given platform is realized by determining the optimal parameterization of these components for the respective platform. The approach allows for the selection of a `good` skeleton instantiation at runtime w.r.t. the actual input size. The applied optimization technique is genetic programming, which iterates over a learned set of candidate parameters, measures their execution time and results in a mapping from input size ranges to optimal parameters for the skeleton. Notably, the approach allows to scale w.r.t. the time required to derive this mapping, by merging input size ranges subject to investigation based on their predication error (resulting in a trade-off between accuracy and time required for optimization).

To evaluate different selection policies for the multi-variant code, a hardware simulator has been developed in PEPPHER. Using this simulator, the effect of the PEPPHER approach can be evaluated on a variety of platforms.

They key demarcating features of MQuAT in comparison to PEPPHER are the lack of QoS dependency specifications and the focus on a single criteria (namely, performance). Moreover, the PEPPHER approach addresses another level of abstraction (code instead of architectural level). The applied optimization technique (genetic programming) allows for *better* configurations and does not necessarily find *optimal* configurations. This is mainly, because not every possible configuration is examined by the approach. Notably, the amount of configurations to be examined is used to scale the optimization leading to a trade off between accuracy of and the time required for optimization.

RT-DESCRIBE: Self-adaptive Embedded Systems

The RT-DESCRIBE project [127, 126, 131] focuses on self-adaptive, distributed embedded devices. It, thus, addresses a comparable adaptation reasoning problem (i.e., how to determine an optimal or valid system configuration). Zeller et al. investigated this problem in the context of automobiles, which are a particular kind of distributed embedded device, due to the vast amounts of interconnected ECUs running software for the automobile.

In [131], Zeller et al. address the problem of determining a valid mapping of software components to ECUs by SAT solving and simulated annealing. They model the automobile as a heterogeneous, distributed real-time system comprised of sensors, functions and actuators by means of a graph $G_f(V_f, e_f)$, where V_f is the set of vertices denoting the sensors, functions and actuators and e_f is a set of edges between these vertices denoting their relation to each other. The data-flow between sensors, functions and actuators is expressed as another graph $G_r(V_r, e_r)$, where V_r denotes the available ECUs and e_r de-

notes the relation between ECUs. They define a system configuration $c(t)$ as a function mapping components to ECUs ($V_f \rightarrow V_r$). Restrictions to the mapping are specified in Φ, a set of pseudo-boolean constraints (variables in 0,1, but coefficients in R). Notably, Zeller et al. do not apply PBO, although their problem model exactly suites PBO, but, in contrast to MQuAT, apply SAT solving and simulated annealing to identify a valid instead of an optimal solution.

Zeller et al. investigated several types of constraints, comparable to those in MQuAT as shown in [60]. First, RT-DESCRIBE offers an assignment set–a manual selection of valid mappings (software component to ECU), which is the result of design time analysis (e.g., airbag function must not be migrated). Second, cardinalities (e.g., each function has to be mapped exactly one time) are considered. Third, resource constraints are taken into account using the worst case requirement of a function. In contrast, QCL used in MQuAT allows to specify multiple different quality modes with different resource requirements. Furthermore, real-time scheduling constraints (like earliest deadline first) using worst case execution time (WCET) techniques are supported. Finally, network constraints and timing constraints for end-to-end timing chains are supported. These chains describe a path through the system, starting at a sensor, going through several functions realized by software components on ECUs and ending at actuators. For each function the WCET and for each connection the worst case transmission delay of a message is considered.

The solving techniques (SAT solving and simulated annealing) do not guarantee optimality, but ensure the determination of a valid system configuration. Zeller et al. evaluated their approach and showed that the runtime of their SAT solving approach is below 4s for 40 ECUs, 60 Sensors, 60 Actuators and 2000 Functions. For less than 1600 functions SAT solving takes less than 2 seconds. The biggest setup evaluated by Zeller et al. comprised 100 ECUs, 120 Sensors, 120 Actuators, 2500 Functions and took 18 seconds [131]. Notably, current automotive setups are much smaller than the evaluated setups, but could become that large in the future.

In contrast to MQuAT, RT-DESCRIBE does not search for optimal configurations, the constraints of the optimization problem are not generated, the notion of contracts is not utilized and only the mapping, but not the selection problem is considered. Finally, RT-DESCRIBE is specialized for the automotive domain, although most of the constraints are of general nature.

QoSMOS: A Formal Approach to Self-adaptive Service-based Systems

In [30], Calinescu et al. present QoSMOS[2]–a generic architecture for adaptive service-based systems (SBS). The central constituents of QoSMOS are formal specifications of QoS requirements (using probabilistic temporal logics) including the specification of

[2]http://www.ict-qosmos.eu/

dependencies between QoS requirements, model-based QoS evaluation using verification techniques, learning monitoring of QoS properties and reasoning techniques, based on high-level, user-specified goals and multi-objective utility functions.

QoSMOS focuses specifically on SBS, which are comprised of a set of web services under the control of a workflow engine. A workflow, to be executed by the workflow engine, is specified using abstract web services, for which possibly multiple concrete web services exist. This is comparable to the notion of component types and implementations in the CCM. In addition, NFPs are defined per component type (i.e., abstract web service), but, in contrast to MQuAT, contracts are not utilized to specify dependencies between NFPs. Instead, formal, temporal, probabilistic formulas are used to specify the system's behavior and to detect dependencies between NFPs using formal verification techniques.

The optimization problem of QoSMOS is comparable to MQuAT as it consists of a selection problem and a resource allocation problem. The selection problem covers the choice of an optimal concrete service for an abstract service and the resource allocation problem addresses the question, how to distribute the budgets of available resources to the selected services. QoSMOS specifically focuses on how to adjust the allocated resources (e.g., determining the optimal CPU frequency) and supports the computation of an optimal parameterization of components.

QoSMOS advances over MQuAT by its formal specification of QoS requirements by probabilistic temporal logics. The drawback of this formal approach is implied complexity for developers, which have to provide the specifications. For that purpose, the ProProST (probabilistic property specification templates) approach has been developed, which enables the developer to provide the required QoS specification using structured English instead of probabilistic temporal logic.

Like MQuAT, QoSMOS supports multi-objective optimization. But, in contrast to MQuAT, the optimization aims for maximum utility, where different dimensions of utility represent the different objectives. The QoSMOS approach to multi-objective optimization is realized by single-objective optimization by combining the different objectives as a weighted sum. Thus, in contrast to MQuAT, the trade off between cost and utility is not considered and the Pareto front of the multi-objective optimization problem is not investigated. Furthermore, QoSMOS focuses explicitly on service-based systems and dependencies between NFPs of different services cannot be expressed explicitly.

OCTOPUS: The MO2 Style

The OCTOPUS project[3] aims at runtime adaptability of embedded systems. In particular, advances in model-based reasoning for system level control have been envisioned. Within OCTOPUS, Arjan de Roo investigated the adaptation reasoning approach, which

[3] http://www.esi.nl/octopus/

resulted in an architectural style for systems capable of multi-objective optimization.

In [43], de Roo et al. introduce this architectural style for software systems (the MO2 style) as an extension to the component and connector style [36]. According to the MO2 style, the basic elements constituting software are adaptable components (AC), multi-objective optimization components (MOO-C) and transformation components (TC). These three types of components are connected by relations with each other.

The purpose of MOO-Components is to realize a solver for the MOO problem. Therefore, it requires the decision variables, objectives and constraints as input. These are provided exclusively by the ACs. In order to adjust the decision variables delivered by ACs, TCs can be used to perform mathematical operations on these decision variables.

In comparison to the expressiveness of CCM and QCL, the MO2 style has three major drawbacks: (1) global optimization is hard to express, (2) no means to express contextual dependencies exist and (3) no distinction between software and hardware exists. Each of the three issues can be addressed in the MO2 style, but each solution introduces further problems.

The first issue is because of the lack of means to express cross-component dependencies (e.g., a video player, which requires a decoder to provide a certain data rate in order to provide a certain frame rate in turn). Using transformation components, the data rate can be combined with the frame rate, so a MOO component is able to optimize both. But, because objective functions can only be provided by ACs, it cannot be expressed how the data rate of the decoder influences the frame rate of the video player. This requires an objective function provided by the MOO component itself. Thus, the MO2 style cannot be used to express optimizations of composite systems, where properties of parts of the system depend on each other in a non-trivial way. In MQuAT, such dependencies are expressed using QCL.

Furthermore, contextual dependencies between decision variables cannot be expressed in the MO2 style. For example, a navigation system could have a decision variable for the selection of a certain network device for communication. Only if WLAN is available, large map updates shall be downloaded. Such a contextual dependency cannot be easily expressed in the MO2 style. To address the problem, the decision variable for the choice of downloading the map updates or not, needs to be split into a separate variable for each (context) situation. In this small example, no problem arises, but for larger systems a combinatorial explosion of decision variables results, which significantly affects the scalability of the MO2 style. In contrast, MQuAT allows to express contextual dependencies using QCL, too.

Finally, the MO2 style does not distinguish between software and hardware. This prohibits the application of the style to distributed systems, with multiple devices hosting software. Although it is possible to store the complete hardware infrastructure in the ACs, this leads to high redundancy and again a combinatorial explosion of decision variables. In comparison, MQuAT distinguishes between hardware and software in its CCM and allows to explicitly describe the dependencies between hardware and software

as well as software and software components using QCL.

Nevertheless, for software components to be deployed on a single embedded device, without contextual dependencies and without the need of global optimization, the MO2 style provides a good framework. MQuAT, in contrast, aims for multi-objective optimization in highly adaptive distributed hardware/software systems.

SPEEDS: Design-Time Contracts

The SPEEDS[4] project focused on a design-time approach for embedded systems development based on component-based design. Besides development, the approach covers testing, integration and certification. Of particular interest w.r.t. adaptation reasoning is the contract language contract specification language (CSL) developed in SPEEDS, which is organized in viewpoints to realize separation of concerns [47]. The CSL allows for the specification of assumptions and guarantees of the system's components w.r.t. the respective viewpoint (e.g., real-time and safety).

In [107], Quinton et al. present the SPEEDS contract framework for hierarchical component-based systems. They show how to formally compose contracts and introduce the concept of dominance between contracts, where "dominance means for contracts what refinement means for components" [107, p.8]. The contract framework is divided into two levels: L0 and L1. The L0 level is for low-level semantic composition of contracts, whereas the L1 level provides more complex composition operators and enables circular reasoning. The application area of the contract framework is verification support for system development at design-time.

According to [16], in SPEEDS, a contract is a pair of assertions on behavior descriptions expressing assumptions and promises. A behavior is, e.g., defined by means of a labeled transition system. The assumption part of contracts is checked against the behaviors to deduce if the promise part is true or not. If the promise part is true, it is a guarantee.

The heterogeneous rich components (HRC) meta-model, like most meta-models for component-based systems, provides concepts for hierarchical components and ports. In addition, contracts and viewpoints are covered by HRC. The behavior of an HRC component is defined by means of state machines. In [12], Baumgart et al. describe the extensions applied to the HRC meta-model of SPEEDS to the common system meta-model (CSM) of the consecutive project CESAR. The CSM meta-model extends HRC by a requirements level.

In contrast to MQuAT, both SPEEDS and CESAR investigated a design-time approach. They utilized a notion of contracts similar to QCL in that contracts comprise assumptions and guarantees (i.e., requirements and provisions w.r.t. other components). The main focus of SPEEDS and CESAR was not on optimization using contracts, but

[4] http://www.speeds.eu.com/

on design-time verification. Thus, a *valid* configuration of the system is intended by the approach. The concept of viewpoints allows to specify multiple criteria, which are composed according to utility theory. Thus, costs and the trade-off between costs and utility as well as MOO are not considered. The scalability issue does not apply to SPEEDS, as at design time there usually is no limited budget for adaptation reasoning.

MADAM/MUSIC: Maximizing User Satisfaction

The consecutive research projects MADAM[5] and MUSIC[6] investigated self-adaptive systems with special focus on mobile, ubiquitous applications. First, in MADAM, a development methodology for adaptive mobile applications has been developed. Both, MADAM and MUSIC, rely on the paradigm of CBSD.

According to MADAM/MUSIC [5, 23], the key constituents of software are component types and plans, which specify realizations of these types. Component types are either atomic or composite. A plan for an atomic component type denotes an implementation of that type. Plans for composite types specify a set of comprised required component types and their interconnection, whereof each component type again is either atomic or composite. Thus, the proposed meta-model for mobile, self-adaptive systems is hierarchical and component-based.

Each plan further describes the QoS properties of the realization, which can be expressed as functions of other properties, the required resources and a utility function, denoting the utility of this particular plan for the user in comparison to alternative plans.

The reasoning approaches developed in MADAM/MUSIC realize single-criteria optimization with a maximum utility objective. In MADAM the distribution of component in containers or devices has not been considered. The proposed optimization algorithms of MADAM are greedy search, brute force search and shortest path search [5].

The brute force search algorithm is a naïve approach, which enumerates *all* system variants, assesses them using the utility functions, and selects the variant with the highest total utility. By assessing all system variants it ensures to find the optimal one. This approach is infeasible, because of the combinatorial explosion of variants, which are subject to this assessment.

In contrast, the greedy search algorithm avoids to iterate all variants, but does not necessarily find the optimal variant in turn. Depending on the start position of the algorithm (i.e., the first type for which a plan is to be selected), the algorithm iteratively searches for better solutions. Once the algorithm finds a solution, which is worse than the preceding one, it aborts. In consequence, a "better" solution is identified, but not necessarily an (near-)optimal one. Notably, both brute force and greedy are NP-hard in their complexity, whereas the third proposed algorithm, the shortest path search, has polynomial complexity.

[5] http://cordis.europa.eu/fetch?CALLER=PROJ_ICT&ACTION=D&CAT=PROJ&RCN=71917
[6] http://ist-music.berlios.de/

To apply the shortest path search algorithm to the problem of identifying the optimal system variant in terms of maximum user utility, the plans and their utilities are transformed to an acyclic graph, which is used as input to the algorithm. Notably, such a transformation only works, if the distribution of components onto several devices is omitted, because these would introduce cycles into the graph. For example, if two components are to be mapped on two devices, the graph would contain four nodes (two for the components and two for the devices). The graph needs to express all possible variants of the system by means of paths through the graph. In consequence, there have to be edges between the components and the devices and edges from the devices to the components, which inevitable introduces cycles to the graph, which prohibits the application of a polynomial time shortest path search algorithm.

Although in MUSIC the distribution of components to multiple devices is considered, the proposed reasoning approach is only the naïve brute force [23] algorithm. Nevertheless, in addition to the reasoning approaches of MADAM, user preferences are considered. This denotes a weak type of multi-objective optimization, because multiple objectives are combined using these preferences. But, because the objectives are all the same (i.e., maximum utility), there is no need to apply techniques for multi-objective optimization.

In addition to MUSIC, Mohammad Kahn [71, 72] developed a further reasoning approach as an extension to the MUSIC approach. In his PhD thesis, he focus on unanticipated adaptation in the context of self-adaptive, mobile applications. He realized unanticipated adaptation by extending MUSIC, so the variability model of an application is created at runtime and, in consequence, the reasoning is able to work on the current state of the system, which includes unanticipated components, which have been added at runtime. Kahn identifies the scalability problem of adaptation reasoning, which denotes the combinatorial explosion of system variants to be evaluated. He extends the MUSIC approach by supporting the composition of utilities, but also aims for utility maximization and does not consider the trade off between cost and utility. Nevertheless, he developed a reasoning approach, which is significantly better than prior approaches w.r.t. computational complexity. Namely, the developed reasoning approach has a linear complexity w.r.t. the number of plans (i.e., realizations of component types)[7]. First, for every plan the utility is computed using the information from the runtime system. These plans are filtered, so only those plans remain, which satisfy the user. Then, for each resource the identified solutions are validated and the best solution, which fulfills the resource requirements and has the least utility-sacrifice is chosen using a greedy search algorithm (i.e., a "better" solution is found, but not necessarily the best). Finally, architectural constraints are evaluated in the same way.

Kahn admits that he does not necessarily find the optimal solution and enumerates four assumptions, which have to be met, for the algorithm to provide an optimal solution. These are: (1) the utility of a plan can be derived from it's QoS properties, (2) a

[7] But is still NP-hard w.r.t. the number of variants to be assessed.

composed utility depends on its constituting utilities and possibly external factors, (3) part utilities influence the composed utility only in a positive way and (4) utilities of different plans for the same component type need to be comparable. These assumptions concisely outline the limitation of Kahn's approach.

A central difference between Kahn's approach and MQuAT is the specification of dependencies between NFPs. The contracts used in MQuAT allow to explicitly define how NFPs relate to each other. Using Kahn's approach, only composite NFPs can be expressed and, according to his third assumption, partial NFPs need to have a positive influence on the composite NFP.

Thus, MQuAT advances over MUSIC and Kahn's approach by utilizing contracts to enable the specification of dependencies between NFPs and by its reasoning approaches (ILP, PBO, ACO), whereof ACO allows for multi-objective optimization and all three approaches optimize for efficiency (i.e., consider the trade off between cost and utility), whereas in MUSIC and Kahn's approach only user satisfaction is considered.

DiVA: Quality Grade Checking

In the DIVA[8] research project an architecture for dynamic adaptive systems has been investigated. In [57], Fleurey et al. show the DIVA approach to identify target system configurations. Notably, in the DIVA approach the components do not declare their provided and required NFPs. Instead a sophisticated *quality grading model* is provided, which allows for the specification of quality dimensions (e.g., energy, time and accuracy) and an assessment of code in terms of a finite set of grades. It furthermore allows the specification of priorities between quality dimensions.

According to the DIVA quality grading model, first the context is to be specified in terms of a set of variables. For example, a boolean variable `lowBattery`, which expresses, whether a device is currently on low battery or not. Next, the quality dimensions are defined. These denote aspects of the software system, for which different variants exist. For example, a dimension `Networking`, which provides the variants `bluetooth` and `GPRS`. For each variant dependencies to other variants (of other dimensions) can be defined. In addition, the applicability of a variant can be restricted by propositions on context variables (availability clauses) or a variant can be enforced for a given assignment of context variables (required clauses). For example, the `bluetooth` variant should just be considered, if a bluetooth signal is available (expressed by the context variable `BTSig`).

For the variability model, several properties can be defined, which denote concerns of the application. These can be `power consumption`, but also application specific concerns like `map detail` for a mapping robot. All variants are assessed by the developer in terms of these concerns using a finite set of (quality) grades, which express the impact of a variant to the respective concern. Finally, these concerns can be prioritized using

[8] http://www.ict-diva.eu/

propositions on context variables. For example, the `power consumption` concern can be defined to be of high priority, if the battery of the robot is low (expressed by the context variable `LowBatt`).

Although software-hardware dependencies can be expressed in a coarse-grain level using the quality grades, DIVA abstracts from concrete NFP values to symbolic categories (e.g., the grades *low* and *high* for the *CPU usage* dimension). An approach to estimate the concrete hardware utilization of specific software components like we showed in [62] is not part of the DIVA approach.

In contrast to MQuAT, the determination of the target system configuration for the adjustment to context changes, the DIVA approach searches for any valid configuration, but does not search for the optimal one. The applied techniques are SAT solving, binary decision diagrams and a brute force search algorithm. The reasoning techniques have been implemented in Alloy [68], for which an automatic transformation from the DIVA variability model has been developed using Kermata [92]. Notably, DIVA does support the consideration of multiple NFPs, but on a very coarse-grain level. For example, an implementation can be graded 5 in terms of energy consumption and 3 in terms of responsiveness. The two dimensions (energy consumption and responsiveness) are prioritized and result in a total quality grade of the implementation, which, in turn, includes multiple qualities.

SALTY: Contract Negotiation for Consistency

A further component-based approach to self-adaptive systems is the *ConFract* approach from the *SALTY* project, first presented in [32]. The *ConFract* approach particularly focuses on self-healing systems. In contrast to MQuAT, the system does not aim for an optimal configuration in terms of efficiency, but to recover from invalid configurations. Notably, the collect, analyze and act phases are not affected by this difference. Only the decide phase demands for another (simpler) type of reasoner.

In ConFract a special approach for the collect phase has been developed [52]. Le Duc et al. present an adaptive monitoring framework called *ADAMO* to be used in *ConFract* for resource usage profilers. *ADAMO* is a profiling infrastructure, which is self-adaptive in terms of the need-to-know principle. Depending on the quality of information required by the users of the software system, the profiling infrastructure adjusts itself to collect exactly what is needed.

In *ConFract*, the knowledge about the system is represented by a component model called *Fractal* [26] extended by a contract language, which is comparable to the Object Constraint Language (OCL) [125]. These functional contracts are used to specify how a valid system is characterized and to initiate self-healing in case of contract violations. Using ADAMO, the developer is able to explicitly specify resource usage profilers as part of the system. In consequence, the functional contracts, which use data generated by these profilers, can be used as non-functional contracts, so dependencies between

NFPs can be expressed. Notably, the *ConFract* contract language is not a QoS contract language according to [17], but a behavioral contract language.

In [33], Chang et al. describe their approach to contract negotiation for self-healing systems. In their approach the system continuously checks the contracts for their validity. If a contract is not valid (i.e., violated), the contract negotiation process is started to determine reconfiguration actions for the system to recover. For each contract provision an *atomic negotiation* is initiated, which follows a negotiation protocol between the negotiation parties. This approach to contract negotiation, hence, is close to negotiation in multi-agent systems and comprises announcing, bidding and rewarding steps (i.e., is comparable to an auction). The matter of negotiation is the invalid provision.

Each negotiation involves three parties: the contract controller, which initiates the negotiation, the participants of the provision (i.e., the components referenced in the provision) and an external contributor, which contributes deeper decision and analysis capabilities (e.g., a system administrator). The process of negotiation is managed by the contract controller continuously asking for proposals (system modifications) on how to restore the validity of the contract and validating the effectiveness of the proposals until a working modification is found.

Two negotiation policies for ConFract are presented in [38]: *concession-based* and *effort-based negotiation*. The first policy bases on concessions made by the negotiation parties. The proposed modifications are mitigations of the provision including its withdrawal. An example is the progressive reduction of a video's display size in accordance to the energy budget of the device used for playback and the abortion of playback if the device falls short of a certain minimum energy budget. According to the effort-based policy, the negotiation parties do not propose mitigations of the provision, but efforts to restore the provision. These efforts are actions affecting the quality of the implementation(s). Both negotiation policies execute until a proposal which restores the provision is found and fail otherwise.

Notably, *ConFract* in combination with *ADAMO* does not consider the second phase of the MQuAT contract creation process (deployment time), where functions mapping user input to resource usage are automatically derived. Instead, the developer has to specify this information on his own in the form of a monitoring query against ADAMO. Thus, in contrast to MQuAT, the determination of how the user input affects certain non-functional properties has to be done by the developer. In addition, the contract negotiation process is realized as an auction, including announcing, bidding and rewarding steps, whereas in MQuAT the negotiation process is the computation of an optimal solution for a system of equations.

COMQUAD: Approximate Contract Negotiation for Real-Time

In the collaborative research center 428 named COMQUAD[9] a software development methodology for component-based software systems with non-functional properties has been investigated with special regard to the assurance of the NFPs and adaptation mechanisms of component-based software systems. In this project Mesfin Mulugeta wrote his PhD thesis on contract negotiation [93].

In COMQUAD a standard component architecture has been used, which was enriched by quality contracts denoted in the component quality contract language (CQML) [1], whereof the extension CQML+ has been developed by Röttger and Zschaler [111] to support the expression of resource dependencies. The quality contract language (QCL) shown in this thesis has been significantly influenced by CQML+ and is a fundamental revision of it.

Based on CQML+, Mulugeta developed a two phase approach to contract negotiation [95]: a coarse-grain phase and a fine-grain phase. During the coarse-grain phase the provided and required *coarse* NFPs (i.e., those comprising lower-level NFPs) are compared to each other. If multiple matching contained NFPs are identified, then fine-grained contract negotiation is performed. In this second phase, Mulugeta proposes a branch and bound algorithm working on the formalization of contract negotiation as a constraint solving problem (CSP), which is meant to determine an optimal system configuration. The variables of the CSP represent the selection of QoS Profiles (i.e., contracts in QCL). The constraints comprise user requirements on specified NFPs, conformance constraints ensuring the matching of provided and required NFPs and resource constraints, which ensure that resource allocation does not exceed the available resource budgets.

The objective function of the CSP is utility maximization, where the utility function is the sum of weighted QoS provisions. In consequence, his approach to contract negotiation only checks if enough resources are available, but does not consider the tradeoff between resource usage and user utility. The fine-grained contract negotiation algorithm iteratively assigns a profile to each variable and checks if the utility requirement is still fulfilled. Notably, the variables are ordered in accordance to the user preferences. Thus, the solution of the algorithm will be the last assignment which fulfilled the user's demand. In consequence, the approach determines a "better" solution [93], but not necessarily the optimal solution.

In [94], Mulugeta and Schill discussed contract negotiation in the presence of multiple clients. They identified three situations: (1) system with light-load: all users can be satisfied, because enough resources are available, (2) over-load: not all users can be satisfied (i.e., some users have to wait) and (3) the request rate of the users is known, which enables the detection of the first two situations. The general question asked in [94] is, whether re-negotiation at runtime is necessary, what is shown for the over-load sce-

[9] http://st.inf.tu-dresden.de/comquad/

nario. Finally, Mulugeta differentiates between centralized and distributed negotiation. For centralized negotiation only one container realizes the negotiation, whereas in distributed negotiation all containers negotiate their own components, and communicate solely to handle cross-container dependencies. A hybrid approach is central negotiation to detect bottleneck resources followed by distributed (local) negotiation, to reduce the communication efforts.

In contrast to the contract negotiation approach presented in this thesis, Mulugeta's approach does neither support the determination of the optimal solution nor the optimization of multiple objectives.

3.3. Summary

In this chapter an overview on state of the art in self-optimizing software systems has been presented. A novel taxonomy highlighting central challenges in this area of research has been introduced and was used to classify 9 research projects. In consequence, two not yet addressed challenges can be identified:

First, the consideration of multiple NFPs in parallel. Although MOO approaches have been developed, these approaches do not consider dependencies between NFPs. In consequence, the result of these approaches (i.e., the configurations identified as being optimal) is inaccurate. In this thesis, each optimization approach leverages from NFP dependency specifications as part of the development method. Moreover, four optimization approaches are presented, whereof three are a priori MOO approaches and one is an a posteriori MOO approach.

Second, the scalability of the applied optimization approaches has to be analyzed to be considered at runtime. Only few projects provide such analyzes. Notably, two approaches to vary the time and resources required for optimization have been developed: the SPEEDS project investigated concern-based problem abstraction and PEPPHER allows to vary the problem granularity. In this thesis concern-based problem abstraction is supported in a similar way to the SPEEDS project: each NFP denotes a separate concern and can be separately omitted from optimization. Granularity-based problem abstraction is supported by enabling the software developer to model the system under development using a hierarchical component model. Finally, a scalability analysis is provided for each optimization technique part of this thesis.

Part II.

A Development Methodology for Self-Optimizing Software Systems

4
A Multi-Quality-aware Software Architecture

The construction of software systems can be compared to the construction of a building. Software is made from basic elements, which are connected to each other in a specific way. These two constituents denote an architecture: building blocks and connectors. For software various kinds of building blocks and connectors have been investigated. CBSD is a meta-architecture, in that it further restricts the building blocks. According to Szyperski et al. [117] software should be constructed from components, elements of reuse, which explicitly define what they require and provide in turn.

A clear benefit of CBSD is the split of the program into multiple reusable and, for self-adaptive software most importantly, exchangeable components. In consequence, many approaches to self-adaptive software base on CBSD.

The problem addressed in this chapter is the inability of current software architectures for self-optimizing software systems to support scalable reasoning and to capture the relationships between the system's NFPs and the user's utility.

The contributions given in this chapter are the following:

- A self- and context-aware, component-based software architecture for self-optimizing software systems. (Section 4.1)
- An extended contract concept (and language) covering the complex interdependencies between NFPs, cost and utility. (Section 4.2)

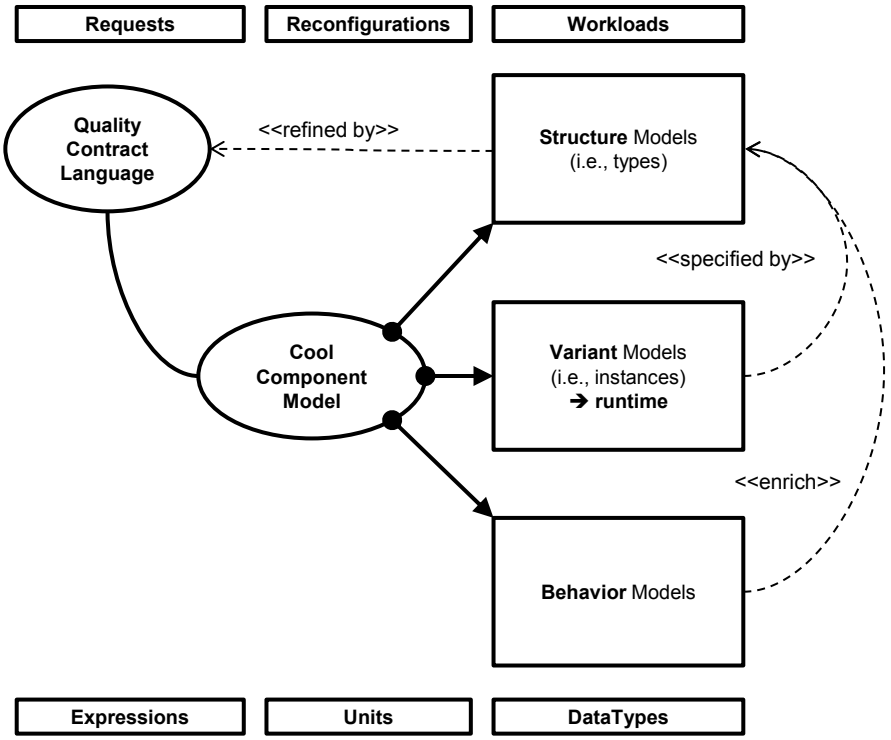


Figure 4.1.: Constituents of the Multi-Quality-aware Software Architecture.

The key requirements of an architecture for self-optimizing software systems are:

/R1/ Types. To enable the optimization of software, the users, hardware and contextual elements need to be covered. In other words, the requirement is self- and context-awareness, as shown by Salehi and Tahvildari in [112] (cf. Figure 2.2).

/R2/ Behavior. An architecture for self-optimizing software needs to support abstract, quality-specific behavior descriptions to enable the prediction of the effects implied by action alternatives. Without such knowledge, the system is unaware of the potential gains in efficiency it is aiming for.

/R3/ Models@Runtime (M@RT). The concept of models at runtime [91], which cover the runtime state of the system, are yet another important prerequisite for self-optimizing software. The application of modeling techniques tailors the information available from the system, so reasoning can be performed in feasible time (i.e., the system remains tractable).

/R4/ NFPs. As the subject of optimization is the non-functional behavior, the NFPs of the system to be optimized need to be specified explicitly.

/R5/ Dependencies. Finally, to improve the quality of optimization, an explicit specification of dependencies between non-functional properties is beneficial [117].

The main constituents of the meta-architecture are a component model and a contract language as depicted in Figure 4.1. The requirements /R1/, /R2/ and /R3/ are each covered by a separate package of the component model. The contract language realizes requirement /R5/. Requirement /R4/ cross-cuts the former, because NFPs are defined for the types (/R1/), have to be considered in behavior abstractions (/R2/) and at runtime (/R3/) and are, obviously, part of dependency specifications between NFPs (/R5/). Thus, the component model is used to specify the structure (i.e., structural models) and behavior (i.e., behavior models) of the system. In addition, the concepts for variant models are defined by the component model, which are meant to cover the runtime state of the system. The contract language is used to specify the non-functional behavior of the system and dependencies between components of the system. This enables an explicit description of valid system variants, which differ in their non-functional, but not in their functional behavior. The runtime environment (cf. Part III of this thesis) leverages the information comprised by runtime models to manage the self-optimizing system.

In addition, concepts like expressions, units (i.e., measurement units like, e.g., megabyte) and datatypes as well as special purpose concepts like user requests, reconfigurations and workloads are part of the component model. In the following, first the component model followed by the contract language will be discussed in more detail.

4.1. The Cool Component Model

The CCM is a balanced set of interrelated concepts, which are meant to be used by software developers to specify the system under construction. It extends the basic notion of CBSD by elaborated concepts tailored for the construction of self-optimizing software systems. It consists of a set of metamodel packages. In the following, the three packages of the CCM (cf. Figure 4.1)—structural models, variant models and behavior models—will be explained in more detail.

4.1.1. Structural Models: System Architecture Description

This metamodel package, depicted in Figure 4.2, covers concepts to specify `ComponentTypes` of the system under construction and, thus, addresses requirement /R1/. A component type denotes an exchangeable part of the system, which adheres to Szyperski's definition of components [117]. A key characteristic of structural models is the distinction of

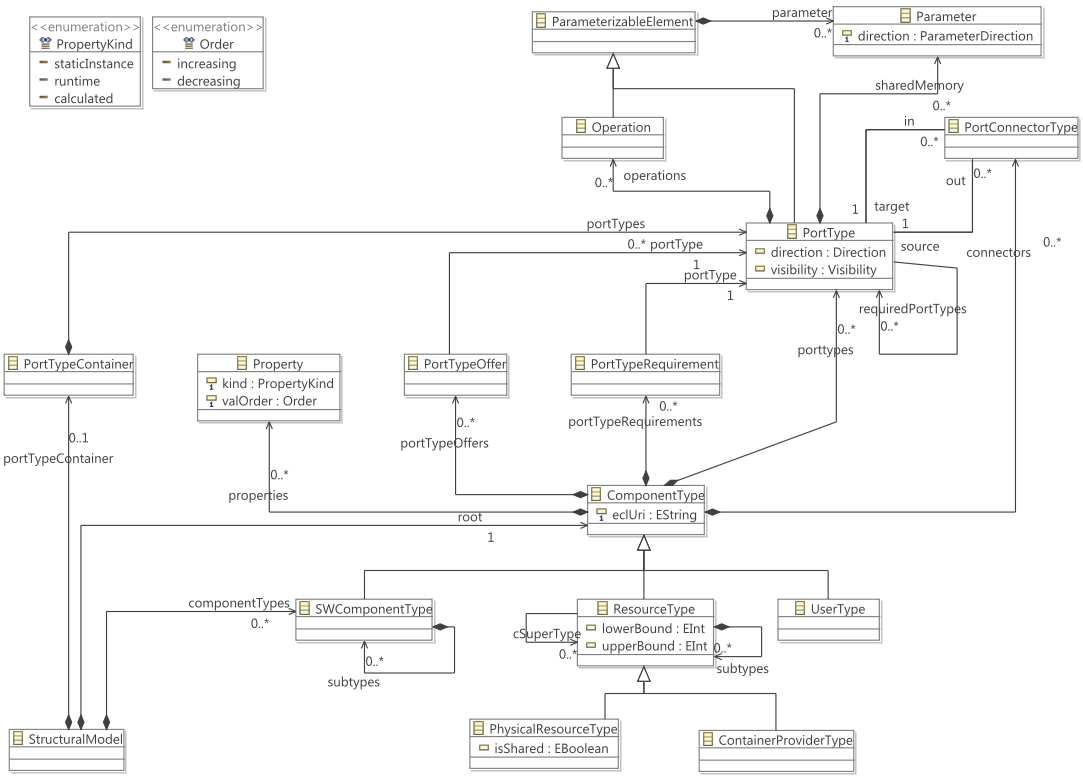


Figure 4.2.: Concepts and their Relations in the Structure Package of the CCM.

component types in `UserType`s, software types (`SWComponentType`) and `ResourceType`s. Whereas software and resource types are hierarchical components, user types are not. That is software and resource types are allowed to contain subtypes, which allows for structured system design following the wide-spread divide-and-conquer principle. The distinction between these three kinds of component types is important, because each kind (users, software and resources) as well as their interrelationships differ in their characteristics. A software component *runs on* a resource and a user *interacts* with a software component (as will be shown in the next subsection). Without a distinction of users, software and resources, this information is hard to express. The interaction and deployment information is a central part of the optimization problem and, hence, needs to be covered.

An example for a structural model is shown in the next section, in Figure 4.4, combined with a variant model to illustrate the close relation of these models.

To explicitly describe functional requirements and provisions, a component type comprises `PortType`s, which are classified as either provided or required ports (by their

`Direction`). Port types are comparable to methods in general purpose programming languages. That is, they are characterized by a name, a set of parameters and a return type. In terms of object-oriented programming languages, port types are interfaces offering a set of `Operation`s. Provided port types need to be implemented by a component realizing a component type. In case a provided port is called, the respective implementation of the realizing component will be used. Required port types are meant to be used by the component instead of directly referring to another component type. The determination, which method will actually be used in case a port is called, is performed at runtime. This process demands for another concept: `PortConnectorType`s, which are used to define the control flow between a required and a provided port type. Thus, at runtime, whenever a required port type is called by a realizing component, the call is dispatched to the provided port type defined by a port connector type of the invoked required port type. This basic dispatch mechanism can be refined by sophisticated port connection strategies, which compute port connector types at runtime. This decouples the system architecture specification from dispatch mechanisms (i.e., no concepts have to be introduced into the meta-model, which are specific for a certain kind of dispatch mechanism). The CCM allows for the specification of hardware and user types, too. In consequence, the concept of port connector types can be used to relate hardware, software, and users to each other.

To support open systems, the CCM allows to specify `PortTypeOffer`s and `PortType-Requirement`s. The principle idea is that port types are not specified for a specific component type, but as separate functional artifacts, which can be added to the system at runtime. To relate to existing functionality, port types can refer themselves to required port types. Whenever a new port type is added to the running system, they are connected to the existing component types by evaluating the component types' port type requirements.

Finally, to realize requirement /R4/, for each component type a set of NFPs can be defined (`Property`). Typical properties of software component types are `response_time` or `resolution`. For resource types, `cpu_time` or `free_memory` are examples. Properties of user types are not in the scope of this thesis. Each property is measured according to a specified `Unit` (e.g., seconds) and can be further characterized by the way its values are derived (`PropertyKind`) and the order of these values (`Order`). Property values can be fixed for a concrete component instance (e.g., the maximum frequency of a CPU is fixed), which is denoted as `staticInstance`. But, many values are only available at runtime and have to be measured (e.g., the amount of free memory). Such properties are `runtime` properties. Finally, there are `calculated` properties, for which the values can be derived (i.e., calculated) from other properties. Whereas the property kind characterizes how the property's value can be determined, the order denotes how the values of a property relate to each other. For `increasing` properties, higher values are better than lower values. For example, it is better to have a higher framerate of video than a lower. In consequence, for `decreasing` properties, lower values are better than higher ones. Many

time-related properties (e.g., response time) fall into this category.

The attentive reader might have noted that not all concepts explained above are contained in Figure 4.2. This is because some concepts are extracted into separate packages, which are shared by all packages of the CCM and QCL. The extracted concepts belong to the following categories: *expressions*, *datatypes*, *units*. In addition, special purpose packages exist for *requests*, *reconfigurations* and *workloads*. An overview of the most important concepts from each of these categories will be shown at the end of this section.

4.1.2. Variant Models: Abstract Runtime System Representation

The purpose of variant models is to give an abstract view onto the current state of the running system. This basically realizes requirement /R3/, i.e., the M@RT concept: the application of modeling, abstracting a given domain for a given purpose, to enable reasoning about the system. Without abstraction such reasoning is nearly impossible, because the amount of information is usually too high to be processed in a feasible time and often the complexity of the information leads to undecidability of reasoning [6]. The variant models of the CCM are such runtime models. They are constructed at runtime by the runtime environment (cf. Part III of this thesis).

Figure 4.3 depicts the concepts for variant models. The structuring concept is the component, whereof one denotes the *root* component. Depending on the purpose of the variant model, either a resource, software component or user is the root. In each case, the resulting structure is a tree, as all concepts allow for nesting. User variant models form a collection of users arranged in hierarchical groups. The resource concept is further specialized into `PhysicalResource`s and `ContainerProvider`s. A physical resource (e.g., a CPU or a core of a CPU) is used by a container provider and vice versa. Container providers (e.g., servers or mobile phones) allow for the deployment of software components. The type of the root component denotes the type of the whole variant model (resource, software or user). In consequence, to express the deployment of software onto container providers requires at least two models: one for the *resource infrastructure* and another one for the current software deployment, which refers to the resource infrastructure model.

Every kind of component has `Port`s and `PortConnector`s, which denote the concrete interfaces the component provides and how they are connected. Ports and port connectors are connected with port types and port connector types, in that they provide a realization of the respective type for the concrete component.

A further important concept is the `VariantPropertyBinding`. This concept allows to bind concrete values to the properties of the component type of the concrete component. That is, variant property bindings realize the assignment of values to *static instance* properties of a variant, compute the value of *calculated* properties and initiate the measurement of *runtime* properties.

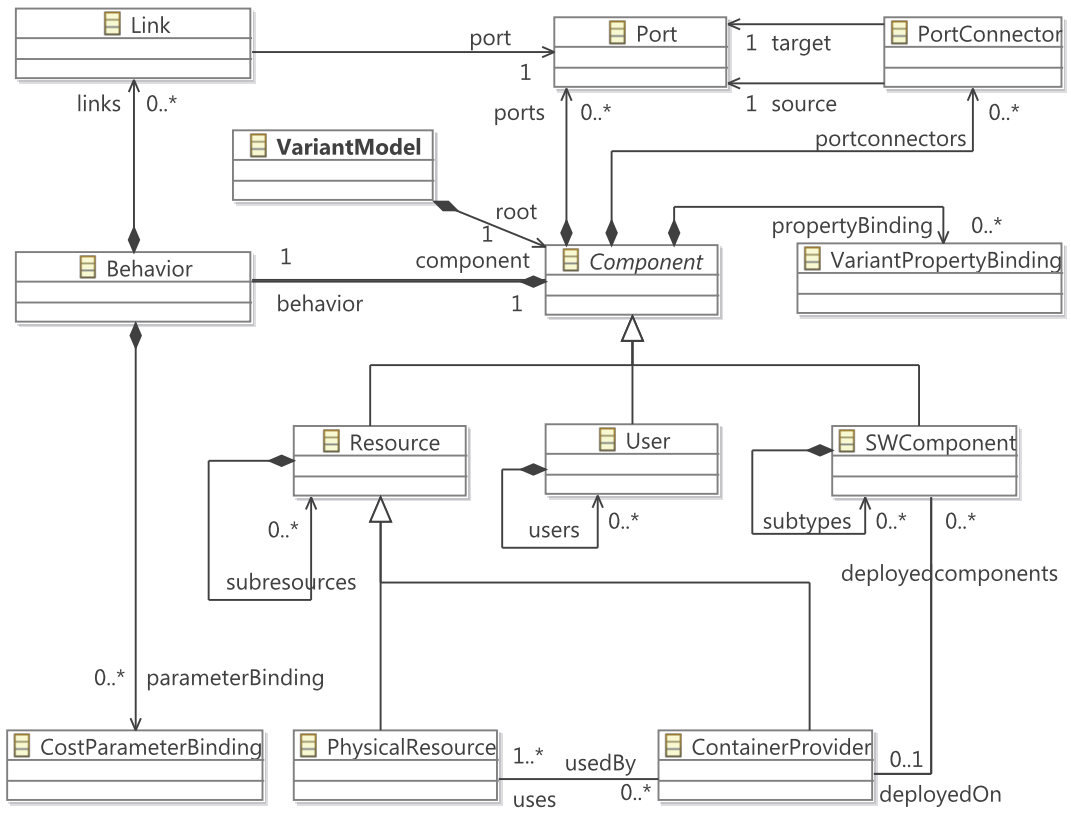


Figure 4.3.: Concepts and their Relations in the Variant Package of the CCM.

Finally, components are connected to a `Behavior`, which connects the respective component with behavior models. `Link`s are used to connect the behavior model with the ports of the component. As will be shown in the next subsection, behavior models include parameterizable templates (`CostParameter`s). These are defined for component types, whereas concrete components have concrete values for these parameters, which is expressed by `CostParameterBinding`s.

An interesting characteristic of the CCM is the relation between structural and variant models. Namely, elements of structural models are types for elements in variant models. For example, a component in a variant models refers to a component type of a structural model as its type. The component type defines properties, which are bound to concrete values for components of this component type by variant property bindings. Thus, the CCM realizes 2D meta-modeling, as described by Atkinson and Kühne [7]. Elements of a variant model have two types: one given by the variant meta-model and another

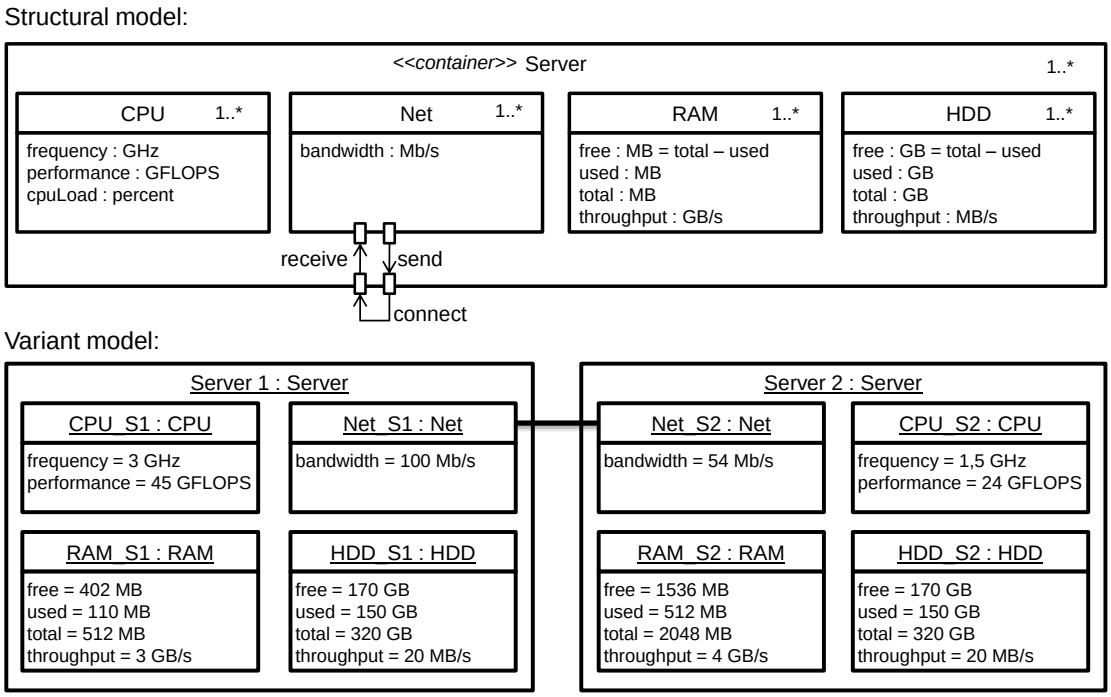


Figure 4.4.: Example of Structure and Variant Model.

one given by instances of the structural meta-model forming an orthogonal *instance-of* relationship. The benefit of 2D meta-modeling for the CCM is extensibility, even at runtime. That is new types of resources, software components or users can be added (or existing types can be removed) at runtime, because the connection between variant and structural models is interpreted at runtime. This feature is likely to prove beneficial in practice, as the self-optimizing system under construction is not bound to a fixed set of resource or software types, but can be extended in unanticipated ways.

Figure 4.4 depicts in its upper part an exemplary structure model specifying the types of a typical server landscape and, in its lower part, a variant model covering a specific hardware infrastructure comprised of two servers.

4.1.3. Behavior Models: Quality-specific Operational Semantics

As stated by requirement /R2/, besides a representation of the currently running system and its architecture, knowledge about its behavior is required to reason about actions (alternatives) of the system. Thus, behavior models have been introduced to the CCM, which serve exactly this purpose. By now the behavior package offers concepts for the prediction of energy-related aspects of the system. Namely, the energy state chart

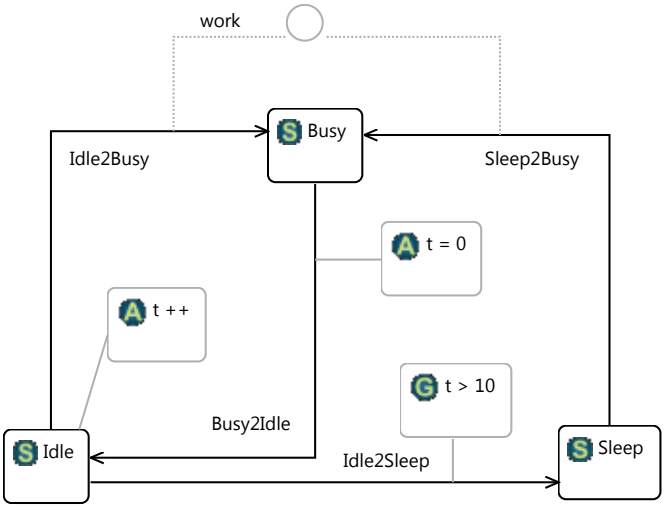


Figure 4.5.: Exemplary Energy State Chart for CPUs.

(ESC) [14] concept as introduced by Benini et al. [14] is supported. But, the package is designed in an extensible way as depicted in Figure 4.6 and described in the following paragraphs.

The root of behavior models form `BehaviorTemplate`s. This abstract concept allows to specify a set of `CostParamater`s for a behavior. These parameters are bound for concrete components as described in the last subsection. Notably, cost parameters are not necessarily fixed values, but can be complex expressions as offered by the expression package. Additionally, behaviors can be composed from other behaviors (`ComposedBehavior`). The interface between behaviors is defined by `Pin`s and `PinConnector`s. Pins are characterized by their visibility (private versus public) and direction (provided versus required). This seemingly redundant interface description enables to specify influencing factors between behaviors of different components, which are not related to the functional connection by ports and port connectors.

The `Occurrence` and `Workload` concepts belong to the workload language, which will be described in more detail at the end of this section. The concepts are depicted, to show their connection to the `Pin`s of behaviors.

The remaining concepts describe the structure of ESCs. They are a extension of classical state charts [65] in that states and transitions have a power consumption rate and transitions have a specified delay. An exemplary ESC for a CPU is depicted in Figure 4.5. It comprises three states: *idle*, *busy* and *sleep*. These three states reflect roughly the principle of current CPUs behavior. In a real case, this ESC would contain a variety of C and P states defined by Advanced Configuration and Power Interface (ACPI). For clarity, the example is reduced to the presented three states. The transitions between

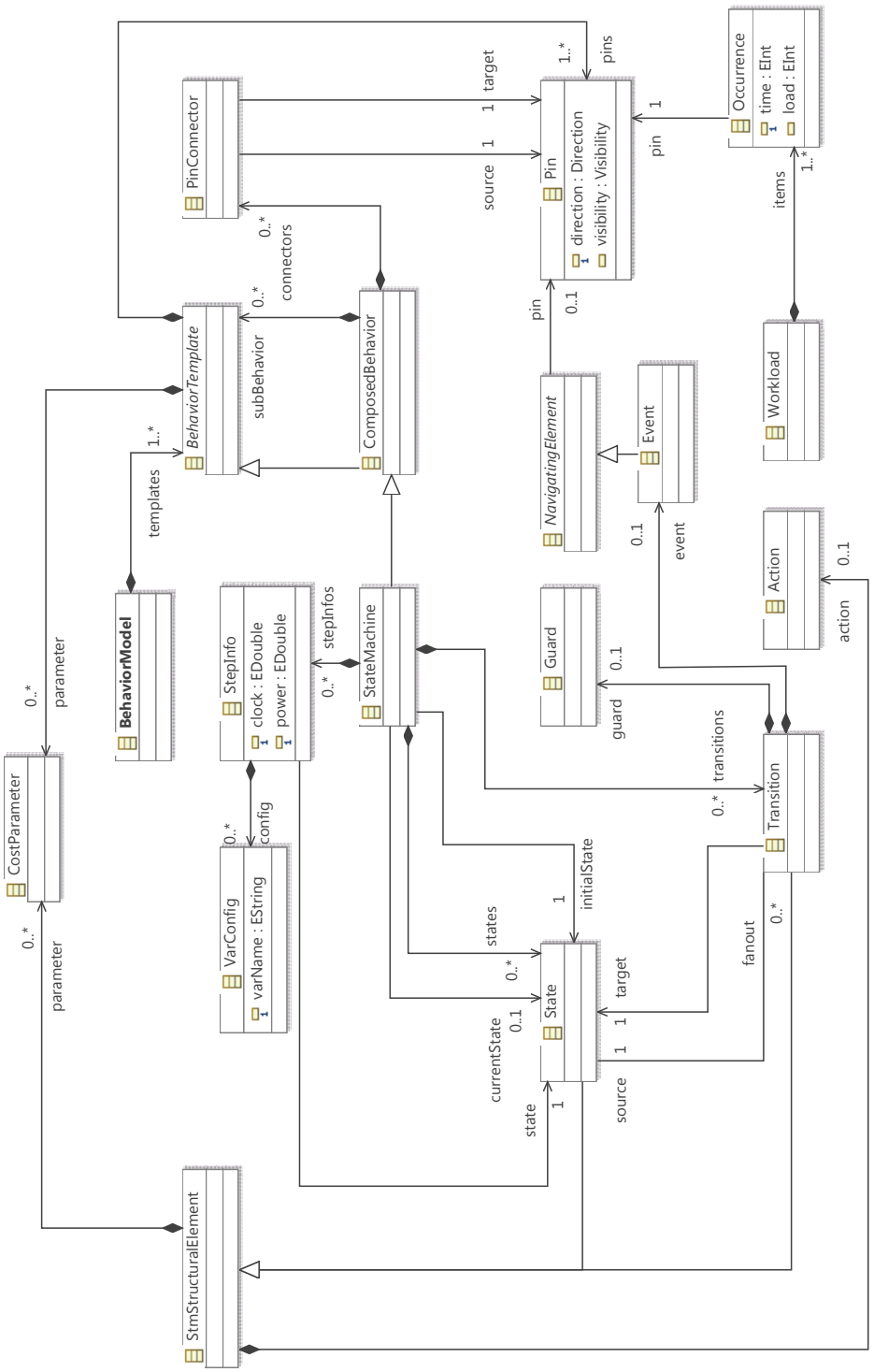


Figure 4.6.: Concepts and their Relations in the Behavior Package of the CCM.

the states are annotated with `Event`s, `Guard`s and `Action`s. In addition, variables can be defined for the ESC. In the example, the variable t has been defined, which is incremented as long as the CPU is in *idle* mode. If more than 10 time units have passed, the transition to the *sleep* state triggers. The transitions from *idle* and *sleep* to *busy* wait for the event *work*, which is the `Occurrence` of a `Pin` depicted as a grey circle.

Each state and transition has a cost parameter for the power consumption rate. Transitions additionally have a cost parameter for the implied delay. This allows to reuse the behavior model (template) for other CPUs, which only differ in the rates and delays. Due to standardization efforts in the hardware community (ACPI) this is the common case. At runtime, a workload against the ESCs is derived and a discrete event simulator (DES) [9] derives the implied total energy consumption for this workload. To enable the analysis of individual steps in the simulation, the concepts `StepInfo` and `VarConfig` exist. Each simulation step is stored in a `StepInfo` entry, covering the accumulated energy consumption and elapsed time units as well as the current variable assignments (`VarConfig`). This data can, for example, be visualized to guide the developer.

ESCs are regular automata without stack. This expressiveness suffices to model the induced energy consumption by hardware resources. To simulate non-functional properties of software components more complex behavior models are required. Such additional types of behavior models can be easily added to the CCM, which offers the `BehaviorTemplate` concept as central point for such extensions.

4.1.4. Shared Platform Concepts

As mentioned, the three main constituents of the CCM (and QCL) share a set of common concepts. These are *expressions*, *datatypes* and *units*. In the following these packages will be explained in more detail.

The Expression Language

Expressions are omnipresent in the CCM and QCL. For example, the ESCs in the behavior package of the CCM use expressions for guards and actions. The cost parameter bindings of behavior templates are expressions, too. The same holds for variant property bindings in variant models. Also the QCL uses expressions in several types of clauses. Figure 4.7 depicts the concepts, which allow for the specification of expressions. For clarity, only the main concepts are shown.

The reason for an expression language, instead of reusing existing languages like the OCL, is better performance (in evaluating expressions) due to the tailored set of features required by CCM and QCL. Standard expression languages could be used, too.

There are six different types of expressions supported by the CCM. First, `ParenthesizedExpression`s, which allow to embrace expressions with parenthesis to influence the evaluation order. Second, `LiteralExpression`s, which denote literals (i.e., val-

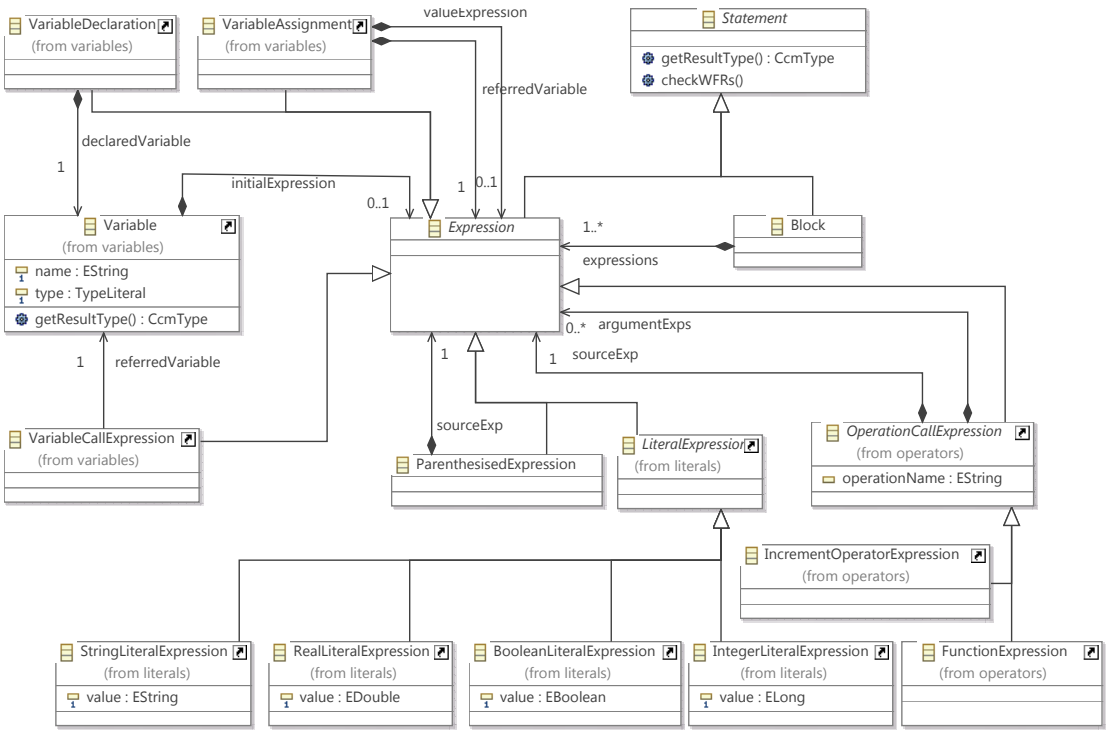


Figure 4.7.: Concepts and their Relations in the Expressions Package of the CCM.

ues) of a certain data type (including strings, integers, reals and booleans). Third, `OperationCallExpression`s, which can be unary or binary. Unary operation call expression only have one source expression to which an operation is applied. For example, the `IncrementOperatorExpression` increments the value of the source expression. Binary operation call expressions apply an argument expression to a source expression. A typical example is addition, where the value of the target expression is added to the value of the source expression. A special kind of operation call expression are `FunctionExpression`s. They are characterized by having at least one variable in the target expression, which is to be set for the evaluation of this expression. For example, `f(x) = x++` is a function expression whose target expression is an increment operator expression, which in turn has a variable call expression as its source expression. For clarity, only two types of operation call expressions are shown in Figure 4.7. The remaining three types of expressions concern `Variable`s. There are expressions to declare variables (`VariableDeclaration`), to assign values to variables (`VariableAssignment`) and to call (i.e., use) variables (`VariableCallExpression`).

To relate a set of expressions to each other, the `Block` concept is provided. Both, blocks and single expressions are denoted `Statement`s, which can be evaluated w.r.t.

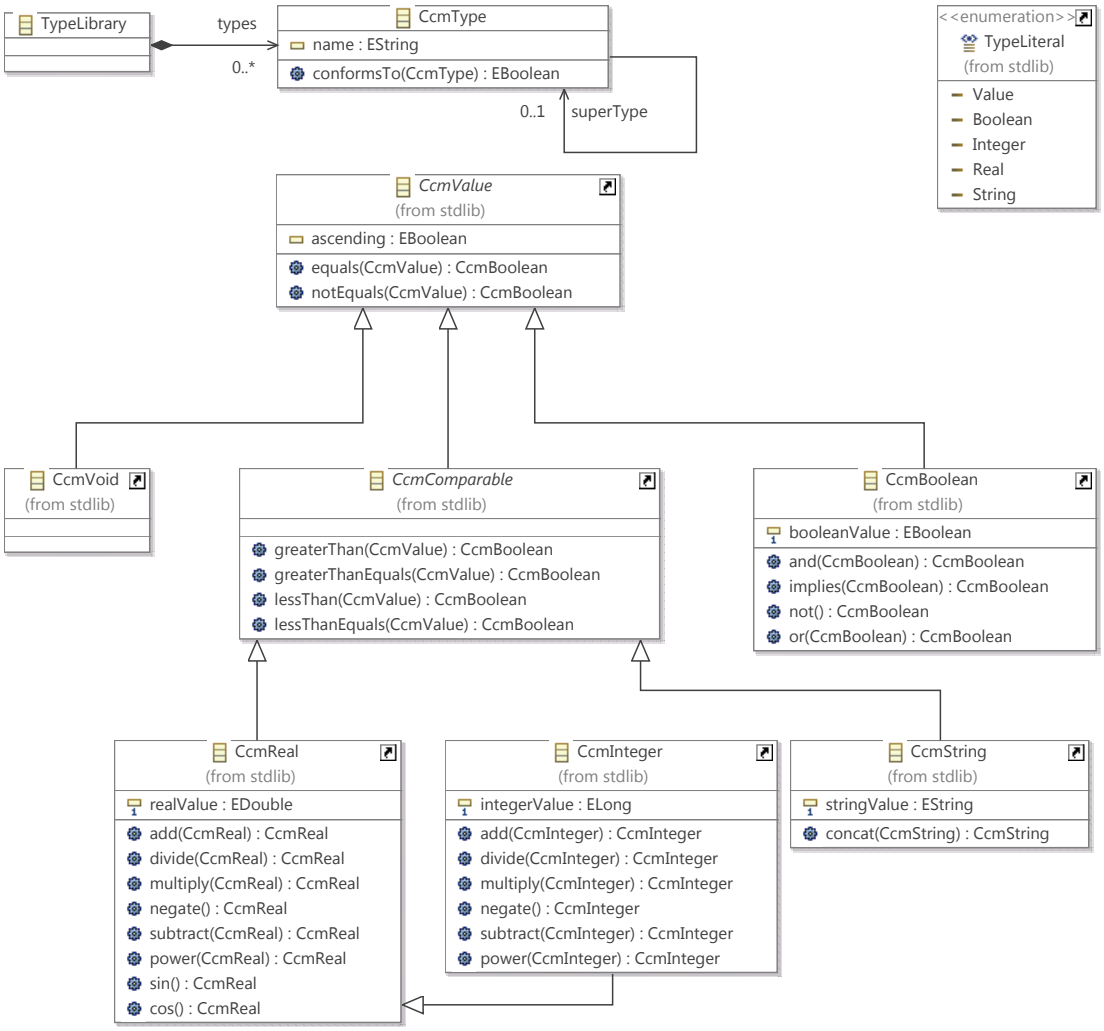


Figure 4.8.: Concepts and their Relations in the Datatypes Package of the CCM.

well-formedness rules (operation `checkWFRs()`) and w.r.t. their concrete value.

Datatypes

The CCM is meant to be platform, especially programming language, agnostic. Hence, an abstraction layer for data types is part of the CCM. Figure 4.8 depicts the related concepts provided by the CCM. The concept `CcmType` allows to specify a hierarchy of types by its supertype relation. Conformance between types can be discovered for concrete types using the `conformsTo()` operation.

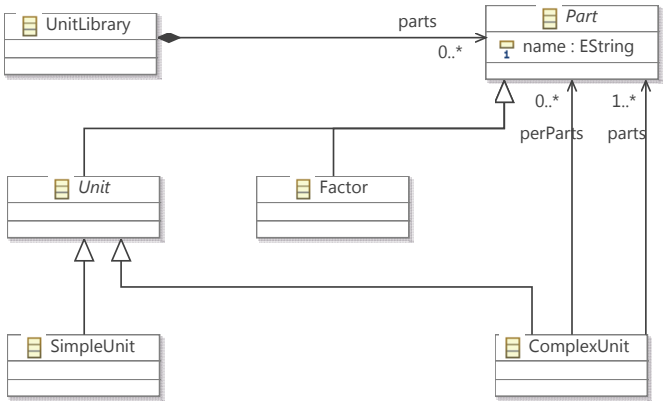


Figure 4.9.: Concepts and their Relations in the Units Package of the CCM.

All types are contained in a `TypeLibrary`. Besides types, also values adhering to these types need to be expressed in the models of a M@RT-based system. Therefore, the `CcmValue` concept is defined, which characterizes a value order as ascending or not. An ascending value order implies that higher values are better than lower values. This is in line with the specification of NFPs in the structural models. The CCM currently supports five data types: *void*, *real*, *integer*, *string* and *boolean*. Integers and reals are *comparable* in that propositions like *greater than* can be made about values of this type. Moreover, each type of value supports a set of operations to be performed. For example, it is possible to compute the cosine of a real value or to concatenate two strings. This functionality is used by the expression package when evaluating the expressions. For each platform to be supported by the CCM the operations shown in Figure 4.8 need to be implemented accordingly.

Units

The modeling and interpretation of NFPs requires to specify the unit of measurement in addition to the data type of the value. For example, the response time could be measured in milliseconds or nanoseconds (unit) and is stored as an integer (data type). If there is no unit specified for an NFP, either all NFPs need to be measured using the same unit or they cannot be put into relationship to each other. If, for example, the response time of some process is measured in milliseconds, but the cpu time required by the same process is measured in nanoseconds (because another profiler is used for this NFP) the knowledge about the units is a prerequisite to compute the ratio of cpu time to response time. This also motivates the need to support the conversion of units. That is a conversion relationship between units is required (e.g., $1s = 1000ms$). For complex units, consequently, complex conversions are required. For example, the throughput of a

```
1 library {
2         simple unit second :  Integer;
3         simple unit Watt :    Real;
4         simple unit MB :      Integer;
5         simple unit Mb :      Integer;
6         simple unit mips :    Integer;
7
8         complex unit Mb_s  =  Mb per second;
9 }
```

Listing 4.1: Example Unit Specification.

hard disk drive could be measured in megabyte per second (MB/s), but the throughput of a (slow) network device in kilobyte per second (KB/s).

Figure 4.9 shows the concepts related to units of the CCM. A `SimpleUnit` denotes an atomic unit, which only has a name and no relationships to other units by itself. A `ComplexUnit`, in contrast, comprises several `Part`s, which can be units or `Factor`s. Factors denote numbers used to specify a conversion in terms of multiples (e.g., $1s = 1000ms$). To realize complex units, which have a *per* relationship between its parts, the `perParts` reference is provided. For example, the definition of the complex unit Hertz ($1Hz = 1/s$) has two parts a factor (1) and the unit *second* as a *perPart* (i.e., 1 per second). This allows to compose units and their conversion logic. For example, the unit megabyte can be defined as a complex unit with two parts: a factor of 1024 and the unit kilobyte. The complex unit megabyte per second only relates the units megabyte and second to each other. For easy accessibility, all units are contained in a `UnitLibrary`, which allows to specify new, custom units, too. Listing 4.1 depicts an example.

4.1.5. Special Purpose Packages

On top of the CCM there are several special purpose packages required for for self-optimizing systems. The currently provided concepts are *requests*, *reconfigurations* and *workloads*. In the following these packages will be explained in more detail.

Requests

The plain invocation of a software system's functionality does not necessarily lead to satisfied users. If, for example, the overall efficiency is the optimization objective, the system could decide for a configuration, which implies very low costs, but provides very low QoS (and, thus, utility) to the user, too. Thus, the user's minimum requirements (as well as objectives) need to be covered in addition. For this purpose, the request language has been added to the CCM. Using this language, the (minimal) QoS requirements of the user for a specific call and the user's objectives can be specified. Figure 4.10 depicts the concepts of the request language.

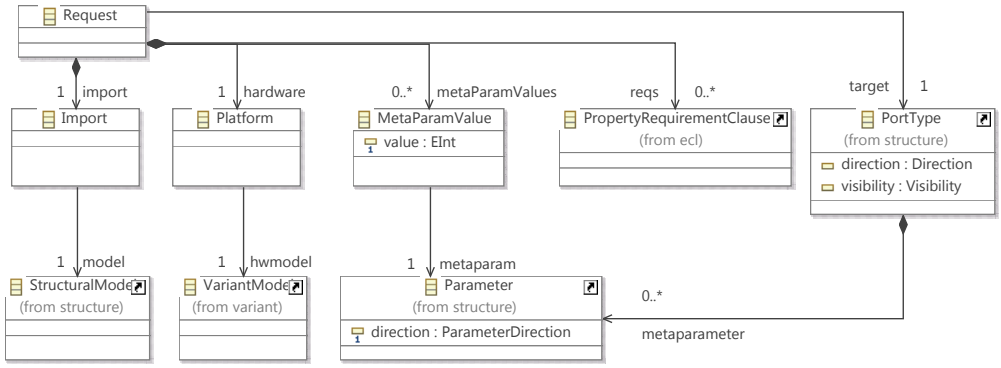


Figure 4.10.: Concepts and their Relations of the Request Language of the CCM.

```
1 import ccm [./sort.structure]
2 target platform [./current.variant]
3
4 call ListUtil.sort expecting {
5         list_size = 200000
6         response_time max: 500.0 [ms]
7         response_time minimize
8         encryption_key_size maximize
9 }
```

Listing 4.2: Example Request to Sort a List.

Each request refers to a current hardware infrastructure (variant model, `Platform`) and imports a structure model representing the available software (`Import`). To specify the functionality to be invoked, the request refers to a `PortType`. This port type has a set of meta-parameters, which provide additional information for existing parameters. For example, a parameter `list` can have a meta-parameter `size_of_list`. This information is required, to abstract from concrete user requests. Such concrete requests define values for all parameters and meta-parameters (`MetaParamValue`). Finally, a request can contain multiple property requirement clauses. Such clauses specify minimum or maximum values of NFPs. They are reused from QCL and will be explained in more detail in the next section. In addition, they are used to specify user objectives by omitting a minimum or maximum value. Listing 4.2 depicts a sample request, where a user wants to sort a list of 200.000 elements, which are confidential and, hence, need to be encrypted. The user is expecting a maximum response time of 500 milliseconds and has the contradicting objectives to get the list sorted as fast as possible and to use the biggest encryption key possible.

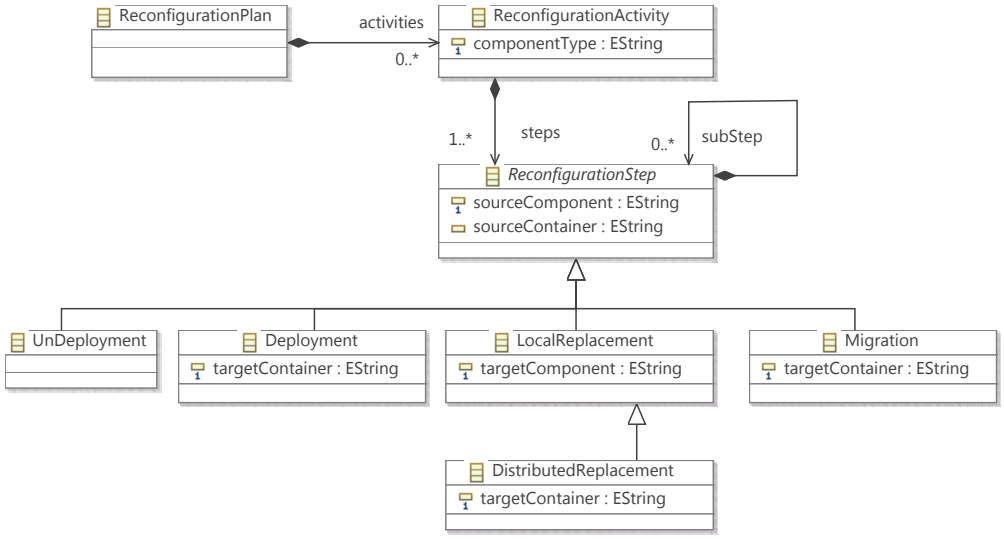


Figure 4.11.: Concepts and their Relations of the Reconfiguration Language of the CCM.

```
1 reconfigurationPlan {
2  reconfiguration for SWComponentType Sort {
3         migrate SWComponent HeapSort from 192_168_0_5 to 192_168_0_10
4  }
5  reconfiguration for SWComponentType Encrypter {
6         replace SWComponent SHA2-512 on 192_168_0_5 with SHA2-1024 on 192_168_0_10
7  }
8 }
```

Listing 4.3: Example Reconfiguration Script.

Reconfigurations

The aim of self-optimizing software is to reconfigure itself continuously to stay in an optimal configuration. Thus, means to describe how to reach a target configuration from a current configuration are required. The reconfiguration language of the CCM serves this purpose. It allows to specify scripts of reconfiguration operations to be applied to the current configuration in order to reach the target configuration. Figure 4.11 depicts the concepts of this language.

The main (root) element of the language is the `ReconfigurationPlan`. It comprises a set of `ReconfigurationActivity`s, which are bound to a component type. Such activities encapsulate a set of `ReconfigurationStep`s to be applied to instances of the respective component type. Reconfiguration steps are hierarchical (i.e., can be nested), which is not necessary for a reconfiguration language, but improves the structure of large reconfiguration plans. All steps refer to at least one component (`sourceComponent`) and one con-

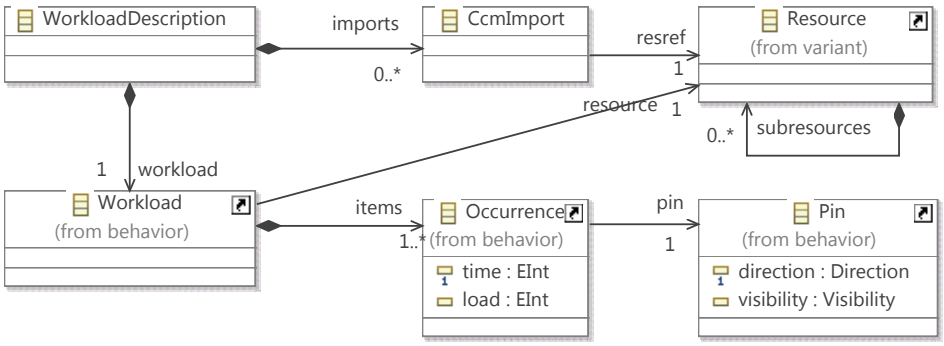


Figure 4.12.: Concepts and their Relations of the Workload Language of the CCM.

tainer (`sourceContainer`). The steps supported by the language are `UnDeployment` (to remove a component from a container), `Deployment` to a `targetContainer`, `Migration` of a component from a source to a target container, `LocalReplacement` of a source component with a target component (both of the same component type) and `Distributed-Replacement` of components including the migration to a new target container. The presented set of reconfiguration steps allows for the specification of all possible reconfigurations.

Listing 4.3 shows an exemplary reconfiguration script. It prescribes the migration of a `HeapSort` implementation from one server to another, the deployment of an `SHA2-1024` implementation on the server where the `HeapSort` implementation was targeted at and the undeployment of the `SHA2-512` implementation from the other server (i.e., a distributed replacement).

Workloads

Finally, the CCM offers a workload language, which allows for the specification of workloads on resources. This feature is required for behavior prediction by simulating behavior models. In this thesis, an example is the simulation of ESCs. The respective concepts of the workload language are shown in Figure 4.12.

A `Workload` comprises a set of `Occurrence`s, which denote events at a specified point in simulation time with a given `load`. The load is to be interpreted in the context of the `Resource` referred by the workload. For example, for a CPU the load can be interpreted as time units or as number of instructions to be performed. Each occurrence refers to a `Pin` of a behavior model. Notably, the workload concept as described does not support the specification of workloads against multiple resources. However, this does not pose a problem, due to the hierarchical structure of the CCM. The workload refers to a resource, which comprises all relevant resources (and, thus, references to their behavior models) for the intended simulation. Details on the simulator can be found in 6.2.

4.2. The Quality Contract Language

As mentioned above, the CCM allows for the definition of NFPs for soft- and hardware component types. For each instance of a component, the values of the NFPs are either fixed, computed or measured at runtime. However, as stated by requirement /R5/, to configure and optimize an application at runtime, the dependencies between component types and especially the dependencies between NFPs have to be declared. For example, it has to be declared what minimum *bitrate* a `Decoder` component has to provide in order for a `VideoPlayer` implementation to provide a certain *frame rate*. That is NFPs cannot be interpreted in isolation, but their relationship among each other needs to be considered. To express such dependencies we developed the QCL, which allows for the definition of contracts for components specifying (probably) multiple quality modes providing and requiring different NFPs (i.e., qualities).

The concept of contracts denotes an assumption/guarantee relationship. More general, if a precondition holds, the associated postcondition will hold, too. This also fits the interpretation of contracts in economics, where a set of partners make a binding agreement on a service (precondition) with a corresponding return service (postcondition). We use the concept of contracts to express the dependencies between components and NFPs. For example, if a certain *bitrate* is provided (precondition) by a `Decoder`, a certain *framerate* will be provided (postcondition) by a `VideoPlayer`.

The use of contracts in software development has been thoroughly investigated by Bertrand Meyer [89]. In a more recent article, Beugnard et al. recapitulate on the application of contracts in software development and discuss different types of contracts: syntactic, behavioral, synchronization and QoS contracts [17]. Syntactic contracts are used to specify the compatibility of components in terms of their interfaces. Behavioral contracts additionally allow to consider the state of the components by pre- and postconditions. Synchronization contracts are even more advanced in that they allow for constraining the order of operation invocations and, thus, allow for coordinating the interaction between components. Finally, QoS contracts extend all prior types of contracts by their ability to consider the non-functional properties of the components.

We extend the QoS contract concept with *quality modes*, because of the inaccurate predictability of NFPs and the concept of *utility* as will be explained in the following. It is hardly possible to determine an exact value for a certain NFP. For example, the response time of an algorithm for a given input has an error due to measurement and due to the execution platform, which, usually, are too complex to provide predictable timing behavior. Furthermore, often a specific value is not of interest, but the range of the value. The frame rate of a video, for example, should be between 20-30 frames per second for the user to perceive a fluent video playback. This example shows the influence of *utility* on the management of NFPs in self-adaptive systems. The users' limited perception tailors the utility gained by an offered quality. It would not improve the utility for a user, if the video playback offers 60 frames per second, because the user is

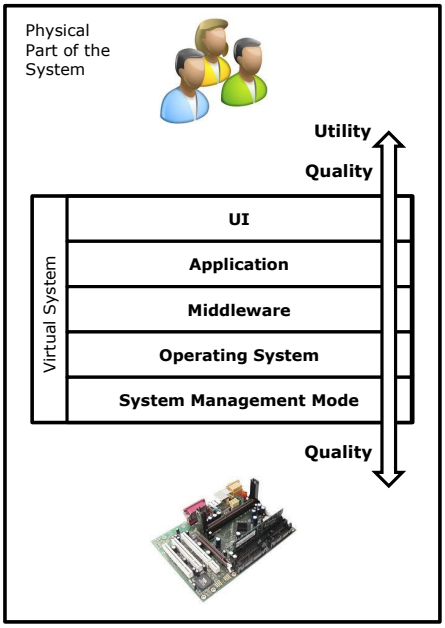


Figure 4.13.: Layering of NFPs. From Quality to User Utility.

incapable of perceiving this improvement. The same can be shown for the response time of an algorithm. The user is incapable of perceiving the difference between 2 milliseconds and 1 milliseconds despite the 50% improvement. Moreover, the user's utility splits the offered quality into ranges. A response time of less than 20 milliseconds is perceived as *immediate response*. If the response time is above 1 seconds, the user notices the disturbance. If the response time further raises, the user will rate the algorithm as slow. Thus, the utility for the user denotes quality ranges. We call these ranges *quality modes* in the context of QCL. Quality improvements or degradations within these modes do not affect the utility of user.

In addition, different types of NFPs can be identified w.r.t. their level of abstraction. There are *low-level* NFPs, which directly relate to a physical phenomenon (e.g., cpu time) and are not affected by any other NFPs. In contrast, *high-level* NFPs are either affected by other NFPs or are fully composed of other NFPs. The frame rate, for example, is affected by a variety of NFPs (e.g., bitrate, cpu frequency, etc.). The abstraction levels are aligned with the layers of the system as depicted in Figure 4.13. To model the relationship between NFPs and utility, it is important to consider the embedding of the virtual system into the physical world. The hardware used to execute software is physical as well as the users of the software. Thus, the origin of NFPs is in physical phenomena, which translate into higher-level NFPs. First, the system managed mode of the hardware, which periodically performs system safety tasks, affects the

```
1  import ccm [../Presenter.structure]
2  import ccm [../Server.structure]
3
4  contract ImageViewer implements software Presenter.show {
5
6   mode lowQuality {
7    requires component Analyzer {
8     accuracy min: 0.5 [%]
9     refreshRate min: 300 [ms]
10   }
11
12   requires resource CPU {
13    frequency min: 400 [MHz]
14   }
15
16   provides accuracy min: 0.5 [%]
17   provides refreshRate min: 300 [ms]
18   provides imageWidth min: 800 [px]
19   provides imageHeight min: 600 [px]
20  }
21
22  mode highQuality {
23   requires component Analyzer {
24    accuracy min: 0.9 [%]
25    refreshRate min: 60 [ms]
26   }
27   /* More requirements and provisions here ... */
28  }
29 }
```

Listing 4.4: Example Contract for an "ImageViewer" Implementation of Presenter Component.

overall behavior of the system. Next, the Operating System (OS) introduces additional system behavior. Optionally, middleware is utilized by the software and imposes further behavior. Finally, the application and its user interface (UI) form two further layers of abstraction. Notably, the application itself can be regarded as a separate layer from the UI, because there are different NFPs regarding the application logic and the interaction part of the application. Finally, the users perceive the complete system (usually via peripheral devices). This final step includes the translation of qualities offered by the system and utility experienced by the users. To keep the system manageable, we merge the application and user interface layer to the software layer as well as all lower layers to the resource layer. Thus, the middleware, operating system and the system management mode are considered as virtual resources, with characteristic behavior.

Thus, a concept used to specify NFPs and their dependencies has to support (1) the distinction between quality and utility, (2) the distinction between software and resources, (3) the definition of ranges (to enable the specification of quality-utility mappings), (4) references to other components and (5) the specification of how an NFP

is connected with other NFPs. Our concept of QoS contracts supports all these requirements. Listing 4.4 shows a sample contract for an *ImageViewer* implementation of a *Presenter* component, which takes data from an analyzing component as input and provides a visualization of this data to the user. The contract specifies two modes: `lowQuality` and `highQuality`. These modes reflect the user's utility function on the quality of the implementation. The user will not perceive quality improvements, which do not fall into these two modes. The `highQuality` mode, for example, provides an accuracy of 0.9 (i.e, 90%) to the user. A higher accuracy cannot be perceived by the user and, thus, does not improve the user's utility. Each mode contains requirements and provisions of NFPs. The provisions are guarantees of a certain NFP to be greater or less than a specified value. For example, the refresh rate of the *ImageViewer* implementation will be at least 300 milliseconds in *lowQuality* mode. The requirements divide into software and resource requirements[1]. In both cases (software and resource requirement), an assumption is made about the NFPs provided by the referenced component. Thus, in the exampe of Listing 4.4, an accuracy of 50% is guaranteed, if there is a CPU providing a frequency of at least 400 MHz and an analyzer component providing the same minimum accuracy as well as a refresh rate of at least 300ms.

Please note that the components referenced by requirement clauses are not concrete components, but types (component types in CCM). Thus, all possible component instances, which fit to the respective type, are to be investigated to determine the validity of the contract. Moreover, the cost and utility implied by the quality modes can be determined, which allows to search for the optimal system configuration. The process of determining valid system configurations using the presented contract concept is called *contract checking*, whereas the search for an optimal configuration is called *contract negotiation*. Both approaches will be explained in more detail in Chapter 6.

In the following chapters, approaches based on QCL contracts will be discussed. To ease the understanding, the concepts and their relationships of QCL will be discussed in the following. Listing 4.4 on page 75 shows an example contract. QCL contracts are models (with a concrete textual syntax), which adhere to the meta-model depicted in Figure 4.14. The root element of these models is the `QCLFile`, which forms a container for all contracts referring to a single CCM component type. To resolve the references to component types, port types, properties and parameters a QCL file has to import references to the respective structural models (`CcmImport`). As there are also references to other contracts, these can be imported, too (`QCLImport`). Each QCL file comprises a set of named `QclContract`s. For software components, the name represents the descriptor of the implementation variant, whereas for resources the name is a unique identifier of the respective resource (e.g., a serial number). Contracts refer to a port type of a component type (e.g., in Listing 4.4 the component type `Presenter` and its port type

[1] There is no need to divide the provisions into software and resource provisions, because provisions are specified for a certain component which either is a software artificat or a resource.

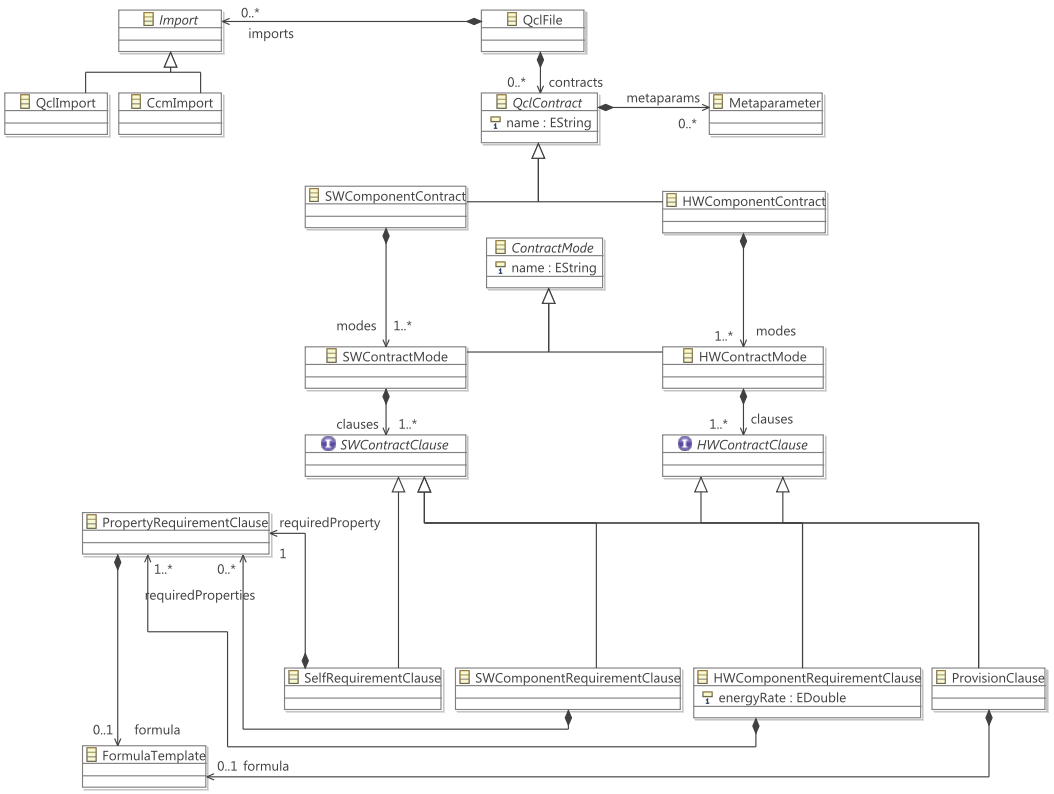


Figure 4.14.: Concepts and their Relationships of the QCL.

`show` are referenced). The meta-parameters of the port type, defined in the structural model of the respective component type, are resolvable due to this reference. Depending on the kind of component type the contract refers to, it is either a `SWComponentContract` or a `HWComponentContract`. This distinction allows to handle contracts for software and resources in different ways and prohibits the accidental use of concepts, which are meant for the other kind of component type. Each contract comprises a set of `ContractMode`s—the quality modes reflecting the mapping from quality to user utility. Finally, the modes comprise clauses, which are either `*RequirementClause`s or `ProvisionClause`s. There are four kinds of requirement clauses: self-, software-, resource- and property-requirement clauses. Self, software and resource requirement clauses all comprise property requirement clauses and differ in the kind of component type they refer to (to the current component itself, to a software component or a resource). Property requirement clauses refer to the properties specified for the component type the enclosing requirement clause refers to. In the example of Listing 4.4 lines 9–1, a

```
1 import ccm [./audio.structure]
2 target platform [./current.variant]
3
4 call AudioUtil.improve expecting {
5         reduction_amount = 12 [dB]
6         loudness_target = -16 [LUFS]
7         response_time minimize
8         file_size minimize
9 }
```

Listing 4.5: Example Request with Multiple Objectives.

`HWComponentRequirementClause` referring to the component type `CPU` is shown, which comprises a single `PropertyRequirementClause` referring to the property `frequency` of the `CPU` component type.

Finally, `Provision-` and `PropertyRequirementClause`s both define ranges of allowed values for the respective NFP. The boundaries can be defined using fixed values, mathematical expressions (using the expression language) or `FormulaTemplate`s. The need for more than fixed boundaries will be explained in more detail in Chapter 5, where the derivation of contracts from a given implementation will be discussed.

4.3. Architectural Aspects of Multi-Objective Optimization

The Cool Component Model along with the Quality Contract Language enable the developer to specify the system subject to self-optimization. To do so, multiple variants and their interrelation can be described using quality contracts. Notably, no concept specific to multi-objective optimization is part of the CCM or QCL, except for the request language.

Specifying a user request against the system includes the specification of the user's objectives and constraints. That is, as already outlined in Section 4.1.5, each request comprises a function to be invoked along with a set of constraints and objectives. Listing 4.5 depicts an invocation of an audio application, which shall normalize the loudness of an audio file to -16 LUFS (suitable for mobile devices) and reduce the noise up to 12 dB. The user has two objectives: the response time should be minimized, i.e., he wants to get the result as soon as possible. On the other hand side, the user wants the resulting audio file to be as small as possible. This is a common objective if many of the processed audio files are to be hosted online where storage space is costly. But, both objectives are conflicting, because a smaller file size either requires to use lower bitrates resulting in worse quality or to apply compression methods which consume time and, hence, conflict with the objective to minimize the response time.

Thus, CCM and QCL only cover the specification of requests with multiple objectives against the software system, but do not include any further special concepts for MOO.

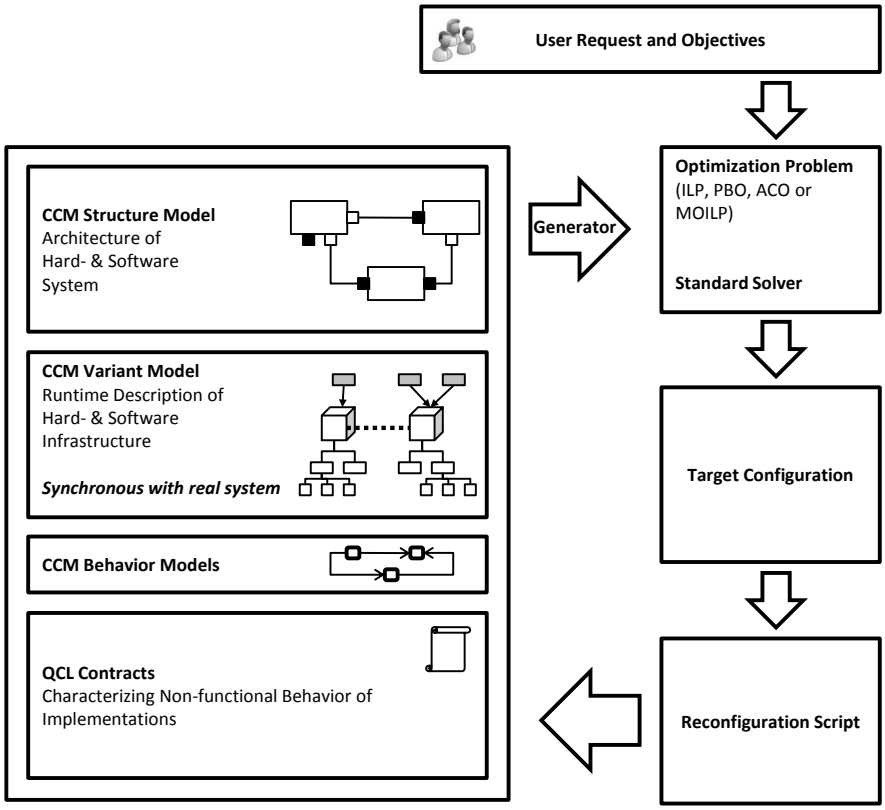


Figure 4.15.: General Architecture for Multi-Objective Optimization in MQuAT.

An architectural style for MOO, as introduced by Arjan de Roo [43], is not part of CCM and QCL. Nevertheless, in Chapters 7, 8 and 9 approaches to MOO based on CCM and QCL are presented. The general architecture is depicted in Figure 4.15.

In summary, the system is modeled by the developer, where the variant models of the CCM are kept in synch with the actual running system. These models are used to generate MOO problems in the respective language (i.e., ILP, PBO, ACO or MOILP). To process these problems standard solvers are used. Their result is used to derive a target configuration, which denotes the best configuration according to the user request and current state of the system. By comparing the current and target configuration a reconfiguration script is derived and executed against the system.

In comparison to Arjan de Roo's MO2 style, MQuAT denotes a fixed MOO architecture, which can be modelled using the MO2 style. The models on the left hand side are adaptable components. The optimizer is a MOO component and the target configuration and reconfiguration script steps are transformation components. The application of de Roo's MO2 style leads to a very abstract architecture of four components only.

4.4. Summary

In this chapter the architecture of MQuAT and its main constituents–the cool component model (CCM) and the quality contract language (QCL)–have been presented. The key demarcating features of CCM compared to other component models for self-adaptive software are the strict distinction between hard- and software, the use of runtime models (i.e., models reflecting the runtime state of the system) and the application of QoS contracts to cover the non-functional behavior of component implementations. In addition, the CCM comprises behavioral models to capture complex, non-functional behavior, which is to be simulated.

The models and contracts described in this chapter form the basis for all approaches to contract negotiation as shown in Figure 4.15 and discussed in Chapter 6.

5
Refinement of Platform-Specific Quality Contracts

This chapter is based on the following publication:

> Sebastian Götz, Claas Wilke, Sebastian Richly and Uwe Aßmann: *Approximating Quality Contracts for Energy Auto-Tuning Software.* In Proceedings of First International Workshop on Green and Sustainable Software, IEEE (2012) 8–14.

QoS Contracts, as presented in Sect. 4.2, are software artifacts, which have to be provided by the developers of self-optimizing software systems. That is a developer is not just in charge of realizing software component implementations, but has to provide a characterization of how his implementation will behave at runtime by means of QoS contracts. But, this task is in the general case impossible, due to two problems:

(1) Execution platform not known at design time. The values of NFPs for a particular software component implementation depend on the properties of the utilized hardware resources. For example, a purely CPU-bound implementation (i.e., an implementation which does almost no input/output operations, except at its start and end, but performs a lot of computation) will execute faster on a CPU with high frequency than on a CPU with low frequency resulting in different values for the NFP `processing time`. In consequence, QoS contracts declaring NFPs of implementations are hardware- and thus, often server-specific (except for homogeneous server farms).

(2) User input not known at design time. NFPs often depend on client input, which is either not known at design time or too many possible user inputs exist. For example, the time required by a sort algorithm depends on the number of elements subject to sorting, which is given (or at least selected) by the client. Further examples are the size of a video to be played, the resolution of an image to be processed and the length of an audio file to be converted. To manage the variety of possible client inputs w.r.t. their effect on the non-functional behavior of the implementation at hand, they are typically classified. But, such a classification is usually resource-specific and—as the available resources are not known at design time—can neither be provided by the developer. Instead, only an abstract classification, which refers to the properties of the available resources can be given. For example, two typical classes of client input result from whether the input fits into memory or not, which obviously depends on the size of the available memory. Thus, contracts are bound in a resource-specific manner to the input parameters of their implementations. Typically not the parameters themselves, but their meta-data (e.g., the size, length or resolution) is what influences the concrete values of NFPs.

Under these conditions, the declaration of QCL contracts cannot be fully performed by the developer. Instead, we identified three phases of contract concretization: First, at *development time* contracts can be expressed as *templates* that need to abstract from the utilized resources and client input. These templates can be provided by the developer. Second, at *deployment time* the concrete resources on which components are deployed as well as their configuration are known and thus, resource influences on NFPs can be approximated resulting in resource-specific contracts. Finally, at *runtime*, when a client actually requests a service provided by a component, the concrete NFPs related to the requested service and utilized resources can be computed by instantiating the contract.

In the first phase, the developer creates a contract template. In the second phase, this template is expanded with resource-specific mathematical functions, which take information about future user interaction (e.g., size of a list to be sorted) as input. Finally, in the third phase, these functions are evaluated against the concrete user request.

The contributions given in this chapter are the following:

- A process to specify QoS contracts at design time and to approximate their concrete instantiations at runtime.

This chapter shows how a combination of micro-benchmarking and multiple linear regression can be used to derive non-functional contracts at deployment time from micro-benchmarks developed at design time. The resulting contracts are meant to be instantiated at runtime to predict the resource requirements (i.e., related NFPs in general) for specific service requests and thus, can be used as a basis to optimize the system w.r.t. the trade-off between multiple qualities. Throughout this chapter a simple sort component is used as a running example for illustration and to discuss the limitations

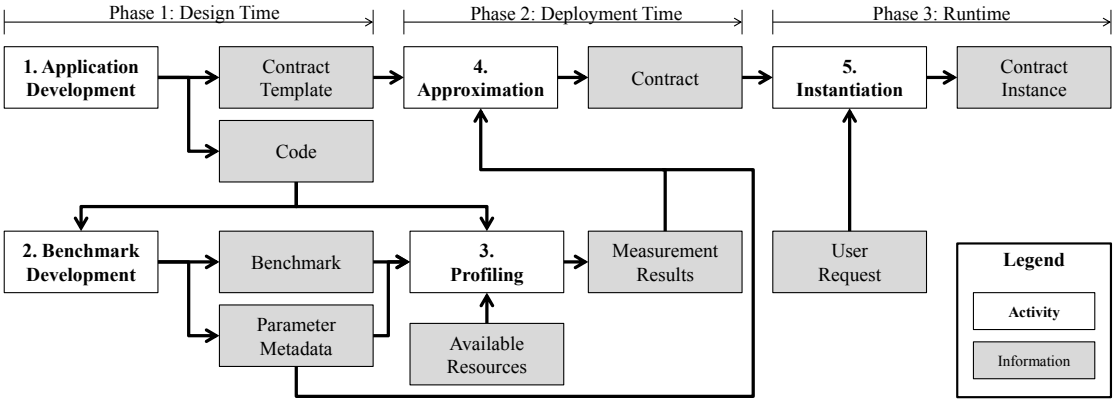


Figure 5.1.: Process of Contract Creation.

and threats to validity of the approach. Finally, an approach to assess the accuracy of QCL contracts is presented.

5.1. The Process of Contract Generation

In this section, we discuss the process of contract creation, consisting of three major phases related to the application's design time, deployment time and runtime (cf. Fig. 5.1). The first phase is at *design time*, consisting of the activities **application development**—where the component implementation (code) is created—and **benchmark development**. By benchmarking (i.e., application of profiling techniques) the developer investigates the non-functional behavior of his code to determine possible quality modes and their demarcating features. In principle, the developer should be able to determine the quality modes of his code without benchmarking, because quality modes cover different levels of the user's satisfaction. Nevertheless, to identify the major influences (NFPs of resources or utilized software components) implying the quality modes, the developer needs to investigate his code more closely by profiling techniques. Using the gained knowledge, the developer is able to create a benchmark, which will characterize the non-functional behavior of the code on each server the code shall be deployed on in the second phase: *deployment time*. In addition, in the first phase, it is important to explicitly express the relevant meta-data of the component's service parameters (e.g., for a sort algorithm, the size of the list to be sorted).

At *deployment time*, the execution of the benchmark (**profiling** activity) leads to measurement results which are correlated with the service parameter meta-data of each benchmark run in the form of mathematical functions mapping the meta-data to the measurement results (**approximation** activity).

```
contract HeapSort implements Sort {
  /* Quality mode for fast CPUs. */
  mode fast {
    requires resource CPU {
      min frequency:1*cpu.frequency.max[GHz];
      min time:<f_cpu_time>(size_of_list)[ms];
    }
    provides min response_time:
      <f_response_time>(size_of_list) [ms];
  }
  /* Quality mode for slower CPUs. */
  mode slow {
    requires resource CPU {
      min frequency:
        0.4 * cpu.frequency.max [GHz];
      min time:
        <f_cpu_time>(size_of_list) [ms];
    }
    provides min response_time:
      <f_response_time>(size_of_list) [ms];
  }
}
```

Listing 5.1: QCL Contract Template for a Sort Component.

Finally, at the *runtime* phase—when a client requests a service—the request's metadata is known and, thus, the approximated NFP functions can be evaluated (**instantiation** activity). The resulting information (e.g., the request- and resource-specific approximated CPU time) is used in later steps of the control loop to optimally configure the system for clients' service requests.

Design Time: Application and Benchmark Design

During design time of self-optimizing software, application developers do not just need to design and implement, but are required to explore the non-functional behavior of their implementations. The measurement of selected NFPs, which enable to negotiate between multiple implementation variants at runtime (e.g., `heapsort` versus `quicksort`), is essential. For example, with regard to the servers' energy consumption, the focus should be on the identification of the required CPU time and the amount of data read and written to hard disk drives and network devices.

To gain the values of NFPs at runtime, the developer cannot simply measure these NFPs on his own machine. The NFPs depend on the utilized resources and the actual user requests. Hence, the developer needs to write a benchmark suite for his implementation, which—in the best case—covers its complete non-functional behavior (i.e., all possible behaviors). This suite is run at deployment time when the available resources are known.

Notably, already at design time, developers are able to identify *quality modes*. These

$$cpu\_time(size\_of\_list) = \\ 11.608,259\ size\_of\_list^2 - 841,968\ size\_of\_list - 1.032,142$$

Figure 5.2.: Approximated Function for Data from Table 5.1.

are setups leading to different levels of user satisfaction in which the code requires different amounts of resources and provides different NFPs in return. For the introduced sorting example, which is a purely CPU-bound algorithm, different frequencies of the CPU lead to valuable differences in the CPU time, which are perceived as immediate (fast) or delayed (slow) response. A developer would, thus, identify two quality modes, which he specifies in a contract template for his code as shown in Listing 5.1. For each mode a *fixture* is specified, which—in the example—represents the CPU frequency. For the `fast` mode, 100% of the maximum available CPU frequency is specified. For the `slow` mode, only 40% are specified. The fixture describes values for NFPs that can vary depending on the service request's input data at runtime. E.g., the NFP `response_time` of the sort example is derived from the function template `f_response_time` time which depends on `size_of_list` of the input data (c.f. Listing 5.1 line 9). In this context, <`f_response_time`> represents a function which is meant to be approximated during profiling and will be replaced by the approximated function during the contract creation process (activity **approximation**).

Quality modes can be seen as modes of operation leading to levels of user satisfaction in a specified system setup (e.g., fixed CPU frequency). In general, two types of setups need to be considered: (1) NFPs can be influenced by the control loop to reach a certain system configuration (e.g., setting the CPU frequency) and (2) the NFPs cannot be influenced (e.g., if the mode is defined to apply for a certain minimum amount of remaining main memory capacity). For optimization (determining the best configuration), this difference is important, because in the first case each mode can be reached and, hence, has to be considered, whereas in the second case only the currently applicable mode needs to be considered, because the other mode cannot (or is not meant to) be reached (setting CPU frequency versus main memory capacity). From an optimization point of view, the rational behind quality modes is to enable executions with lower quality, which in turn lead to lower costs.

In the next subsection, we show how the expressions of function templates can be determined at deployment time.

Deployment Time: Profiling and Approximation

To profile the resource usage of an application, a variety of benchmarking and performance measurement tools is available. Common tools support the measurement of total

Table 5.1.: CPU data (mean ($\overline{cpu\_time}$), standard deviation (σ_{cpu_time}) and standard error of the mean ($SE_{\overline{cpu\_time}}$)) for HeapSort example.

$size\_of\_list$	$\overline{cpu\_time}$ [μs]	σ_{cpu_time} [μs]	$SE_{\overline{cpu\_time}}$ [%]
500.000	1.388,7	749,4	7,6
1.000.000	9.616,3	1.549,4	2,3
1.500.000	23.956,8	2.508,3	1,5
2.000.000	43.951,7	3.210,4	1,0
2.500.000	69.491,5	4.088,5	0,8
3.000.000	100.901,7	4.909,4	0,7
3.500.000	137.647,4	5.674,7	0,6
4.000.000	181.522,2	6.639,0	0,5
4.500.000	230.407,3	7.343,5	0,5
5.000.000	284.935,8	8.449,4	0,4

execution time (e.g., response time, wall time), CPU time, network as well as disk I/O and memory utilization. Developers manually invoke these tools and interpret the results to deduce knowledge about the application's non-functional behavior.

However, the profiling process should be automated where possible to ease the time-consuming process of profiling, approximation and contract instantiation. Thus, at deployment time, the designed benchmarks are used to profile the NFPs of designed software components deployed on a certain server w.r.t. to varying input data (e.g., lists of different sizes). Afterwards, these measurements are used to approximate a function reflecting the correlation between input data and the profiled NFP (cf. Equation 5.2 for an example). In consequence, the prediction of NFPs depending on the service requests' input data is enabled.

For that purpose, we developed a profiling infrastructure that performs the complete deployment time phase automatically. It executes the benchmarks and profiles the server's hardware as well as specific NFPs of software components, resulting in data correlating the software component's input data and its NFPs. Table 5.1 shows example data collected for a heapsort algorithm on a HP Envy 15 laptop[1]). Afterwards, a statistical tool—R [31] or Eureqa [115] in the developed prototype—is used to derive the function from the collected data using multiple linear regression. The resulting functions (cf. Fig. 5.2) replace the function templates in the contract template. The resulting QCL contract is shown in Listing 5.2.

Eureqa [115] is based on an evolutionary algorithm that continuously searches for functions, which approximate the data collected by the benchmark suite. Each function

[1]CPU: Intel Core i7 720QM quad core 1.6 GHz, 8 GB RAM, SATA2 HDD, running Fedora 16 with Linux Kernel Version 3.2.2

```
contract HeapSort implements Sort {
  /* Quality mode for fast CPUs. */
  mode fast {
    requires resource CPU {
      min frequency: 2.0 [GHz];
      min time: 1.147*10^(-6)
        * size_of_list^2-1922 [ms];
    }
    provides min response_time: 2.152
      *10^(-6) * size_of_list^2-1917 [ms];
  }
  /* Quality mode for slower CPUs. */
  mode slow {
    requires resource CPU {
      min frequency: 0.8 [GHz];
      min time: 1.552 * 10^(-6)
        * size_of_list^2-1821 [ms];
    }
    provides min response_time: 3.552 *
      10^(-6) * size_of_list^2-1901 [ms];
  }
}
```

Listing 5.2: Example QCL Contract for a Sort Component.

has a fitness value, which expresses the variance relative to the measured data. To ensure termination, the approximation activity uses a target fitness value to be reached as well as a timeout to terminate the approximation algorithm (the example function shown in Fig. 5.2 has an R^2 of 1.0 (i.e., exactly represents the data of Table 5.1). Figure 5.3 depicts this correlation graphically. The dots represent measurements, whereas the line represents the approximated equation.

Alternatively, as Eureqa is likely to become a commercial tool, the free tool R [31] for statistical analysis allows to perform multiple linear regression to derive a mathematical function approximating the correlation between the parameter meta-data and the measured NFPs. Using R, a deeper understanding of how parameter meta-data and measured NFPs correlate with each other is required (i.e., the type of function). Some NFPs have a linear correlation (i.e., the NFPs in- or decreases linearly with the parameters). Others have a quadratic, exponential or rooted correlation. The application of regression analysis (cf. [66] for an introduction to the field of statistics) allows to determine the coefficients of these functions. If these coefficients are not correlated with each other techniques of linear regression can be applied. For example, in the quadratic function $response\_time(list\_size) = \beta_0 + \beta_1 \cdot list\_size + \beta_2 list\_size^2$ the coefficients $(\beta_0, \beta_1, \beta_2)$ are not correlated with each other. Hence, the approximation of these coefficients can be done by linear regression analysis. The accuracy of the approximated functions can be characterized by the coefficient of determination (R^2), which expresses the deviance of measured to predicted values (using the approximated function) on a scale between 0

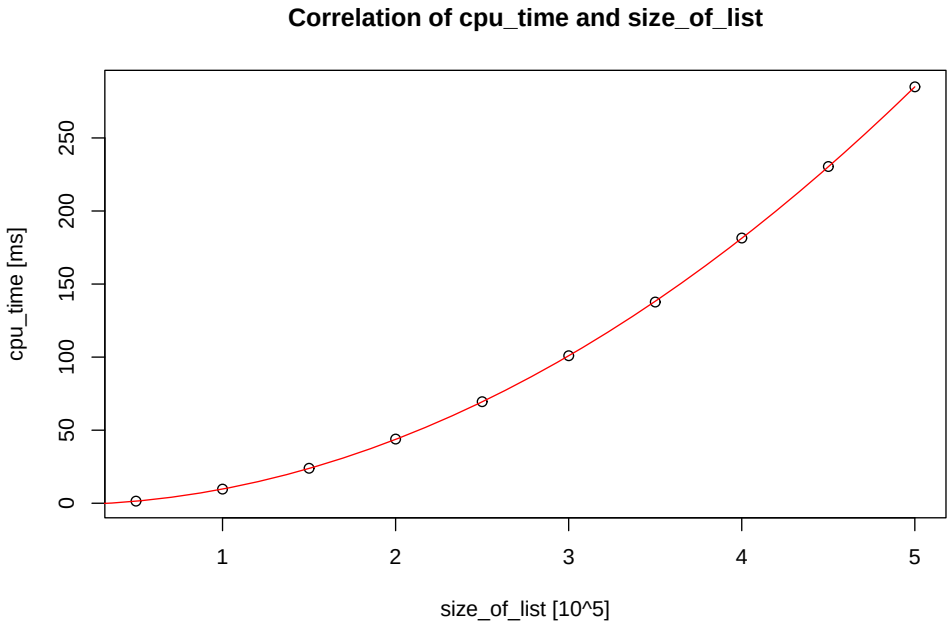


Figure 5.3.: Measurements (Dots) and Regression Function (Line) for HeapSort.

and 1. In Sect. 5.2, we further discuss the accuracy and quality of the profiled data as well as the derived function.

Notably, the benchmark suite needs to be executed for each mode of the quality contract. This is because each mode defines fixed requirements, which have to be fulfilled for the mode to be applicable. These requirements, hence, specify a setup, which is to be performed before the benchmark suite is executed. In our example, the benchmark suite collects CPU and response time once for the CPU running at 100% of its maximum frequency and again at 40% of its frequency. Thus, for the first mode the CPU frequency is to be set up to 100% and for the second mode the CPU frequency is to be set up to 40%. To set the frequency of a CPU various tools exists (e.g., `acpi-cpufreq` for Linux [25]). This setup is part of the benchmark suite. Depending on the resource properties being part of the setup (e.g., remaining main memory, CPU frequency, CPU load or the type of network device), sophisticated setup code results.

Runtime: Contract Instantiation

The third and last phase of quality contract creation in self-optimizing software systems is the instantiation of contracts at runtime, when a client requests functionality of the

```
1  //size_of_list = 500.000
2  contract HeapSort implements Sort {
3    /* Quality mode for fast CPUs. */
4    mode fast {
5      requires resource CPU {
6        min frequency: 2.0 [GHz];
7        min time: 47 [ms];
8      }
9      provides min response_time: 104 [ms];
10   }
11   /* Quality mode for slower CPUs. */
12   mode slow {
13     requires resource CPU {
14       min frequency: 0.8 [GHz];
15       min time: 140.2 [ms];
16     }
17     provides min response_time: 244 [ms];
18   }
19 }
```

Listing 5.3: User Request-Specific QCL Contract.

component implementation characterized by the contract. At this point in time, the functions derived during the deployment phase can be evaluated for the actual user input. In our example, the client could request to sort a list of 500.000 elements, which results in a *contract instance* as shown in Listing 5.3.

These concrete values can now be used for optimization, i.e., to determine the best system configuration (selection of implementations and their mapping to servers) in terms of energy efficiency as we showed in [60].

5.2. Threats to Validity

In this section, we show threats to the validity of our approach and discuss its limitations. For that purpose we focus on three central issues: (1) limitations, due to the resolution of measured NFPs, (2) the accuracy of functions comprised by contracts, and (3) limitations due to correlations of service input data and the profiled NFPs.

Measurement Resolution Limitations

It is important to note that NFPs on the level of an application cannot be measured without an error value. This is, because the application runs on top of an OS, which in turn runs on top of hardware. Both, the operating system and the hardware influence the NFPs of software: on hardware level the system management mode preempts even the operating system and the operating system itself is responsible to assign resources to the application process. In consequence, a minimum resolution w.r.t. a targeted minimum error for the measurement of NFPs exists, which depends on the actual hardware and

OS. On a standard Linux, for example, the minimum resolution of timing information is determined by the frequency of the process scheduler. By default the scheduler runs every 10 milliseconds (ms). In consequence, it is impractical to measure the time required by a method call, which runs in less than 10ms. Though it is possible to use timing information provided by the real-time clock, which is implemented in hardware and usually provides a resolution in nanoseconds, but the scheduler will not delegate the control to another process until the 10ms time slice passed by. More sophisticated schedulers, especially those of real-time operating systems, allow for finer grained time measurements (due to more sophisticated scheduling approaches). Notably, the same minimum resolution can be shown for other NFPs like the amount of data read or written. For data transfers (to a disk or network device) the device driver and file system determines the minimum size of a data block to be read or written.

For the measured sort example depicted in Sect. 5.1, we measured a standard error of 7.6% for lists of 50.000 elements which questions the usability of the approximated resource usage function for lists of this size. Besides, statistical outliers are up to 80% higher and lower than the mean. However, for lists consisting of 200.000 elements the error decreases to 1% and below for even bigger lists whereby outliers decrease to ±12% and to ±5% for 500.000 elements. This matches the assumption that our approach is limited due to minimum measurement resolutions of the hardware infrastructure. Another reason is that during short execution times, unpredictable events like garbage collection or the hardware's system management mode can influence the results more intensely. Only methods, whose NFPs are higher than the minimum resolution can be considered. If the NFPs have lower values, the characterized implementation cannot be compared with other implementations. For the sort example, the parameter `list_size` plays an important role. For lists of less than approximately 150.000 elements, the algorithm runs in less than 10ms and hence, cannot be practically characterized in terms of time. Prolonging the runtime by multiple executions to compute average values for the NFPs does not solve this problem, because the error of such average values are multiples of 100%.

Accuracy of Derived Functions

As every measurement of NFPs introduces an error, the accuracy of the derived functions can never reach 100%. Nevertheless, the error (and deviation) of an approximated function can be estimated. The minimum error is determined by the minimum resolution of the NFP as pointed out in the last subsection. In principle, it is not possible to measure a maximum error in general. For example, the time required for a method to execute can theoretically be delayed forever. Practically, the delay is bound to some external entity, which imposes the delay (e.g., waiting for user input, waiting for crashed servers to reboot). Nevertheless, for implementations which are not affected by such external influences, it is possible to get a practical error estimation by computing the average

error of multiple benchmark executions. Knowing the average error of a particular NFP directly leads to the average error of a derived function on this NFP. For our example function shown in Fig. 5.2, the average error equals the error of the NFP *cpu_time*.

Parameter–NFP Correlation

A third limitation of our approach lies in the fact that services' NFPs are not always related to their input parameters and the parameters' meta-data. Three different types of parameter–NFP correlations (as identified by Seo et al. for the correlation between parameters and computational cost [116]) can be identified:

1. Services always behaving the same, independent of their input data,
2. Services behaving in a predictable manner w.r.t. their parameters,
3. Services behaving unpredictable (e.g., database queries or randomized data).

An example for a service of the first category is the playback of a predefined e-Learning video, where the service's parameters only cover information about the user requesting the video. The sort algorithm is an example for a service of the second class. The third class covers services, which refer to external information, which is not covered (or coverable) in measurements. For example, services using information from unmanaged information sources in the World Wide Web. Their behavior is hardly predictable, as changes to such external information sources cannot/can hardly be predicted.

Whereas NFP approximation for the first two categories is supported, the third kind of services cannot be supported, as approximated functions cannot be based on the input data and, thus, only coarse-grained statistical approximations are possible. To support such services, the external information source has to be included in the approximation approach.

Applicability of the Approach

The approach to quality contracts presented in this chapter is applicable to all types of applications, which fulfill one basic requirement:

> The application's behavior has to be measurable at a resolution, which allows to neglect the measurement error.

In consequence, if the application is to be optimized for response time, the approach is limited to applications, which have a minimum execution time exceeding the measurement resolution of the utilized resources by at least 10 times (to limit the maximum

possible measurement error to 10%). It is important to note that current computer systems provide a very coarse time measurement resolution. Although high precision timers exist, which allow to measure at a granularity of nanoseconds, the effective resolution for measurements at application level is much lower due to disturbances and uncertainties introduced by the operating system, virtual machines and middleware. For example, time measurements in a Java virtual machine are able to retrieve timestamps in nanoseconds, but on a current PC[2] provide an effective resolution of at most 10 milliseconds. Thus, the presented approach is impractical for online transaction processing applications (e.g., an online shop), whose methods are usually processed in less than a millisecond.

The approach is feasible for data-intensive applications, which have execution times of at least a few seconds. For example, audio processing as performed by auphonic[3], video transcoding, image processing and scientific computations.

5.3. Summary

In this chapter we presented a process capable of handling the server and user dependencies of NFPs. The process of contract creation, divided into three phases, has been presented in Sect. 5.1. Furthermore, we discussed threats to validity in Sect. 5.2. The three phases of contract creation comprise design time, deployment time and runtime. At design time, the developer is in charge of writing a benchmark suite for his implementation and to specify a contract template, which contains information about possible quality modes. Each of these modes is specified in terms of a fixture, which denotes certain settings of resource properties (e.g., CPU frequency). In addition, the provided and required NFPs are abstracted by function templates. The developer only specifies the relevant parameter meta-data, like the length of a music file to be played. At deployment time, the benchmark suite is executed and the server-specific expressions for the function templates are computed. Finally, at runtime, the user input is used to compute the parameter meta-data, which allows evaluating the functions. The resulting NFP values are then used to optimize the software system.

[2] Alienware X51, Intel i7-2600, 8 GB RAM, 64bit Windows 7, Oracle HotSpot JVM 7u21
[3] http://www.auphonic.com

6

From Contract Checking to Economic Multi-QoS Contract Negotiation

The notion of QoS contracts allows to cover the non-functional behavior of as well as the interaction between components of a complex hardware/software system. QoS contracts can be leveraged in multiple stages as show in Figure 6.1. The most basic functionality enabled is *contract checking*: an evaluation of a system against its contracts. Such an evaluation reveals whether the system is running correctly, or that it deviates from it's expected behavior. Moreover, contract checking does not necessarily need to be applied to the currently running system. Instead all possible system configurations can be evaluated by this approach. In consequence, *valid* system configurations in terms of contractually specified expected behavior can be determined. If the system configuration takes uncontrollable, continuously changing external parameters into account, then contract checking suffices to build a self-healing software system. The system continuously performs contract checking to determine, whether the current configuration behaves as intended and, if this is not the case, determines the next valid system configuration to reconfigure to.

For self-optimizing software systems contract checking does not suffice, because for such systems the aim is not just to run an arbitrarily valid system configuration, but the currently best possible one at all times. Thus, a mechanism to determine the best system configuration of all possible ones is required: *contract negotiation*.

Contract negotiation denotes a global optimization problem of a system of components, which are known and controllable by central coordinators as known from

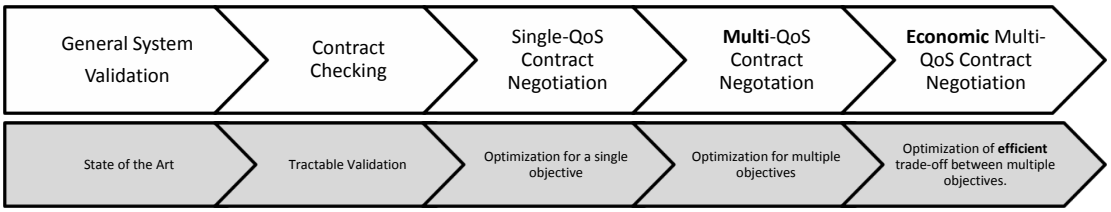


Figure 6.1.: From Contract Checking to Economic Multi-QoS Contract Negotiation.

the self-adaptive system's community. Please note that in this thesis contract negotiation does not denote the process of biding between autonomous components as known from the self-organizing system's community.

As shown in Figure 6.1, contract negotiation can be divided into three stages. The first stage covers the optimization of a single objective (e.g., performance). More advanced is the optimization for multiple objectives, which demands for techniques from multi-objective optimization. Even further advanced are approaches to contract negotiation, which consider the efficiency (i.e., economics) of the system whilst searching for an optimal solution in the presence of multiple objectives. In this context, efficiency is a special additional objective, which is to be optimized in combination with all other objectives. Such approaches seek for an optimal trade-off between the objectives. As pointed out in Section 2.1, a key challenge for self-optimizing systems is this consideration of multiple qualities in combination. Notably, the negotiation of multiple qualities faces the problem of incomparable NFPs. For example, read/write throughput and persistent energy (battery) can hardly be put into relationship to each other. Indeed, most relationships can only be identified within a certain usage context. If, for example, the persistent energy is used to power an additional network device, the throughput increases. In this example, one NFP has been traded for the other. The counterexample is to save energy for the sacrifice of less throughput. Depending on the characteristics of NFPs (e.g., perishability and fungibility; see Section 2.1) such alternatives can be negotiated. In order to do so, these relationships need to be reflected in QoS contracts. The final requirement for multi-QoS contract negotiation is to aim for a balanced system in terms of economic resource utilization. That is the trade-off between NFPs has to be negotiated in terms of efficiency: achieving the best possible quality for the least possible cost.

The purpose of this chapter is to introduce and show the advancement of economic multi-QoS contract negotiation in comparison to contract checking and general approaches to self-adaptive systems, which do not utilize QoS contracts. Therefore, in the following, first an approach for dynamic contract checking based on the CCM and QCL is presented, which is intended to show how the concepts introduced in Chapter 4 can be utilized. Finally, a short, high-level introduction to economic multi-QoS contract

negotiation is given and an approach to handle the general issue of merging objective functions is shown. The concrete approaches to contract negotiation will be presented in the Chapters 7 and 8.

The contributions given in this chapter are the following:

- An argumentation on how contract negotiation using QoS contracts advances over state of the art. (Section 6.2).
- A new approach to merge multiple incomparable objectives by discrete event simulation. (Section 6.2).

6.1. Contract Checking

The evaluation of QoS contracts against a set of system configurations bases on the variability provided by the system's meta-model. In the case of CCM this variability is threefold: (1) component types have multiple implementations, (2) software components are deployable on multiple containers and (3) software components can operate in different quality modes (which denote the utility perceived by their user).

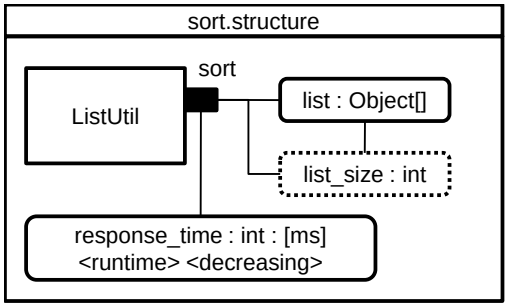
Contract checking allows for the determination of valid system configurations for a given user request–i.e., the determination which component implementations, deployed on which resources, are able to fulfill a specific user request including his QoS demands whilst adhering to the system's constraints.

Listing 6.1 depicts a user request to sort a list of 200.000 elements, which are confidential and, hence, need to be encrypted. The user specifies that he requires the list to be sorted in at most 500 milliseconds. Additionally, he specifies two contradicting objectives: the list should be sorted as fast as possible, but the size of the encryption

Listing 6.1: Request to Sort a List.

```
1 import ccm [./sort.structure]
2 target platform [./current.variant]
3
4 call ListUtil.sort expecting {
5   list_size = 200000
6   response_time max: 500.0 [ms]
7   response_time minimize
8   encryption_key_size maximize
9 }
```

(a) Textual Representation of Request



(b) Referenced Structure Model

Figure 6.2.: Example Request to Sort a List.

key should be as big as possible, too. The referenced CCM structure model (cf. line 1 in Listing 6.1 and Fig. 6.2(b)) comprises the `ListUtil` component type and its `sort` port type, which has a parameter `list` (meant to contain a list of elements to be sorted) and its corresponding meta-parameter[1] `list_size`. In addition, the NFP `response_time` is defined for the `ListUtil` component type as a decreasing NFP (the lower the value, the better), which is measured in milliseconds. The imported CCM variant model (cf. line 2 in Listing 6.1) links the request to the current system configuration as a starting point, because only if this system configuration is not able to fulfill the user's request, another valid configuration will be searched.

In general terms, the process of contract checking is always initiated by such a request and can be summarized in the following five steps:

1. **Quality Mode Selection:** Iterate over all contracts of the requested software component type and select those quality modes, whose provide clauses fulfill the user demands.

2. **Resolve Quality Paths:** For each determined valid quality mode, follow *recursively* all software component dependencies, which always define a demand onto the component they depend on, and perform step 1 (i.e., find all valid modes).

3. **Collect Resource Requirements:** For each determined *quality path* (i.e., set of connected quality modes) collect all resource requirements as defined in the respective contracts.

4. **Derive Possible Mappings:** Derive all mappings between implementations and resources, adhering to the constraints of the structural model.

5. **Derive Valid Mappings:** For each quality path, eliminate all mappings, which do not fulfill the resource requirements.

Interestingly, if there are no cyclic dependencies between software components, contract checking can be expressed as an attribution as known from attribute grammars [77]. This is because all concepts are expressed using meta-models, for which Karol et al. have shown the general applicability of attributions [28]. Whether the application of attribute grammars is beneficial for contract checking is subject to future work.

In the following, we will elaborate on the five steps of contract checking based on a simplification of the *Non-intrusive Alzheimer disease detection* example from the medical domain [67]. The goal of non-intrusive Alzheimer disease detection is to use magnetic resonance tomography (MRT) pictures of a human's brain to detect so-called voids (i.e., missing data), to classify these voids into sediments (indicating Alzheimer) or obstacles and to defer whether the patient has Alzheimer disease or not, based on the number

[1] cf. Sect. 4.1 for a discussion of meta-parameters

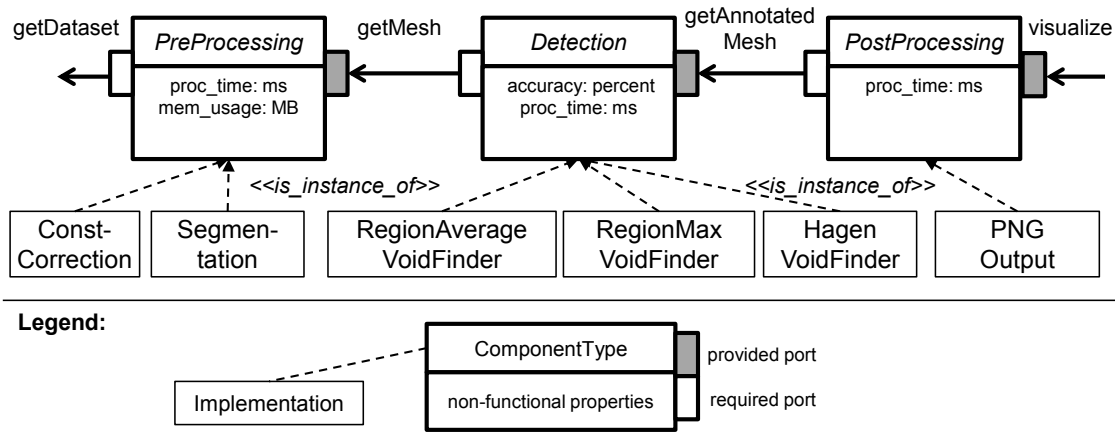
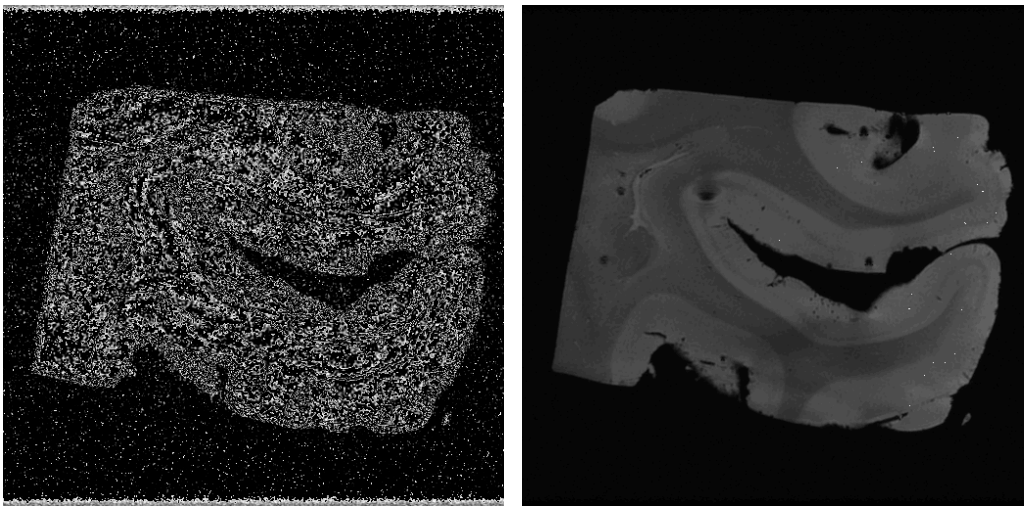


Figure 6.3.: Architecture of Non-intrusive Alzheimer Disease Detection Application.

of identified sediments. Prior approaches to detect Alzheimer disease were based on histological analytics, requiring the extraction of probes from the human's brain. This extraction cannot be performed without harming (indeed killing) the patient and, hence, prior approaches have only been applied posthumous.

The architecture of the software application to detect Alzheimer disease based on MRT pictures is shown in Figure 6.3. It comprises three component types: a `PreProcessor`, `Detection` and a `PostProcessor`. During preprocessing, the MRT pictures (which contain noise due to the nature of measurements) can be adjusted using a constant correction algorithm (`ConstCorrection`) or a sophisticated `Segmentation` approach. In addition, the raw data from the MRT files is transformed into a 3D mesh data structure. The detection of voids being sediments is supported by three implementations: `RegionAverageVoidFinder`, `RegionMaxVoidFinder` and `HagenVoidFinder`. Each algorithm performs a quick search across all points in the 3D mesh, initiating an in-depth search for each point fulfilling certain requirements (called candidate). The in-depth search examines all surrounding points of the candidate according to a set of criteria which are different per algorithm. If the in-depth search evaluates true, the candidate is marked as void. Finally, the postprocessing step transforms the enriched 3D meshes (each representing a set of annotated MRT slices) into a visual format. In the example, the data is transformed to PNG image files. Of course, other formats (especially 3D formats) would be suitable, too. Figure 6.4 depicts an input slice and it's corresponding output, where sediments are marked by white points.

As Figure 6.3 illustrates, the three components form a filter pipeline. To read and write the dataset of medical images, the application uses the Insight Segmentation and Registration Toolkit (ITK). The `getDataset` port type of the `PreProcessing` component type uses the digital imaging and communications in medicine (DICOM) libraries of ITK



(a) Raw Image of Brain Slice from MRT (b) Processed and Annotated Slice

Figure 6.4.: Annotation of Sediments in Brain Slices Indicating Alzheimer's Disease.

```
1  import ccm [./Alzheimer.structure]
2  target platform [./current.variant]
3
4  call PostProcessing.visualize expecting {
5          nrSlices  = 238
6          sliceSize = 512
7          total_time max: 15 [min]
8          total_time minimize
9          file_size minimize
10 }
```

Listing 6.2: Example Request to Alzheimer Detection Application.

to load the image files. The provided `getMesh` port type transforms the image into a 3D mesh suitable for the detection and annotation of sediments. The `getAnnotatedMesh` port type passes the mesh to the `PostProcessing` component type, which transforms the mesh to a set of PNG files (one per slice).

A typical user request for this application is the invocation of the `visualize` port type, with demands on the total processing time and the accuracy of void detection. Listing 6.2 shows a sample request, which denotes the investigation of a dataset covering 238 slices, which each have a size of 512×512 points, with the demand to get the result (i.e., the annotated PNG files) in at most 15 minutes. Additionally, the users objectives are to get the results as soon as possible (minimize total time) and to save as much disk space as possible (minimize file size). The contract of the `PNG Output` implementation

```
1  contract PNG implements software PostProcessing.visualize {
2   mode full_size {
3    requires component Detection {
4      sliceSize = this.sliceSize
5    }
6    requires resource CPU {
7          frequency min: 2.0 [GHz]
8          cpu_time <f_cpu(nrSlices,sliceSize)>
9    }
10   provides proc_time <f_proc(nrSlices,sliceSize)>
11   provides total_time = proc_time + Detection.proc_time
12  }
13 }
```

Listing 6.3: QCL Contract of PNG Output Postprocessor.

of the `PostProcessing` component type is shown in Listing 6.3. Based on this example, we will now discuss the individual steps of contract checking:

(1) Quality Mode Selection. To handle the request shown in Listing 6.2, the contracts of the `PostProcessing` component type are evaluated. In the example shown in Figure 6.3 there is only one contract: *PNG Output*. For this contract all modes are traversed, whereby the `provides` statements are evaluated against the user demand. For each mode, the concrete values of each function template are computed using the meta-parameter values provided by the request. Assume, for the `full_size` mode of *PNG* as shown in Listing 6.3 that the processing time evaluates to 4 minutes. It, hence, potentially fulfills the user demand, because the `total_time` demand of less than 15 minutes is not necessarily violated (depending on the processing time of the detection component). The value order of the NFPs under investigation plays an important role during quality mode selection, because the decision whether a user demand is violated or not depends on the guarantee given by the contract (minimum or maximum value) and the value order. If, for example, a contract ensures a minimum response time t_{min}, and a user demands for a maximum response time t_{max}, the user's demand cannot be guaranteed to be fulfilled, because even if $t_{min} < t_{max}$, it is possible that the concrete response time is greater than the maximum response time demanded by the user (it is only ensured to be greater than t_{min}).

(2) Resolve Quality Paths. The determination of `PostProcessing.full_size` quality mode is only the starting point for contract checking. For each determined quality mode the software component dependencies are resolved. That is, for each `requires component` entry the same process as for the user request is executed. In our example, the *PostProcessing* component type defines a dependency to the `Detection` component type along with a demand: the detector's `slice size` needs to be the same as for the

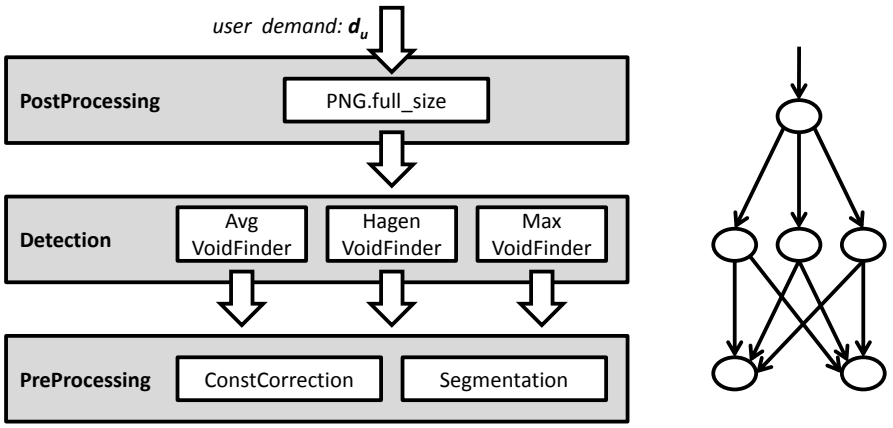


Figure 6.5.: Valid Contract Paths for a Sample User Request. Depicted as Flow of Demands (Left) and as Directed Graph (Right).

post processor. The evaluation of this requirement is indeed performed exactly like the user request to the `PostProcessing` component type. That is, each contract for the software component type `Detection` is evaluated with regard to the demand of a slice size of 512×512. Hence, the evaluation of a single user request leads to a set of valid contract modes, as depicted in Figure 6.5. The resulting structure is a directed graph of connected modes. Cyclic dependencies are detected by keeping a history of visited quality modes. Each path through the resulting graph denotes a possible *quality path*. For our example six such variants exist: two for each implementation of the `Detection` component type.

(3) Collect Resource Requirements. Now that we know which sets of quality modes are valid (i.e., the valid quality paths), we can collect the requirements onto hardware resources defined by these quality modes. The `PNG.full_size` mode puts three requirements onto resources as defined in Listing 6.3: (1) a `CPU` is required, which (2) needs to have at least a `frequency` of 2 GHz and (3) will be utilized for a certain `cpu_time`.

(4) Derive Possible Mappings. Assume a hardware infrastructure comprising two servers with the following properties: *Server 1* has a `CPU` with 3 GHz and 512 MB of `RAM`, *Server 2* has a `CPU` with 1.5 GHz and 2048MB of `RAM`. To determine valid mappings between implementations and servers (or more specific: component containers) w.r.t. the quality paths, the offered properties need to be compared with the required ones. With no requirements specified, one implementation of each component type can be deployed on any server. In our example this leads for the quality path `PNG.full_size-Hagen-VoidFinder.full_size-ConstCorrection.fast` to eight possible mappings as denoted

Table 6.1.: All Possible Mappings for Quality Path (PNG.full_size, Hagen.full_size, ConstCorrection.fast) onto Two Servers. Invalid Mappings are Shadowed.

	Server 1	**Server 2**
lightgray (1)	PNG, Hagen, ConstCorrection	
(2)	PNG, Hagen	ConstCorrection
(3)	PNG, ConstCorrection	Hagen
(4)	Hagen, ConstCorrection	PNG
(5)	ConstCorrection	PNG, Hagen
(6)	Hagen	PNG, ConstCorrection
(7)	PNG	ConstCorrection, Hagen
(8)		PNG, ConstCorrection, Hagen

in Table 6.1. Each component implementation is deployed exactly once. In the example, the different implementations do not support parallel execution. To ensure that only one implementation of each component shall be deployed, the upper and lower bound of each component type can be configured in the structural model.

(5) Derive Valid Mappings. The derived eight possible mappings are checked w.r.t. which resources they require (as defined in the contracts) and the resources provided by the top-level resources (servers) they shall be mapped to. Remember that *Server 2* has a slower `CPU` (1.5 GHz) in comparison to *Server 1* (3 GHz). According to its contract, the *PNG* implementation can only be mapped onto servers having a `CPU` with a `frequency` of at least 2 GHz. In consequence, all mappings, which map the *PNG* implementation onto *Server 2* are invalid. Thus, in our example, only the mappings (1), (2), (3) and (7) remain valid. Now assume that the contract for the `Hagen` detector states that for the `full_size` mode at least 1GB of `RAM` is required and remind that *Server 1* has only 512 MB of `RAM`, whereas *Server 2* has 2 GB. Hence, each mapping, where the *Hagen* detector is mapped onto *Server 1*, is invalid. Namely these are the mappings (1) and (2) of the remaining mappings. For the last check, only the mappings (3) and (7) have to be considered. Let's assume that the contract for `ConstCorrection` preprocessor does not state any resource requirements exceeding the limits of both servers. This leads to a final set of two valid mappings: (3) and (7) as depicted in Table 6.1.

As shown in the last five paragraphs, contract checking derives all valid system variants by evaluating CCM models and QCL contracts against a user request. The complexity of this algorithm is exponential to the depth of dependencies between software components. In particular, it depends on the number of implementations per component type, the number of resources of the infrastructure and the number of quality modes per contract. The dependency depth affects contract checking most, because each dependency leads to a duplication of quality paths by the number of quality modes of the referenced

implementations. Each identified quality path has to be checked against the resources provided by the infrastructure.

The contract checking approach described above has been implemented as an algorithm traversing the CCM models and QCL contracts.

6.2. Economic Multi-Quality Contract Negotiation

In this thesis, contract negotiation is about solving a configuration problem constrained by QoS contracts. It covers the identification of component implementations and their mapping onto resources to serve a given user request with least possible cost whilst still satisfying the user. The selection of appropriate component implementations, their mapping to resources and the determination of component parameters is a configuration problem as formally described by Klein, Buchheit and Nutt in [76].

> Configuration mainly comprises the selection (and instantiation), parametrization, and composition of components out of a pre-defined set of types in such a way that a given goal specification as well as a set of constraints characterizing the domain in general will be fulfilled. [76][p.1].

Contract negotiation as the search for the best system configuration demarcates from dispatching approaches in its scope. By contract negotiation the *globally* optimal system configuration is searched, whereas dispatching approaches seek *locally* for the next best choice. It is important to note that such a local search cannot guarantee a globally optimal system configuration. This is because the influence of all choices on each other is not considered in a local search. Nevertheless, local dispatching is less complex (and, hence, less compute-intensive) than contract negotiation. Thus, a promising approach is an extended dispatching approach, which performs contract negotiation for the next N decisions to be made (instead of only one in the local dispatch case or all in the case of contract negotiation). The investigation of such an approach and the determination of an optimal value for N is subject to future work.

In the following, a discussion is given on how the presented contract negotiation approaches advance over state of the art and an approach to merge multiple objectives into a single objective is presented, whereby single-objective optimization approaches get applicable for multi-objective scenarios.

Advances Over State of the Art

In this thesis, the first type of configuration (selection and instantiation) is addressed by the contract negotation approaches presented in the following two chapters. The advancement over state of the art in self-optimizing software system is twofold: (1) **the**

use of QoS contracts, which renders the search for an optimal system configuration tractable (i.e., drastically reduces the search time in comparison to a holistic search) and (2) **the generation of the optimization problems** at runtime using models of the system's structure and runtime state.

The use of QoS contracts reduces the complexity of the system description. This is because they explicitly describe the relation between the NFPs of different system components. In the current literature these dependencies are implicit knowledge, which has to be derived on demand, often leading to less specific dependencies allowing for more variants. But, the more variants have to be investigated, the more time is required to determine the optimal variant.

In existing work on self-optimizing software systems, the optimization problems are parametrized by the current system's state. Current approaches provide the developer a methodology to develop the optimization problem besides the software system meant to be optimized at runtime. The presented contract negotiation approaches instead do not require the developer to build the optimization problem for their software system, but to follow Model-Driven Software Development (MDSD) [121] and to provide quality contract templates in order to generate the optimization problems at runtime. Thus, the developer does not have to have knowledge about the respective optimization techniques. Instead common knowledge on MDSD and a general understanding of the non-functional behavior of the software system under development is required. As MDSD is part of general education for software engineers and each engineer should have a "feeling" for his code, this shift considerably lowers the burden to build self-optimizing software systems.

Merging Objective Functions by Behavior Simulation

A key challenge in QoS optimization is the comparability of NFPs, whenever optimization problems cover multiple objectives, each concerning another NFP. The question is how to compare the objective functions to each other? A well-known solution is the application of *utility theory*. That is, each objective function represents the utility gained by the respective NFP. Utility itself is expressed as a floating point number between zero and one and, hence, can be interpreted as 0 to 100% utility. The problem of this approach is founded in the notion of utility. In fact, different types of utility are intermingled. For example, the objectives to minimize energy consumption and to maximize performance cannot directly be compared to each other, because energy consumption and performance are incomparable NFPs. Wrapping both objective functions with utility functions requires in-depth knowledge by the developer. First, an explicit mapping of the energy consumption to a utility between 0 and 1 has to be defined. To do so, a deep understanding of the energy characteristics of the software system are required. A possible utility function would linearly map utilities from the minimum possible energy consumption (as 100% utility) to the maximum possible energy consumption (as 0% utility). The same has to be done for performance. But, what is the best or worst possible

performance? And, finally, how to compare user utility due to better performance with utility gained by lower costs? The only way is to directly ask the user for his priorities.

Instead of asking the user for priorities, another approach to achieve comparability between objective functions is *behavior simulation* as will be explained in the following. The principle idea is to translate between NFPs. For example, the NFP `cpu time` can be translated into energy consumed by using the CPU for the respective amount of time. The same can be done for memory usage by examining the consumed energy implied by the respective memory reads and writes.

> To merge objective functions concerning incomparable NFPs, the implied effect of these NFPs on a common NFP can be simulated by a DES.

Thus, a simulator able to translate a set of NFPs into a common NFP is required. Such a simulator needs detailed information about the input NFPs More concretely, the simulator can be realized as a DES and, hence, requires a timed sequence of annotated events as input. To simulate the effect of NFPs on a systems energy consumption, the behavioral models of the CCM are used (cf. Section 4.1.3). Namely, energy state charts (ESC) are used, which specify how energy is consumed. For the above example, an ESC of the CPU and an additional ESC of the memory are required. To translate `cpu time` into energy consumption, the CPU ESC is simulated for the respective amount of time. For memory usage, the memory ESC is used and the correlated timed sequence of memory read and write events is simulated.

Thus, in conclusion, the proposed approach to handle incomparable objectives is to translate them to a common objective function by translating each concerned NFP to a common NFP (e.g., their effect on energy consumption or response time) using a DES and behavioral models covering the common NFP.

6.3. Summary

In this chapter, a bridge between design principles (as shown in Chapter 4) and operation principles part of the following chapters is build. First, an approach to derive all valid system configurations using the models and artifacts from the development method has been shown to illustrate the connection between development artifacts and runtime system configurations. Next, the advancement from contract checking to ecomomic, multi-QoS contract negotiation systems has been outlined. The optimization approaches subject of the succeeding chapters all belong to the final class of systems. Additionally, a method to merge objectives covering different NFPs by simulation was presented as it is a prerequisite for the a priori MOO approaches[2].

[2] A priori approaches to MOO merge all objective functions to make use of single-objective optimization techniques (cf. Sect. 2.3.3).

Part III.

A Runtime Environment for Self-Optimizing Software Systems

7 Exact Approaches for Multi-Quality Auto Tuning

This chapter is based on the following publications:

- Sebastian Götz, Claas Wilke, Sebastian Cech and Uwe Aßmann: *Runtime Variability Management for Energy-efficient Software by Contract Negotiation.* In: Proceedings of 6th International Workshop on Models@run.time, ACM/IEEE (2011) 61–72
- Sebastian Götz, Claas Wilke, Sebastian Richly, Georg Püschel and Uwe Aßmann: *Model-driven Self-Optimization using Integer Linear Programming and Pseudo-Boolean Optimization.* To be published in Proceedings of Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE'13), IARIA XPS Press (2013).

Exact approaches to contract negotiation give a single, exact answer to the question, which system configuration is the best for a given user request. To do so, they base on optimization techniques, which guarantee an exact solution in comparison to approximate, near-optimal solutions. The need for exact contract negotiation is posed by critical systems, which do not allow for non-optimal solutions. For example, the field of embedded software often correlates with strict safety requirements. A typical example is given by software for airplanes or cars where a minimal remaining energy capacity has to be ensured for safe landing or deceleration. If a contract negotiation approach suggests non-optimal solutions these safety requirements are easily violated.

In this chapter, two approaches for exact contract negotiation will be presented: an ILP-based solution in Section 7.1 and a PBO-based solution in Section 7.2. Both approaches are compared with each other and evaluated in Section 7.3.

The contributions given in this chapter are the following:

- An exact runtime optimization approach using ILP. (Section 7.1)
- An exact runtime optimization approach using PBO. (Section 7.2)
- An analysis and comparison of both approaches. (Section 7.3)

7.1. Contract Negotiation by Integer Linear Programming

The aim of this section is to answer the question how to determine optimal system configurations of MQuAT systems using an exact optimization technique called integer linear programming. As described in detail in Chapter 6, a system configuration denotes a set of software component implementations deployed on component-containers, which run on servers (or, more general, computing entities). Thus, the question is, which implementations of which component types need to be mapped onto which containers in order to reach the optimal trade off between user satisfaction and execution costs.

To answer this question a variety of information is required, which is covered by the CCM models and QCL contracts as introduced in Chapter 4. Namely, variant models of hard- and software representing the currently running system, structure models of hard- and software representing the architecture of the system, QCL contracts characterizing the non-functional behavior of the software and a user request including the user's QoS demands.

Hardware Variant Models. The current hardware landscape needs to be considered: which containers are available, which resources do they provide, what are the values of their properties (e.g., the current frequency of a CPU), which state do the resources currently have (e.g., the resource's ACPI mode) and which control capabilities do these resources provide (e.g., switching off a core of a CPU or setting the operation mode of a wireless local area network (WLAN) router from "ad-hoc" to "access point").

Hard- and Software Structure Models. The structure (i.e., types) of the application and the infrastructure's resources need to be considered: which component types exist, which port types do they have, which possible compositions of component types (via their port types) exist and which NFPs are defined for the component types.

QoS Contracts The non-functional behavior of component implementations, which is encapsulated by QCL contracts, needs to be considered: which quality modes (utility levels) do the implementations provide, which effect do the implementations have w.r.t. the NFPs defined for the corresponding component types, which effect do the implementations imply on other components (non-functional dependencies), which resource requirements result from choosing a quality mode of the implementation and which transitions between quality modes are supported.

Software Variant Models. A variant model representing the current software landscape needs to be considered: which implementations are mapped onto which container, which quality mode do they currently operate in, which composition of component types has been chosen and what are the values of the component's NFPs.

User Requests. Finally, the user request including QoS demands needs to be considered: which functionality does the user intend to use, which data will be processed by the software, which minimum guarantees need to be assured to satisfy the user and which objectives are of importance to the user.

All this information needs to be processed in a structured way to derive the targeted optimal configuration. In this section, one structured approach is presented: solving an ILP, which comprises the aforementioned information. As ILP is a mathematical formalism with its own language, a transformation from the structure and variant models as well as the contracts and the request to ILP is required. Figure 7.1 depicts the general approach of this ILP generation, which can be characterized as a model-to-text transformation in terms of Model-Driven Architecture (MDA) [69].

On the left upper side of Fig. 7.1, the structure of the optimization problem formulated as an ILP is shown. An ILP comprises a set of objective functions, a set of decision variables and a set of constraints. The objective functions depend on the user request — which objectives are important for the user — and on the variant (i.e., runtime) model. Decision variables depend on QoS contracts and variant models, too. Finally, the constraints of the ILP depend on all available input information. In the following, the generation of decision variables, objective functions and constraints is discussed in more detail.

7.1.1. The Rational of Decision Variables

The choice of which information to encapsulate as decision variables of the ILP is affected by the effort required to translate an assignment to these variables into a system configuration to be realized by the control loop. In this work, the decision variables directly follow the characterization of system configurations: they denote which implementation is to be run in which quality mode on which container. This information is comprised by the name of the variable, whereas the type of the decision variables is of

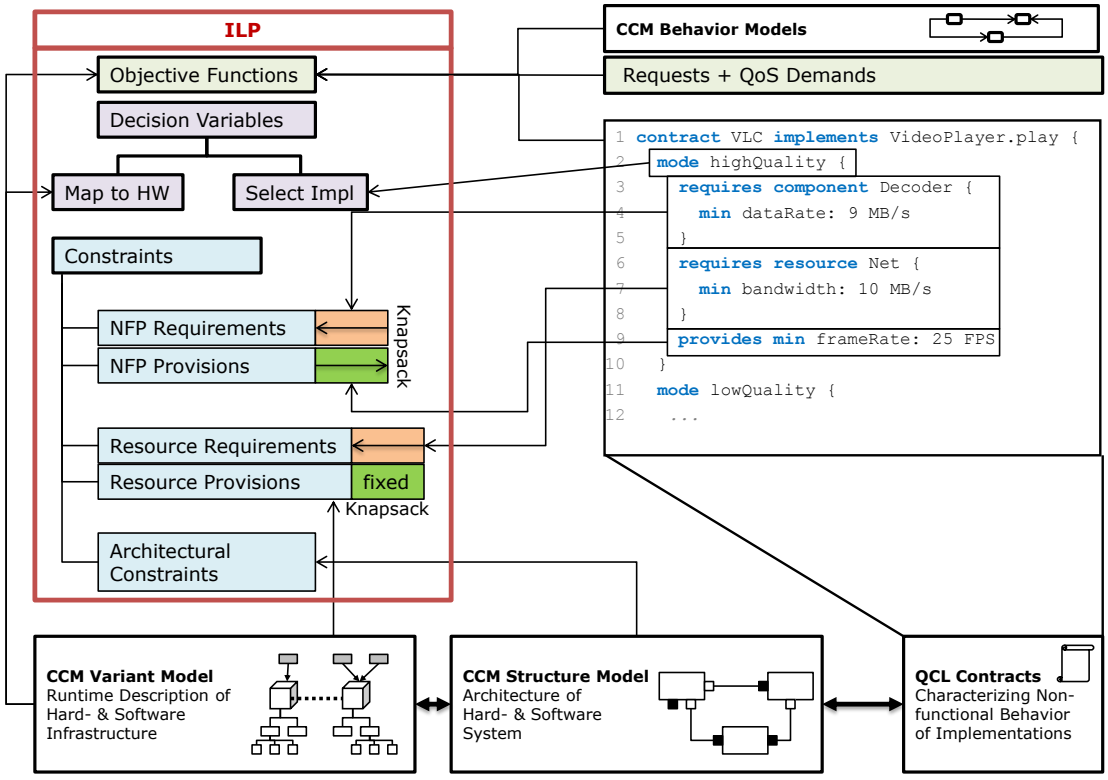


Figure 7.1.: Overview of ILP Generation.

boolean nature. Equation 7.1 shows the general form of decision variables as used in the presented approach. The symbol # in the name of the variables is used to concatenate identifiers.

$$b\#implementation\#mode\#container \in \mathbb{B} \tag{7.1}$$

In addition to these decision variables, additional variables are used in the ILP, which express the resource utilization and resulting NFP values implied by a certain mode (as specified in QCL contracts). These variables have a real value (i.e., are in $\mathbb{R}$). Equation 7.2 and 7.3 denote the general forms of these variables.

$$usage\#container\#resource\#subresource\#...\#NFP \in \mathbb{R} \tag{7.2}$$

$$implementation\#NFP \in \mathbb{R} \tag{7.3}$$

Solving an ILP, which works on these variables, leads to an assignment of values to these variables, which represent the optimal system configuration. Thus, the ILP

solution does not just provide information about the optimal selection of implementations and modes as well as their mapping to containers, but additionally information on the resulting NFPs of the participating components. The variables are used in objective functions as well as in the constraints as will be shown in the following subsections.

7.1.2. Generation of Objective Functions

All objective functions in the context of this work base on assessment functions of system configurations. That is, an objective function assesses system configurations in terms of the respective objective (e.g., energy, performance or reliability) and aims to determine that configuration, which is assessed to be minimal or maximal w.r.t. the current objective. Users specify the objectives (i.e., maximization or minimization of NFPs), which are of relevance to them. For each specified NFP, a respective objective function is derived.

A straightforward objective function based on the variables explained in the previous subsection is resource minimization as shown in Equation 7.4. To avoid the naïve solution to not use any resources, the constraints of the ILP are used. But, still it is obvious that this objective function does not lead to the intended result (i.e., minimum resource consumption), because the units and, in general, the semantics of each resource usage variable are not considered. For example, the formula does not differentiate between utilizing 10 MB of main memory in contrast to utilizing 10 MB of hard disk drive (HDD) space. The information how these two properties differ from each other is contained in the CCM structure model, but is lost by this translation to an objective function.

$$\min : \sum usage\#container\#resource\#subresource\#...\#NFP \tag{7.4}$$

Thus, more sophisticated objective functions are required. A typical approach is the application of utility theory to map each variable to a utility expressed as a real value between zero and one. In the case of resource usage the utility functions can reflect the difference between using space of main memory and an HDD by putting the requested amount of space in relation to the totally available space. Nevertheless, further characteristics of NFPs as discussed in Section 2.1.3 like perishability (i.e., volatility) cannot be considered by utility functions. For example, the difference between a volatile NFP like bandwidth and a non-volatile NFP like remaining battery capacity cannot be reflected by a utility function applied to both NFPs, because there is no common meaning of utility for both NFPs. That is the meaning of utility needs to include the opportunity to *save* the resource for later use (i.e., permanence), which is meaningful for battery capacity or main memory space, but not for NFPs like bandwidth or throughput. To address this problem, the different classes of NFPs, according to their characteristics as described in Sect. 2.1.3, have to be handled by separate objective functions, which poses the need for a posteriori multi-objective optimization approaches (cf. Sect. 2.3.3). The

general form of objective functions according to utility theory is shown in Equation 7.5. The objective is to maximize the overall utility.

$$\max : \sum utility(decisionvariable) \tag{7.5}$$

An even more sophisticated objective is the combination of the previous two types of objectives (i.e., cost minimization and utility maximization): *efficiency* maximization. The general form of this type of objective is depicted in Equation 7.6. Here η denotes *efficiency*. Again the problem of comparability of utilities and, in addition, of costs implied by decisions arises. Notably, a common meaning for all costs and another common meaning for all utilities suffices for a meaningful objective. This can be achieved by focusing on selected qualities like performance, energy or reliability.

$$\max : \eta(decisionvariable) = \frac{utility(decisionvariable)}{cost(decisionvariable)} \tag{7.6}$$

For example, the focus on energy allows to express all costs in terms of implied energy consumption. There is no problem of incomparability, because every decision is investigated only by its effect on energy consumption. A possible objective function, aiming to maximize the energy efficiency of the overall system is depicted in Equation 7.7.

$$\max : \eta_{energy}(decisionvariable) = \frac{utility(decisionvariable)}{energy(decisionvariable)} \tag{7.7}$$

Objectives of the same kind can be retrieved for other qualities like reliability or performance. To do so, these qualities need to be interpreted as costs. Energy consumption is a cost by nature, but performance and reliability are not. Nevertheless, reduction of performance or reliability (i.e., negating positive qualities) models costs and, hence, allows to optimize the efficiency of the system instead of solely maximizing the utility. It is easy to see that utility and cost can be modeled inversely, i.e., cost can be modeled as utility and vice versa. The only requirement is to achieve comparability for costs and utilities. In other words, it is possible to combine as many qualities as can be mapped to two common meanings for comparability (one for costs and one for utilities). But, usually, there is no common meaning between different qualities. Reliability and performance are both positive qualities, but a gain in reliability is incomparable to a gain in performance. To compare reliability and performance, only the end user is able to provide the required information: the relative importance of reliability over performance and vice versa.

Thus, a single objective function usually allows for the optimization of either **one quality** or the **trade off between two qualities**. Notably, comparable qualities can be combined and, hence, be handled as a single quality. In addition, instead of qualities, their utilities can be used to formulate the objective function.

Unfortunately, as described in Section 2.3.1, the ILP formalism only allows for a single objective function. Using the ILP solution, hence, only allows for the optimization of, e.g., performance relative to energy consumption *or* reliability relative to energy consumption, but does not allow to optimize the trade off between all three qualities. In Chapter 8 approaches supporting multiple objectives will be presented.

The Effect of Reconfiguration and Decision Making. Finally, an important aspect of objective functions in the context of reconfigurable systems is the need to consider the **reconfiguration** and **decision making** itself. This is, because both processes require time and resources and, thus, affect the objective functions. In consequence, it does not suffice to assess a system configuration in terms of how it processes a request, but the effect of decision making and the reconfiguration from the current system configuration need to be considered, too. A general objective function should, hence, look as depicted in Equation 7.8, which aims for maximum efficiency. Notably, the costs implied by reconfiguration and decision making need to be assessed w.r.t. the quality of interest for the respective objective function. Thus, an exemplary incarnation of Equation 7.8, which focuses on performance relative to energy consumption, requires the assessment of reconfiguration and decision making costs in terms of required energy as well as the negative utility of implied performance losses.

$$\max : \sum_{i=1}^{n} \frac{utility_i}{cost_i} * \frac{utility_{reconfiguration} + utility_{decisionmaking}}{cost_{reconfiguration} + cost_{decisionmaking}} \tag{7.8}$$

The assessment of $cost_{reconfiguration}$ as well as $cost_{decisionmaking}$ (and the negative utilities) is a non-trivial task. For reconfiguration, the question is how long does it take to reconfigure the system from its current configuration to another given configuration (or how much energy will be consumed by reconfiguration, etc.). The same question arises for decision making. The problem is that both reconfiguration as well as decision making are usually hard to predict. For example, the time required to migrate a component from one server to another can vary depending on the currently available network bandwidth between the servers. Nevertheless, it is possible to assess reconfiguration operations in terms of their context-dependent non-functional behavior in the same way as implementations of component types are assessed. Indeed, in MQuAT, reconfiguration operations are handled like usual software component implementations. The same holds for the decision making, which is a component with an ILP implementation, a PBO implementation, an approximate implementation (cf. Chapter 8) or a MOO implementation (cf. Chapter 9).

7.1.3. Constraint Generation

To narrow the search space, a set of constraints can be generated based on the knowledge about the software structure, the available hardware, the resource requirements and the interdependencies between NFPs. In general, three classes of constraints are generated in the presented ILP approach: constraints negotiating software NFPs, constraints negotiating resource requirements and architectural constraints. In the following, each class will be explained in more detail.

Software NFP Negotiation

The negotiation of software NFPs covers the interdependencies between NFPs of different software components expressed by their provisions and requirements as stated in QCL contracts. The selection of an implementation of a software component type in a certain quality mode induces a set of NFP provisions (by the implementation) and requirements (to other components). To determine an optimal selection includes to find a balance between provided NFPs and required NFPs across all required components to fulfill the user's request. In consequence, this part of the optimization problem can be seen as a dynamic Knapsack problem in the sense that the Knapsack is growing for each additional NFP provision. The goal is to find a selection of implementations, which does not violate any NFP requirements.

This problem is reflected in the ILP by two types of constraint clauses, which are generated for each NFP. First, NFP provisions are expressed as an equality constraint as depicted in Equation 7.9. Depending on the assignment of the decision variables, the available amount of the respective NFP results. The variables a,b,c and i enumerate all implementations (a), quality modes (b), containers (c) and NFPs (i). They will be used with the same semantics for the remainder of this section.

$$\begin{aligned} NFP_i &= \sum^{c} prov_a^b * b\#implementation_a\#mode_b\#container_c \\ a,b,c,i &\in \mathbb{Z} \end{aligned} \tag{7.9}$$

Second, the NFP requirements implied by selecting a certain implementation in a quality mode are expressed as inequality constraint as depicted in Equation 7.10. Notably, the relation between NFP_i and the aggregated NFP requirements is only "less or equal", if the respective NFP is of ascending order. For example, the NFP `memory` is of ascending order (greater values are better), whereas the NFP `response_time` is of descending order. Thus, for `response_time`, the inequality is of "greater or equal than" type.

$$\begin{array}{rcl} NFP_i & <= & \sum^{c} req_a^b * b\#implementation_a\#mode_b\#container_c \\ a,b,c,i & \in & \mathbb{Z} \end{array} \tag{7.10}$$

Resource Negotiation

Besides provisions and requirements of NFPs, the selection of implementations implies resource requirements, too. In contrast to software NFPs, the provision of resources is fixed by the available hardware[1]. Thus, resource negotiation can be seen as a classical Knapsack problem, where the provision of resources denotes the size of the Knapsack and the selection of implementations the packing of it.

Similar to software NFP negotiation, two types of constraints are generated for resource negotiation. First, Equation 7.11 depicts the provision of resources.

$$\begin{array}{rcl} ResourceProperty_i & >= & 0 \\ ResourceProperty_i & <= & maximum_i \\ ResourceProperty_i & = & granularity_i * x \\ x & \in & \mathbb{B} \\ i & \in & \mathbb{Z} \end{array} \tag{7.11}$$

The provided property naturally needs to be greater or equal than zero and less or equal than the maximum offered by the resource. In addition, the granularity of the resource can be restricted. For example, the amount of disk space can only be utilized in blocks of a certain size (e.g., 4KB). In the constraints of Equation 7.11, the terms *maximum* and *granularity* are replaced by the respective concrete values. The term x remains a free variable, which is to be found by the solver of the optimization problem. The restriction of x to be binary (i.e., $\in \mathbb{B}$) enforces the restriction of the resource property to be a multiple of *granularity*.

The second type of constraint covers the resource requirements by selecting an implementation. The resulting constraint is depicted in the following equation.

$$\begin{array}{rcl} ResourceProperty_i & <= & \sum^{c} req_a^b * b\#implementation_a\#mode_b\#container_c \\ a,b,c,i & \in & \mathbb{Z} \end{array} \tag{7.12}$$

For each resource property constraints of these types are generated, whereby the search for valid assignments to the decision variables is further restricted.

[1] We do not consider adaptive hardware here.

Architectural Constraints

Finally, constraints based on the knowledge about the software's architecture and the requested software component are translated into constraints of the ILP. The simplest possible constraint of this type is the necessity to select exactly one implementation of a component type whose port is requested by the user. The corresponding constraint is depicted in the following equation.

$$\begin{aligned} \sum b\#implementation_a\#mode_b\#container_c = 1 \\ \forall a \in T \wedge b \in modes\_of(a) \\ a, b, c \in \mathbb{Z} \end{aligned} \tag{7.13}$$

For all modes b of all implementations available for the component type T the sum of the corresponding decision variables needs to be exactly one. This constraint suffices, if no other component types exist or the requested component type does not use any other component type. If another component type is used, the need to select an implementation of this type needs to be expressed, too. In the general case, constraints have to be generated, which express that the selection of an implementation of component type T_1 implies the need to select an implementation of type T_2. The following equation depicts this kind of constraint.

$$\begin{aligned} \sum b\#impl._a\#mode_b\#container_c &= \sum b\#impl._d\#mode_e\#container_f \\ \forall a \in T_1 \wedge b \in modes\_of(a) \wedge d \in T_2 &\wedge e \in modes\_of(d) \wedge depends(T_1, T_2) \\ a, b, c, d, e, f &\in \mathbb{Z} \end{aligned} \tag{7.14}$$

7.1.4. ILP Generation by Example

In the following, a small example is used to depict the generation of an ILP according to the abstract description given above. The example comprises three component types: a list generator (`ListGen`), a sorter (`Sort`) and a filter (`Filter`). The component types are interrelated as depicted in Figure 7.2.

The `ListGen` component offers a provided port `getList`, which takes the size of the list as a parameter. Its non-functional behavior is fully determined by this parameter. For the sake of simplicity only one implementation exists: a random generator (`RandomGen`), which generates list values using a random number generator. The `Sort` component has a required port to retrieve an input list (called `setList`; not shown in Figure 7.2 for clarity) and a provided port `getSortedList`. The non-functional behavior of this port is determined by the size of the list held by the `Sort` component (which has to be set previously using `setList`). Two implementations of `Sort` exist: a `QuickSort` and a

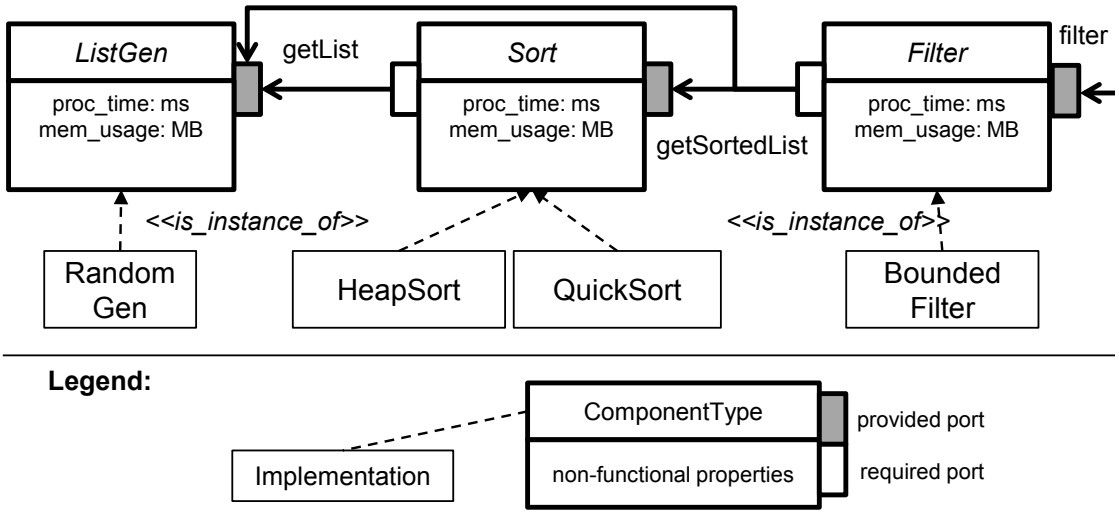


Figure 7.2.: List Generation, Sorting and Filter Example.

HeapSort implementation. Finally, the Filter component has a required port setList (again not shown for clarity) and a provided port filter, which takes an Object as argument, which is to be refined by the Filter implementations. The non-functional behavior depends on this argument and the size of the list, which is held by the Filter component. Only a single implementation of Filter exists: a BoundedFilter, which specifies an upper and lower bound for the integer values of the list. This implementation will delete all values which are above or below the given threshold. The port connectors in Figure 7.2 show that both, Filter and Sort can use the provided getList port of ListGen to retrieve a list of a certain size. In addition, Filter can use the getSorted-List port provided by the Sort component.

In the following, the contracts for each of the four implementations, computed on a single server, will be shown. Listing 7.2 depicts the contract for the RandomGen implementation. The contracts for the QuickSort and HeapSort implementation are shown in Listing 7.3. Finally, the contract for the BoundedFilter implementation is shown in

```
1 import ccm [./sortfilter.structure]
2 target platform [./current.variant]
3
4 call Filter.filter expecting {
5         list_size = 200000
6         energy_consumption minimize
7 }
```

Listing 7.1: User Request on Filter Component.

```
1  contract Random implements software ListGen.getList {
2    mode fast {
3      requires resource CPU {
4        frequency min: 300
5        cpu_time max: f( list_size ) = 6.84444433222476 + 2.19333334170516E-4 * ( list_size )
6      }
7      provides response_time min: f( list_size ) = 5.33921568506459E-4 * ( list_size )
8    }
9    mode slow {
10     requires resource CPU {
11       frequency min: 100
12       cpu_time max: f( list_size ) = 3.46666683611941 + 2.41999998870316E-4 * ( list_size )
13     }
14     provides response_time min: f( list_size ) = 5.38039215642729E-4 * ( list_size )
15   }
16 }
```

Listing 7.2: Contract for "Random" Implementation of List Generator Component.

Listing 7.4. Please note that the sort and filter implementations include requirements to the `ListGen` and `Sort` component, respectively. `QuickSort.immediate` and `BoundedFilter.slow` do not define any NFP requirement whereas all other modes require a response time of at most 200 milliseconds.

For the user request shown in Listing 7.1 on `Filter.filter`, with a list size of 200.000 elements and no upper and lower bounds against a single server[2], the ILP depicted in Listing 7.5 results.

The objective function is of particular interest as it only uses weighted decision variables. The weights represent the effect on CPU energy consumption by taking the decision embodied by the decision variable. The values are computed based on the `cpu_time` as defined in the contracts. For example, the weight of `b#Quicksort#delayed#R1` is computed by evaluating the `cpu_time` expressed in the `Quicksort` contract for container `R1` in the `delayed` mode w.r.t. the user request leading to a value of approx. 191.57 ms. This time is used by the simulator, which simulates the energy consumption of a CPU over time, resulting in a value of 5.700,0 mJ. In contrast, the `immediate` mode leads to almost twice this consumption (10.285,0 mJ).

The architectural constraints express the requirement to choose exactly one implementation per component type. For resource negotiation only the constraints for the CPU frequency are generated, because in the contracts there are only requirements specified against this resource. Software NFP negotiation is realized by six constraints: three expressing the upper limit of the NFPs, two expressing the NFP requirements and one expressing the users requirement against an NFP. Finally, the decision variables are restricted to be of boolean nature.

[2]CPU: Intel Core i7 720QM quad core 1.6 GHz, 8 GB RAM, SATA2 HDD

```
1  contract Quicksort implements software Sort.sort {
2    mode immediate {
3      requires component ListGen {        }
4      requires resource CPU {
5        frequency min: 300
6        cpu_time max: f( list_size ) = 4.08333333923283 + 9.2333333335106E-4 * ( list_size )
7      }
8      provides response_time min: f( list_size ) = 0.00188657142854989 * ( list_size )
9    }
10   mode delayed {
11     requires component ListGen {
12       response_time max: 50
13     }
14     requires resource CPU {
15       frequency min: 100
16       cpu_time max: f( list_size ) = 3.00000011731915 + 9.42857142122537E-4 * ( list_size )
17     }
18     provides response_time min: f( list_size ) =  0.00191028571412205 * ( list_size )
19   }
20 }
21
22 contract HeapSort implements software Sort.sort {
23   mode immediate {
24     requires component ListGen {
25       response_time max: 50
26     }
27     requires resource CPU {
28       frequency min: 300
29       cpu_time max: f( list_size ) = 4.35134128299379E-25 * ( list_size ^ 4 )
30                                    + 113.591832484444 * ( list_size ^ 5 )
31     }
32     provides response_time min: f( list_size ) = 0.00199885714285247 * ( list_size )
33   }
34   mode delayed {
35     requires component ListGen {
36       response_time max: 50
37     }
38     requires resource CPU {
39       frequency min: 100
40       cpu_time max: f( list_size ) = 9.99142857072614E-4 * ( list_size )
41     }
42     provides response_time min: f( list_size ) = 0.00208171428594746 * ( list_size )
43   }
44 }
```

Listing 7.3: Contract for "QuickSort" and "HeapSort" Implementation of Sort Component.

7.2. Contract Negotiation by Pseudo-Boolean Optimization

The key aspect of the configuration problem to be solved by the generated optimization problem is denoted by boolean decision variables, which encompass the decision to select certain implementations and the decision to map these implementations to a certain re-

```
1  contract BoundedFilter implements software Filter.filter {
2    mode fast {
3      requires component Sort {
4        response_time max: 50
5      }
6      requires resource CPU {
7        frequency min: 300
8        cpu_time max: f( list_size ) = 4.3590927186085E-5
9                                     + 5.30301025333666E-12 * ( list_size )
10                                    + 8.87272483899978 * ( list_size ^ 5 )
11     }
12     provides response_time min: f( list_size ) = 0.0 + 1.42035087763293E-4 * ( list_size )
13   }
14   mode slow {
15     requires component Sort {   }
16     requires resource CPU {
17       frequency min: 100
18       cpu_time max: f( list_size ) = 0.945455021501056 + 5.74545446684514E-5 * ( list_size )
19     }
20     provides response_time min: f( list_size ) = 0.418188728480244
21                                                + 1.1751514069185E-4 * ( list_size )
22   }
23 }
```

Listing 7.4: Contract for "BoundedFilter" Implementation of Filter Component.

source. The remaining variable types used in the ILP solution presented in the previous chapter comprise resource usage and resulting NFP values. In this section the possibility to omit non-boolean variables will be shown. The intended goal is to apply more efficient solving techniques to the generated optimization problems, which leverage on the restriction to use boolean variables only. Namely, 0-1 ILPs result, which can be handled by PBO allowing for the application of polynomial complexity algorithms known from solving satisfiability problems in propositional logics (e.g., Davis-Putnam approach [42] or DPLL [41]).

7.2.1. Reformulation of the Configuration Problem in PBO

To apply techniques of PBO to the configuration problem to be generated, all non-boolean variables need to be expressed in a different way in the PBO to be generated. Namely, a new way to express resource negotiation, NFP negotiation including user requirements and the objective function is required. Notably, the architectural constraints defined for the ILP solution can remain unchanged, because they only refer to decision variables. In the following, a solution for each of the constraints subject to adjustment is given.

```
/* objective function: minimize energy consumption (based on cpu\_time) */
min: 5700.0 b#Quicksort#delayed#R1 + 495.0 b#UnsortedFilter#slow#R1
+ 10285.0 b#Quicksort#immediate#R1 + 6160.0 b#Javasort#immediate#R1
+ 385.0 b#UnsortedFilter#fast#R1 + 2250.0 b#Random#slow#R1
+ 5940.0 b#Javasort#delayed#R1 + 2695.0 b#Random#fast#R1;

/* architectural constraints */
b#Random#fast#R1 + b#Random#slow#R1 = b#Quicksort#delayed#R1 + b#Quicksort#immediate#R1
                                    + b#Javasort#immediate#R1 + b#Javasort#delayed#R1;
b#UnsortedFilter#fast#R1 + b#UnsortedFilter#slow#R1 = 1;
b#Quicksort#immediate#R1 + b#Quicksort#delayed#R1
+ b#Javasort#immediate#R1 + b#Javasort#delayed#R1 = b#UnsortedFilter#slow#R1
                                                  + b#UnsortedFilter#fast#R1;

/* resource negotiation */
usage#R1#Core[TM]_i7_CPU_Q_720_@_1.60GHz#frequency <= 1596.0;
usage#R1#Core[TM]_i7_CPU_Q_720_@_1.60GHz#frequency >= 0;
usage#R1#Core[TM]_i7_CPU_Q_720_@_1.60GHz#frequency =
  100 b#Javasort#delayed#R1 + 100 b#UnsortedFilter#slow#R1 + 100 b#Quicksort#delayed#R1
+ 300 b#Random#fast#R1 + 300 b#Quicksort#immediate#R1 + 100 b#Random#slow#R1
+ 300 b#Javasort#immediate#R1 + 300 b#UnsortedFilter#fast#R1;

/* software NFP negotiation */
Sort#response_time = 382.05714282441 b#Quicksort#delayed#R1
                   + 377.31428570997804 b#Quicksort#immediate#R1
                   + 399.771428570494 b#Javasort#immediate#R1
                   + 416.34285718949195 b#Javasort#delayed#R1;
Filter#response_time = 23.921216866850248 b#UnsortedFilter#slow#R1
                     + 28.407017552658598 b#UnsortedFilter#fast#R1;
ListGen#response_time = 107.6078431285458 b#Random#slow#R1
                      + 106.7843137012918 b#Random#fast#R1;
Sort#response_time >= 50 b#UnsortedFilter#fast#R1;
ListGen#response_time >= 50 b#Quicksort#delayed#R1 + 50 b#Javasort#immediate#R1
                                                  + 50 b#Javasort#delayed#R1;
/* user request */
Filter#response_time <= 200.0;

/* boolean restriction */
binary b#Quicksort#delayed#R1, b#UnsortedFilter#slow#R1, b#Quicksort#immediate#R1,
b#Javasort#immediate#R1, b#UnsortedFilter#fast#R1, b#Random#slow#R1,
b#Javasort#delayed#R1, b#Random#fast#R1;
```

Listing 7.5: Generated ILP for List Generator/Sort/Filter Example.

Resource Negotiation without Usage Variables

The expression of resource negotiation with usage variables has been shown in the previous section. For better readability find a repetition of the generation rule here:

$$\begin{aligned} ResourceProperty_i &>= 0 \\ ResourceProperty_i &<= maximum \\ ResourceProperty_i &= granularity * x \\ ResourceProperty_i &<= \sum {}^{c} req_a^b * b\#implementation_a\#mode_b\#container \\ x &\in \mathbb{B} \end{aligned} \tag{7.15}$$

The variables a, b and c enumerate all implementations (a), quality modes (b) and containers (c). These variables will be used with the same semantics for the remained of this section. The constraints shown above, except for the granularity restriction, can be expressed by a single PBO constraint, which implicitly represents the respective resource property (i.e., $ResourceProperty_i$) as shown in the following equation.

$$0 + \sum {}^{c} req_a^b * b\#implementation_a\#mode_b\#container \leq maximum \tag{7.16}$$

The omission of the explicit resource usage variable consequently prohibits its usage in other constraints. But, fortunately, there are no other constraints which rely on resource usage.

NFP negotiation

Alongside with the restriction of resource usage, the dependencies between offered and required NFPs has to be expressed by constraints. In the ILP solution, explicit variables for each NFP have been used to connect separate constraints for their provisions and requirements. The same principle, as for resource usage negotiation can be applied for NFP negotiation, too: the implicit expression of each NFP variable. The following equations repeat the expression of NFP provision and requirement constraints in the ILP solution to ease the comparison to the implicit NFP constraints for PBO.

$$\begin{aligned} NFP_i &= \sum {}^{c} prov_a^b * b\#implementation_a\#mode_b\#container_c \\ NFP_i &<= \sum {}^{c} req_a^b * b\#implementation_a\#mode_b\#container_c \end{aligned} \tag{7.17}$$

The first constraint specifies the upper limit of the respective NFP, which, in comparison to resource usage negotiation, is the *maximum* constant. The second constraint denotes the usage of the respective NFP, which is analogously to the resource usage constraint.

For PBO, both constraints are merged into a single constraint which implicitly represents the possible expressions of the NFP. It is shown below:

$$\sum {}^{c}req_{a}^{b} * b\#implementation_{a}\#mode_{b}\#container_{c} \leq \sum {}^{c}prov_{a}^{b} * b\#implementation_{a}\#mode_{b}\#container_{c} \tag{7.18}$$

For the generation of these constraints the user request needs to be considered, too. This can be accomplished straightforward by incorporating the respective NFP requirements expressed by the user in his request as minimum value of the respective NFP. As for each NFP there is a constraint as shown above, the PBO constraint for an NFP with associated user requirement req_{user} is denoted as follows:

$$\sum req_{user} + {}^{c}\, req_{a}^{b} * b\#implementation_{a}\#mode_{b}\#container_{c} \leq \sum {}^{c}prov_{a}^{b} * b\#implementation_{a}\#mode_{b}\#container_{c} \tag{7.19}$$

Thus, NFP negotiation is basically the more general form of resource usage negotiation as it allows for a variable upper bound instead of a fixed maximum value.

Reformulation of Objective Function

The objective function in PBO can only rely on the decision variables, because only these variables exist. In contrast, the ILP solution allowed for the application of resource usage and NFP variables. In other words, the minimization or maximization of resource usage and NFPs can be directly expressed in an ILP but not in PBO.

The general form of the objective function in PBO is shown below:

$$min : \sum weight * b\#implementation_{a}\#mode_{b}\#container_{c} \tag{7.20}$$

Thus, the major issue to generate meaningful objective functions in PBO is the computation of the *weight* constants for each decision variable. This *weight* has to express the impact the decision represented by the decision variable on the overall objective. For example, if the objective is to minimize the response time of a feature invocation, the weights have to reflect the impact of using a certain implementation in a given mode on a container w.r.t. to the response time resulting from this decision in comparison to alternative decisions. Notably, there might be decisions which do not affect the response time. The respective variables can be weighted zero or, in other terms, these variables can be excluded from the objective function. For all remaining variables the relative impact has to be computed. The example objective function presented in the last Section (cf. Listing 7.5) shows how CPU energy consumption can be used as weight for decision variables.

7.2.2. PBO Generation by Example

To showcase the PBO generation approach, the same example as for the ILP approach as depicted in Figure 7.2 is used. By this, a direct comparison of the resulting generated optimization problems is possible.

The example comprised three components: a `Filter`, a `Sort` and a `ListGen` component. There was one `Filter` implementation, two `Sort` implementations and one `ListGen` implementation. The respective contracts have been shown in Listings 7.2, 7.3 and 7.4.

The resulting PBO problem is shown in Listing 7.6. The objective function stays unchanged, as it relies on decision variables only. As mentioned before, there is no need to adjust the architectural constraints from the ILP solution, too. Hence, also these constraints stay unchanged. The resource usage negotiation constraints are merged into implicit resource usage constraints. Thus, for each pair of three constraints in the ILP solution only one constraint results in the PBO solution. For software NFP negotiation pairs of provision and requirement constraints are merged into implicit NFP constraints. In addition, the user requirement constraint is merged into the corresponding NFP negotiation constraint.

In comparison to the ILP solution, only 7 instead of 12 constraints are generated. As all non-boolean variables are omitted in the PBO solution, less variables are generated, too (8 instead of 12). Thus, the PBO is smaller than the ILP and, hence, is likely to be solved faster than the ILP. In the next section the performance of the ILP solution is compared to the performance of the PBO solution.

7.3. Scalability Evaluation

The above described approach to generate and solve an ILP or PBO based on runtime models of the system subject to optimization is *not likely to scale*. This is, because solving an ILP or PBO is known to be an NP-hard problem, where the processing time required by the solver grows exponentially with number of decision variables of the ILP/PBO. In the following, I will show how ILP/PBO generation and solving performs and show that both approaches are feasible for a certain class of applications (although it is obviously not feasible in general). I will compare the performance of both approaches, showing that the ILP solution outperforms the PBO solution.

7.3.1. Generation of Test Systems for Empirical Evaluation

To empirically evaluate the performance of the exact contract negotiation approaches a set of systems—subject to optimization—needs to be generated. As the ILP/PBO generation only relies on the models of the system (and not the system itself), it is possible to evaluate the approaches against a variety of system types without the need

```
/* objective function: minimize energy consumption (based on cpu\_time) */
min: 5700.0 b#Quicksort#delayed#R1 + 495.0 b#UnsortedFilter#slow#R1
+ 10285.0 b#Quicksort#immediate#R1 + 6160.0 b#Javasort#immediate#R1
+ 385.0 b#UnsortedFilter#fast#R1 + 2250.0 b#Random#slow#R1
+ 5940.0 b#Javasort#delayed#R1 + 2695.0 b#Random#fast#R1;

/* architectural constraints */
b#Random#fast#R1 + b#Random#slow#R1 = b#Quicksort#delayed#R1 + b#Quicksort#immediate#R1
                                    + b#Javasort#immediate#R1 + b#Javasort#delayed#R1;

b#UnsortedFilter#fast#R1 + b#UnsortedFilter#slow#R1 = 1;

b#Quicksort#immediate#R1 + b#Quicksort#delayed#R1
+ b#Javasort#immediate#R1 + b#Javasort#delayed#R1 = b#UnsortedFilter#slow#R1
                                                  + b#UnsortedFilter#fast#R1;

/* resource negotiation cpu_frequency */
  100 b#Javasort#delayed#R1 + 100 b#UnsortedFilter#slow#R1 + 100 b#Quicksort#delayed#R1
+ 300 b#Random#fast#R1 + 300 b#Quicksort#immediate#R1 + 100 b#Random#slow#R1
+ 300 b#Javasort#immediate#R1 + 300 b#UnsortedFilter#fast#R1 <= 1596.0;

/* software NFP negotiation */
/* Sort#response_time */
50 b#UnsortedFilter#fast#R1 <= 382.05714282441 b#Quicksort#delayed#R1
                  + 377.31428570997804 b#Quicksort#immediate#R1
                  + 399.771428570494 b#Javasort#immediate#R1
                  + 416.34285718949195 b#Javasort#delayed#R1;

/* Filter#response_time */
200.0 >= 23.921216866850248 b#UnsortedFilter#slow#R1
                    + 28.407017552658598 b#UnsortedFilter#fast#R1;

/* ListGen#response_time */
50 b#Quicksort#delayed#R1 + 50 b#Javasort#immediate#R1
                          + 50 b#Javasort#delayed#R1
                    <= 107.6078431285458 b#Random#slow#R1
                     + 106.7843137012918 b#Random#fast#R1;
```

Listing 7.6: Generated PBO for List Generator/Sort/Filter Example.

for their physical presence. Thus, a parametrizable system generator is required, which is capable of generating models as usually derived by the runtime environment. This includes hard- and software structure models, hardware and software variant models and QoS contracts.

The following parameters are exposed by the system generator and will be explained in more detail in the following paragraphs.

Hardware Variant Model.

- amount of servers S,

- the number of resources per server N_{res},
- the number of properties per resource $N_{resProp}$.

Software Structure Model.

- number of component types, C
- the average number of required and provided port types per component type,
- the maximum length of a chain of components and
- the number of NFPs defined per component type N_{nfp}.

QCL Contracts.

- number of implementations per component type N_{impl},
- number of modes per implementation N_{mode},
- number of hardware requirements per mode,
- number of NFPs required per software dependency per mode and
- number of provided NFPs per mode.

Hardware Variant Model. For the scalability analysis all generated hardware models represent a server landscape, which adheres to a fixed hardware structure model as depicted in Figure 7.3. In general, other types of hardware are supported by the approach (e.g., humanoid robots or mobile phones). This model specifies a hardware infrastructure to comprise interconnected servers, which in turn consist of at least one CPU, a random access memory (RAM) module, an HDD and a network device. A CPU is characterized by its current load in percent, its frequency and number of cores. A RAM module is characterized by its total size, the amount of free and used memory and the read and write throughput. The same holds for the HDD. A distinction between RAM and HDD, although they expose the same NFPs, is required because software implementations need to explicitly use an HDD and implicitly use RAM. The generation of hardware variant models, hence, allows to vary the amount of servers and the number and properties of CPUs, RAM modules, HDD and network devices.

Software Structure Model. Software structure models describe a set of component types, port types, and relations in terms of port connector types. Hence, any form of directed multigraph (i.e., with multiple links between the same nodes) can be expressed. The question is, in consequence, which types of software structure models potentially lead to different (non-functional) behavior of the ILP/PBO approach and, hence, are worth to generate for investigation. To answer the first question, the consequences of

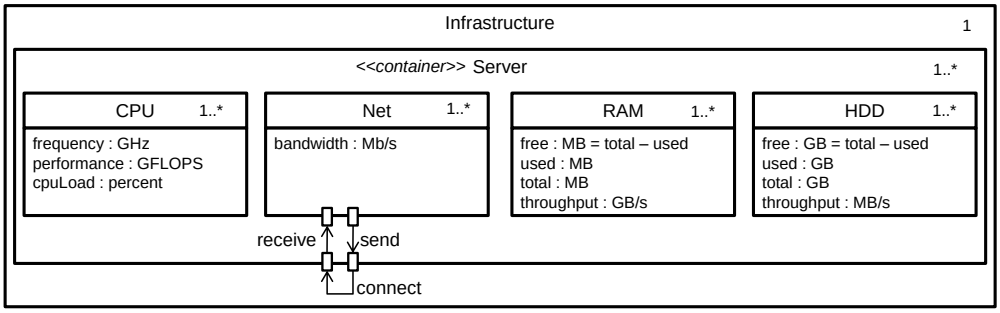


Figure 7.3.: Structural Model Specifying Types of a Server Landscape.

a certain structure need to be investigated. The number of decision variables is not affected by the structure, but only by the number of component types (and servers). However, the number of constraints depends on the structure. This includes software NFP negotiation constraints as well as architectural constraints. The more outgoing links (number of provided ports) a component type has, the more architectural constraints are required: one for each provided port type. The amount of constraints for software NFP negotiation depends on the number of NFP dependencies defined in the QoS contracts of the participating component types and, hence, is implicitly affected by the number of interconnections between component types. If, for example, a certain functionality can be provided to a component type A by two component types, B and C, and A requires a certain NFP for this functionality, the number of software NFP negotiation constraints doubles, because the constraints have to be specified for A using B and A using C. The number of constraints grows even further if B or C have dependencies to further component types themselves. That is the depth of dependency chains influences the number of constraints, too. For the system generator, the valuable parameters for generation are the number of component types C, the average number of required and provided port types per component type, the maximum depth of dependency chains δ and the number of NFPs defined per component type N_{nfp}.

QCL Contracts. The implementations of software component types are covered by QCL contracts, which expose the non-functional behavior of the implementation they represent. An obvious parameter is of course the average number of implementations (and, hence, contracts) per component type. As QCL contracts allow to express multiple quality modes (reflecting levels of user utility as explained in Section 4.2), the average number of modes per contract is another important parameter for the system generator. In addition, the amount of clauses expressing NFP provisions and requirements, as well as resource requirements directly influences the amount of constraints to be generated.

Feasibility of Generated Systems. For proper evaluation results using generated systems an important characteristic of these systems has to be ensured: feasibility. At least one valid configuration for a given request has to exist, because both–ILP and PBO–solvers are fast in determining the nonexistence of a solution and, thereby, hide their actual behavior, which is subject to evaluation. Randomly generating NFP provisions and requirements obviously lead to infeasible systems in most cases. Thus, a mechanism for constrained random generation of NFP provisions and requirements is needed. To ensure the existence of at least one feasible system configuration, a random request is generated, which serves as reference request for which a feasible system configuration is to be ensured. The process of system generation keeps track of how the random request transforms[3] between dependent software component types. For the directly requested component type at least one (randomly chosen) quality mode Q_1 is selected to fulfill the request. Then, for all dependent component types at least one quality mode is chosen to fulfill the requirements of Q_1. The same process is performed for all dependent types of the dependent types and so on. The generated user request to ensure feasibility is reused later as test request for evaluation.

7.3.2. Measurements for Selected Types of Generated Systems

A typical architectural style are pipe-and-filter architectures. This style is characterized by a data flow among component types. The most common type of pipe-and-filter architectures is a chain of components. In other words, there are n component types, where the first component type requires the second, which in turn requires the third component type and so on. For this architectural style the parameter n, denoting the number of component types or the depth of the chain is of interest. In addition, the number of containers c is to be considered, because the number of possible configurations grows exponentially with the number of available containers. Thus, a test system in pipe-and-filter style is characterized as an $n \times c$ system having n component types and c containers. Please note that the approach is not limited to chains of component types, but is able to handle any system, which can be modeled as acyclic data flow diagram.

For the server landscape we assume each server (i.e., container) to have one CPU, one RAM module, one HDD and a network device. Each software component type has one provided and one required port type, except the last component type in the chain, which only has a provided port type. Moreover, each software component type has two NFPs and two implementations, having two modes, which each have four requirements against hardware component types, two requirements against the succeeding software component type and two provisions. Figure 7.4 shows these parameters graphically.

For each generated system two measurements are taken: the time required to generate the respective ILP or PBO problem and the time required to solve the problem using

[3] A dependency between two component types as specified in a QCL contract can be interpreted as a request (cf. Sect. 6.1 for an example).

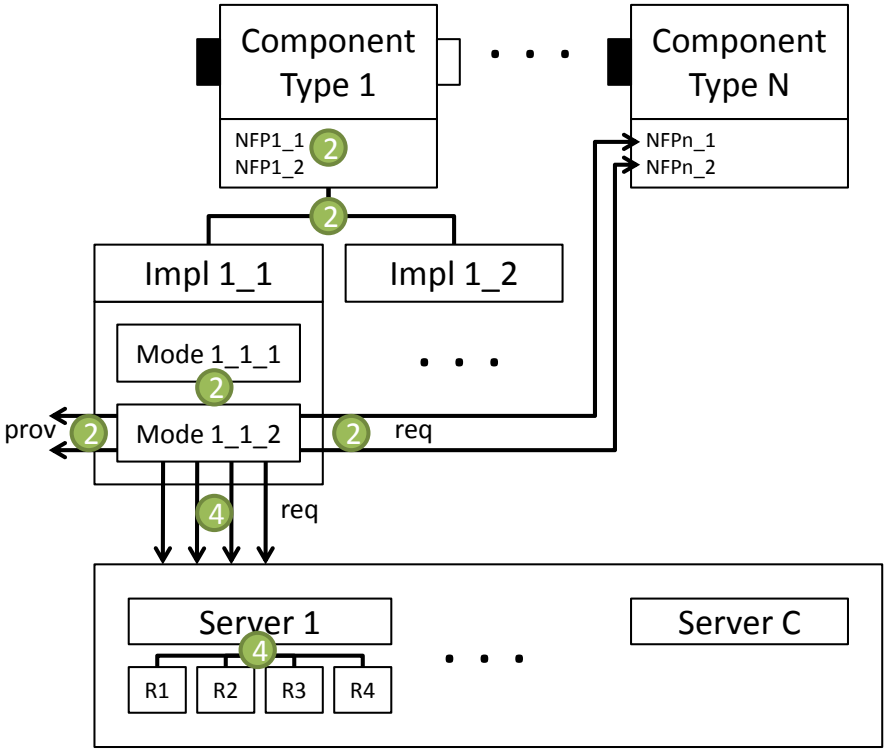


Figure 7.4.: Parameters of Generated Pipe-and-Filter Architectures.

a standard solver. For the ILP solution the solver *LP Solve* [54] has been used in Version 5.5.20. For PBO the *OBPDB* [10] solver has been used in version 1.1.3. All measurements were taken on a DELL Alienware X51 desktop PC running Windows 7 64bit and containing a solid state disk, 8 GB DDR1600 RAM and an Intel Core i7-2600 CPU running at 3.4 GHz having 4 physical cores and 8 virtual cores by hyperthreading.

Analysis of the ILP Solution

For the ILP solution to pipe-and-filter architectures, the number of variables and constraints can be derived from the system generator parameters. The total number of variables is the sum of decision, resource negotiation and software NFP negotiation variables. The number of decision variables $vars_d$ correlates to the number of component types, implementations per type, modes per implementation and the number of servers. The number of resource negotiation variables $vars_{res}$ correlates to the number of servers, resources per server and properties per resource. Finally, the number of software NFP negotiation $vars_{nfp}$ variables correlates to the number of component types and the number of NFPs per component type. If all system generator parameters are set to x, except

for C and S, $var^{(x)}$ denotes the simplified formula to compute the number of variables in this setting.

$$\begin{aligned} vars_d &= C \cdot N_{impl} \cdot N_{mode} \cdot S \\ vars_{res} &= S \cdot N_{res} \cdot N_{resProp} \\ vars_{nfp} &= C \cdot N_{nfp} \\ vars &= vars_d + vars_{res} + vars_{nfp} \\ vars^{(1)} &= C \cdot S + C + S \\ vars^{(n)} &= n^2 \cdot CS + n \cdot C + n^2 \cdot S \end{aligned}$$

Thus, if all generator parameters are set to 1, except for the number of component types and servers, the variables can be computed by $vars^{(1)} = C \cdot S + C + S$. For example, for 100 component types to be mapped on 100 servers, with all other parameters set to 1, a total of $10.000 + 100 + 100 = 10.200$ variables results. For 15 component types on 20 servers 335 variables result.

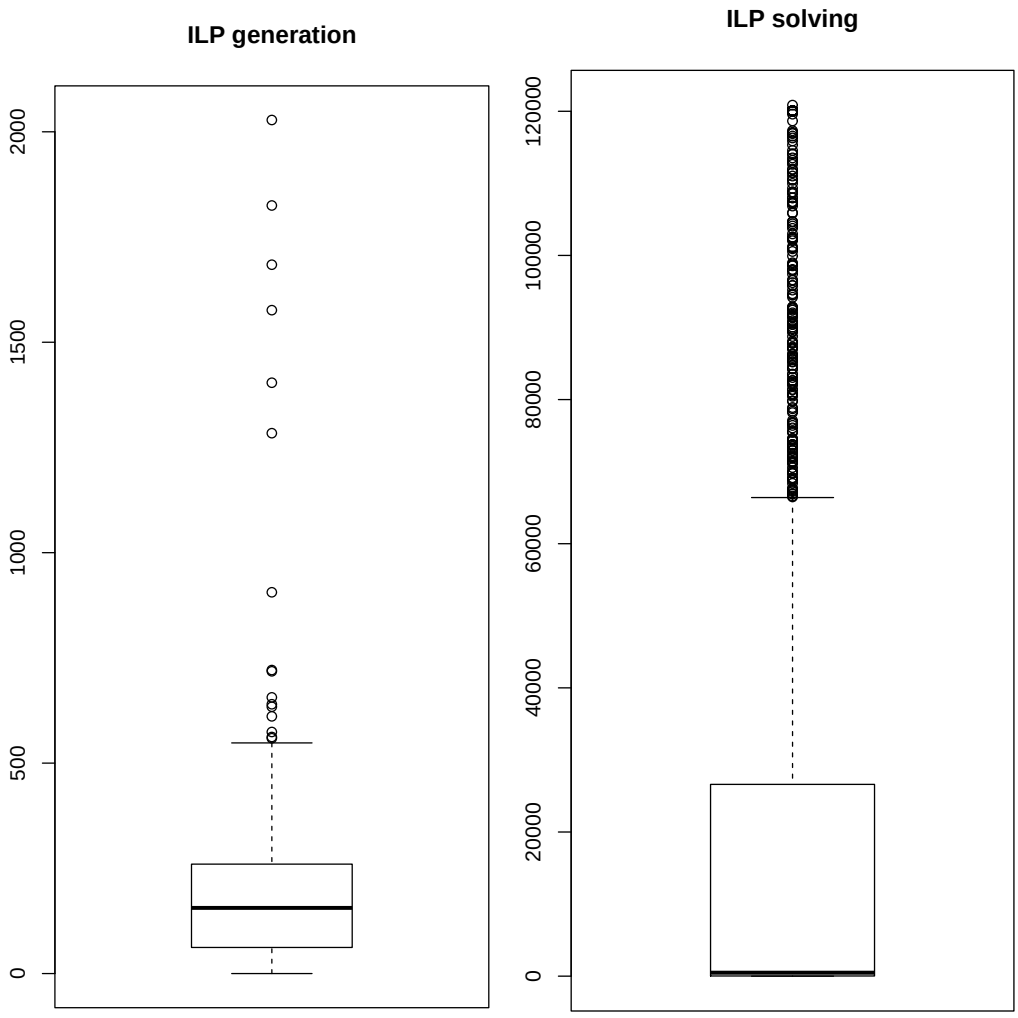
The number of constraints (ctr) can be computed, too, as shown below. For the examples above $ctr^{(1)}_{C=15,S=20} = 105$ and $ctr^{(1)}_{C=100,S=100} = 600$.

$$\begin{aligned} ctr_{arch} &= C \\ ctr_{res} &= 3 \cdot vars_{res} = 3 \cdot S \cdot N_{res} \cdot N_{resProp} \\ ctr_{nfp} &= 2 \cdot vars_{nfp} = 2 \cdot C \cdot N_{nfp} \\ ctr &= ctr_{arch} + ctr_{res} + ctr_{nfp} \\ ctr^{(1)} &= 3C + 3S \\ ctr^{(n)} &= (2n + 1) \cdot C + (3n^2) \cdot S \end{aligned}$$

To assess the scalability of the ILP and PBO solution, a set of pipe-and-filter systems has been generated and measured. These are all variants of $C \times S$ systems for $C = [2..100]$ and $S = [2..100]$. That is for a total of $99 \cdot 99 = 9801$ systems, the generation and solving of ILPs and PBOs has been measured.

Figure 7.5(a) shows a boxplot of the generation time required to generate the ILPs. The median generation time is 156 ms and 75% of all ILPs were generated in at most 260 ms. The longest generation took 2028 ms. Notably, the 99%-quantile is 437 ms, meaning that in 99% of all cases the maximum generation time is less or equal to 468 ms. A natural hypothesis is that the number of components and servers correlates to the generation time. This correlation indeed exists[4]: $T_{gen} = f(C) = 0.0291C^2 + 1.4429C + 5.3851$ with an adjusted R^2 of 0.8956.

[4]Derived using the statistical tool R [31].



(a) Generation Time in Milliseconds (b) Solving Time in Milliseconds

Figure 7.5.: Boxplots of Generation and Solving Time for the ILP Solution.

Figure 7.5(b) depicts a boxplot of the solving time of ILPs. This boxplot reveals the random nature of worst and best case runtime of ILP solving, depicted by the vast amount of outliers. Only for 121 out of 9801 generated ILPs the solver could not return any solution in two minutes for all measurements taken (i.e., in 1.2% of all cases). The solving time had an upper limit at 2 minutes, because the solving time of an ILP can increase to multiple hours or even days[5]. Surprisingly, the median solving time is only

[5]This is the worst case, where the complete problem space has to be explored.

at 478.5 ms. Thus, half of the ILPs could be solved in less than a second. The third quantile (i.e., 75%-quantile) is at 26.58 s. 79.1754165% of all ILPs were solved in less than a minute. Notably, if the timeout of two minutes is reached, the ILP solver returns the best solution found so far.

In the following, a closer investigation of the solving time will be presented. The aim is to investigate if and how the system parameters correlate with the solving time of the generated ILP. The hypothesis is that there is a correlation between the number of component types and the solving time, i.e., the solving time depends on the number of component types. Figure 7.6 depicts this correlation. On each axis a boxplot of the corresponding variable is shown to highlight the high density of solutions in low solving times. An interesting conclusion from this figure is that the predictability of solving time decreases starting at approximately 25 component types. In general the figure illustrates that most ILPs are solved in a few seconds, though the more component types, the more likely are longer solving times.

The correlation between the number of component types and the solving time for scenarios is statistically poor. The linear regression has an adjusted R^2 of only 0.4286. Exponential regression (i.e., $T_{solve} = f(Components) = a \cdot e^x$) reveals a residual standard error of $1.911 \cdot 10^{13}$. Thus, there is no statistically significant correlation between the number of components and the solving time. The reason is the random nature of ILP solving, i.e., solving random ILPs can randomly lead to worst and best case situations.

The same measurements on a slower machine with a standard hard disk drive (not a solid state disk) show, as to expect, slightly poorer results. On an HP Envy 15 Laptop, with an Intel i7 Q720 quad core CPU running at 1,6 GHz, 8 GB of RAM running Windows 7 Home Premium 64bit the following numbers result: the median generation time for ILPs is 535 ms (cmp. to 156 ms), the median solving time for ILPs is at 2,4 s (cmp. to 468 ms). The predictability of ILP solving time starts to decrease already at 20 component types (cmp. to 25). Nevertheless, until 20 component types the solving time is below 3 seconds. Hence, even on the slower machine, the ILP approach to contract negotiation shows up to be feasible.

Analysis of the PBO Solution in Comparison to the ILP Solution

In the following, the same investigations are shown for the PBO solution. First, Figure 7.7 depicts the boxplots of the PBO generation and solving times. Please note that for the PBO solution only systems with up to 30 component types have been measured, because the approach does not scale beyond this number of component types.

In comparison to the ILP solution, the generation of PBOs looks comparably performant at a first glance. Notably, the generation process includes the simulation required to construct the objective function. ILPs for systems of up to 100 component types are generated in up to 2 seconds. PBOs for systems of only up to 30 component types require up to 2 seconds, too. But, whilst the median for ILP generation was at 156 ms, for

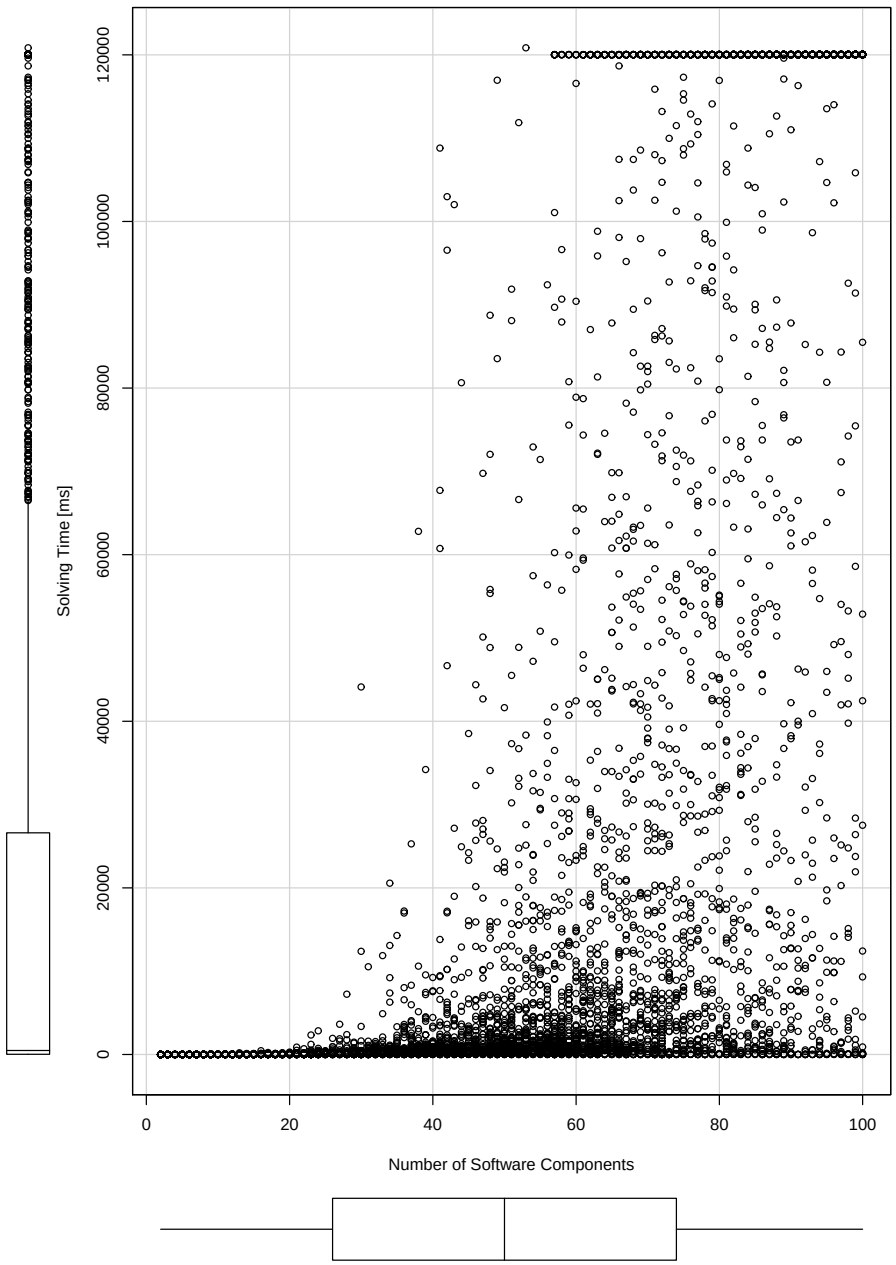
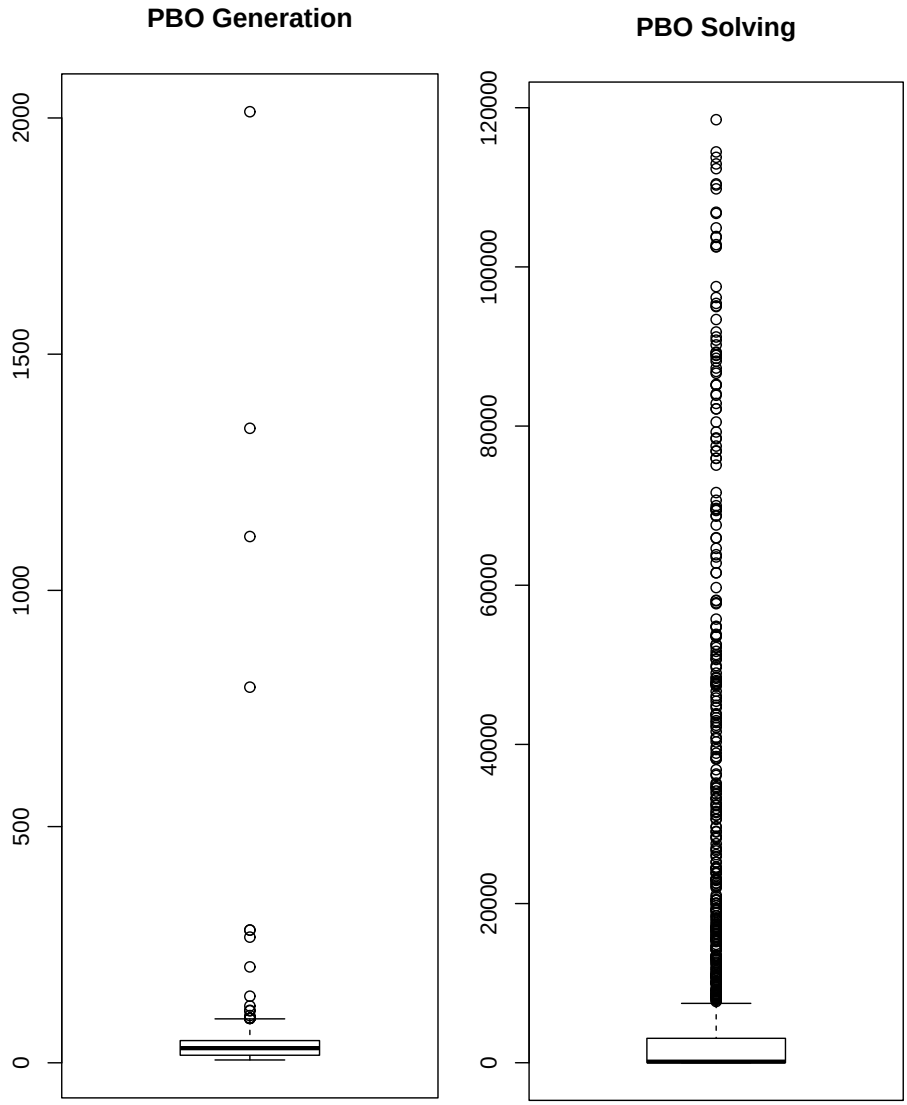


Figure 7.6.: ILP Solving Time in Relation to Number of Components.

PBO generation it is at 31 ms. Also for the 99%-quantile, the ILP solution looks worse than the PBO solution, as for ILP the 99%-quantile is at 468 ms, whereas for PBO it is



(a) Generation Time in Milliseconds (b) Solving Time in Milliseconds

Figure 7.7.: Boxplots of Generation and Solving Time for the PBO Solution.

at 94 ms. But, because for the PBO solution only systems of up to 30 component types have been used, the ILP generation times for up to 100 components cannot be used as a reference. An investigation of ILP generation for systems of up to 30 component types reveals a median of 32 ms and a 99%-quantile of 78 ms. Thus, PBO generation is even slower than ILP generation, although less constraints and less variables have to be gener-

ated. The reason for the better performance of ILP generation is the format required by the solvers. The ILP solution can use long variables names to encode information (i.e., the decision variables encode their meaning in their name), whereas the PBO solution has to use enumerated variables (i.e., x_n). In consequence, the PBO generation has to handle the mapping of decision variables to their short versions, which consumes time and, hence, leads to the measured performance loss.

For PBO the number of variables and constraints can be computed as follows.

$$\begin{aligned} vars &= C \cdot N_{impl} \cdot N_{mode} \cdot S \\ vars^{(1)} &= CS \\ vars^{(n)} &= n^2 \cdot CS \end{aligned}$$

$$\begin{aligned} ctr_{arch} &= C \\ ctr_{res} &= S \cdot N_{res} \cdot N_{resProp} \\ ctr_{nfp} &= C \cdot N_{nfp} \\ ctr &= ctr_{arch} + ctr_{res} + ctr_{nfp} \\ ctr^{(1)} &= 2C + S \\ ctr^{(n)} &= (n+1) \cdot C + n^2 \cdot S \end{aligned}$$

In comparison to the ILP solution, $vars^{(n)} = n^2 \cdot CS$ instead of $vars^{(n)} = n^2 \cdot CS + n \cdot C + n^2 \cdot S$. That is, the PBO solution generates $Cn + Sn^2$ less variables. The number of constraints is $ctr^{(n)} = (n+1) \cdot C + n^2 \cdot S$ instead of $ctr^{(n)} = (2n+1) \cdot C + (3n^2) \cdot S$, which leads to $Cn + 2Sn^2$ less constraints.

An investigation of the solving time of the PBO solution reveals surprising results. The rational for using PBO instead of ILP was the hypothesis that PBO performs better in many cases. For contract negotiation it does not, as Figure 7.8 depicts. The solving time has been limited to two minutes, too. But, in contrast to the ILP solution, the PBO solver does not deliver a suboptimal solution if the timeout is reached. Starting at 15 component types (compared to 25 in the ILP solution), the predictability of the solving time drastically decreases. Most interesting is the lack of fast solutions starting at 25 component types. In comparison, the ILP solution was able to deliver solutions in a few milliseconds even for systems with more than 80 component types. Moreover, the number of determined solutions in a time frame of less than 2 minutes significantly reduces starting already at 20 component types. For example, the measurements taken for 28 component types and 2 to 100 servers (i.e., 99 measurements) only lead to 4

solutions. For the remaining 95 systems the PBO solution could not find a configuration. For this reason, only measurements for up to 30 component types have been collected.

Although the PBO solver is incapable of providing a sub-optimal solution if the timeout is reached and, hence, is hardly practically applicable, it has a special use: it forms the basis for the a posteriori approach to contract negotiation presented in Chapter 9.

The measurements of the solving time indeed benchmark the used solver. Unfortunately, for PBO only one solver could be investigated, because all other solvers do not support the specification of equalities with variables on both sides of the equation. OPBDP in Version 1.1.3 was the only publicly available, working solver which could be investigated.

Thus, in conclusion, the PBO solution performs much worse than the ILP solution to contract negotiation. Whereas the ILP solution is feasible for systems of up to 100 component types, the PBO solution can handle at most 25 component types. The generation of ILPs is below 437 ms for up to 100 component types and PBO generation takes less than 100 ms in 99% of all cases for systems of up to 30 component types. Notice that the generation process realizes a technology bridge [100] from the applications domain to the respective optimization domain. Solving of ILPs is below 500 ms in 50% of all cases and below 27 seconds in 75% of all cases. PBO solving is below 2.6 s in 50% of all cases, but reaches the timeout of 2 minutes already in 70% of all cases. Most notably, the PBO solution is not able to find configurations starting already at 25 component types in most of the cases, whereas the ILP solution is able to determine configurations even for 100 component types. Figure 7.9 depicts the percentage of solutions found by the ILP and PBO approach in correlation with the number of component types. The execution time of the ILP solution is predictable up to 25 component types, whereas the execution time of the PBO solution starts to decrease already at 15 component types.

7.4. Summary

In this chapter, two approaches to exact contract negotiation have been presented. First, the application of integer linear programming (ILP) for contract negotation has been described. Then, a transformation from ILP to pseudo-boolean optimization (PBO) has been shown, which is a special type of ILP restricted to use boolean variables only. An analysis of both approaches revealed the feasibility of both approaches, where the ILP approach is feasible for systems of up to 100 component types and 100 servers and the PBO approach for systems of up to 25 component types and 100 servers.

In conclusion, both approaches have been shown to be applicable. Their major drawback is the limitation to a single objective function, which restricts their applicability for multi-objective optimization to a priori approaches. In the following chapters, first an approximate approach to contract negotiation will be presented in Chapter 8, followed by an approach supporting a posteriori multi-objective optimization in Chapter 9.

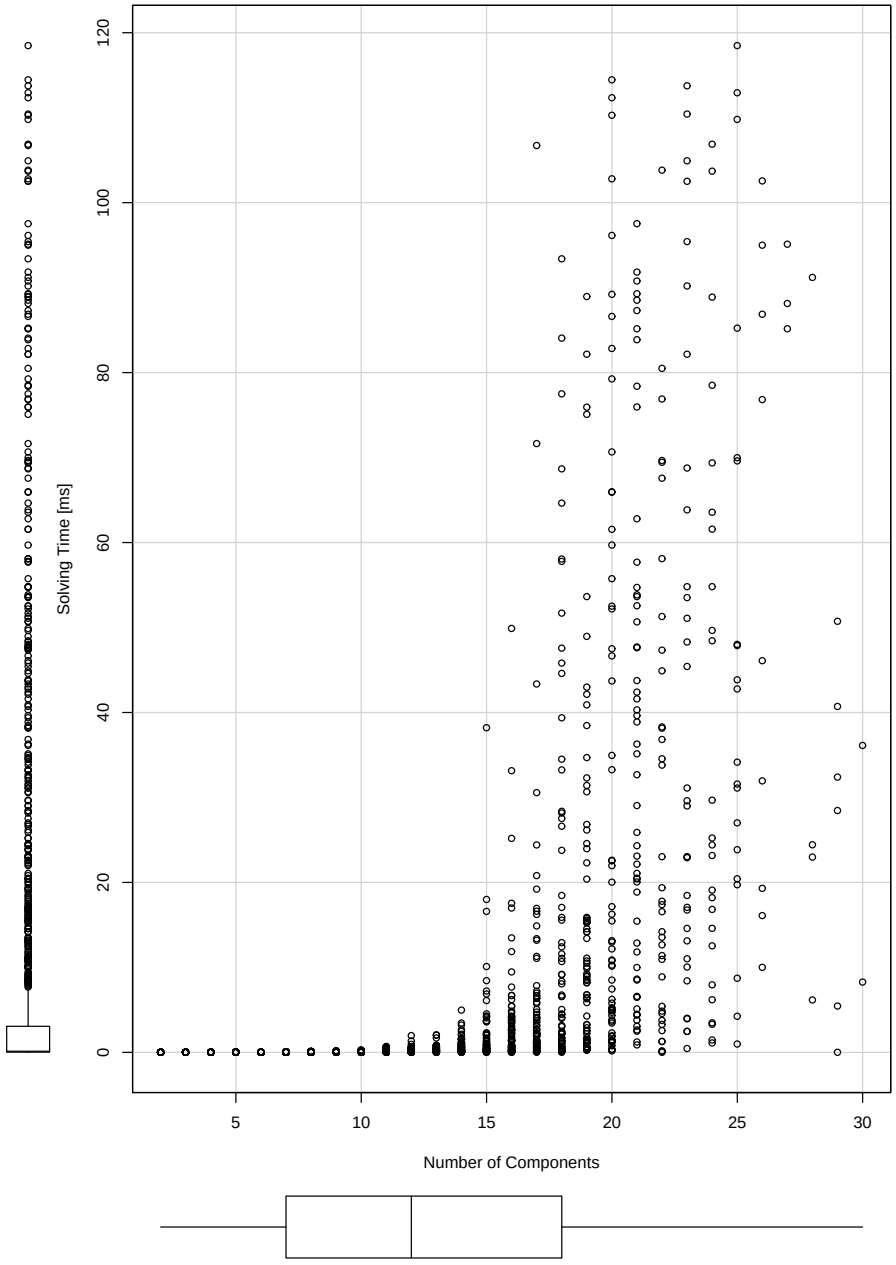
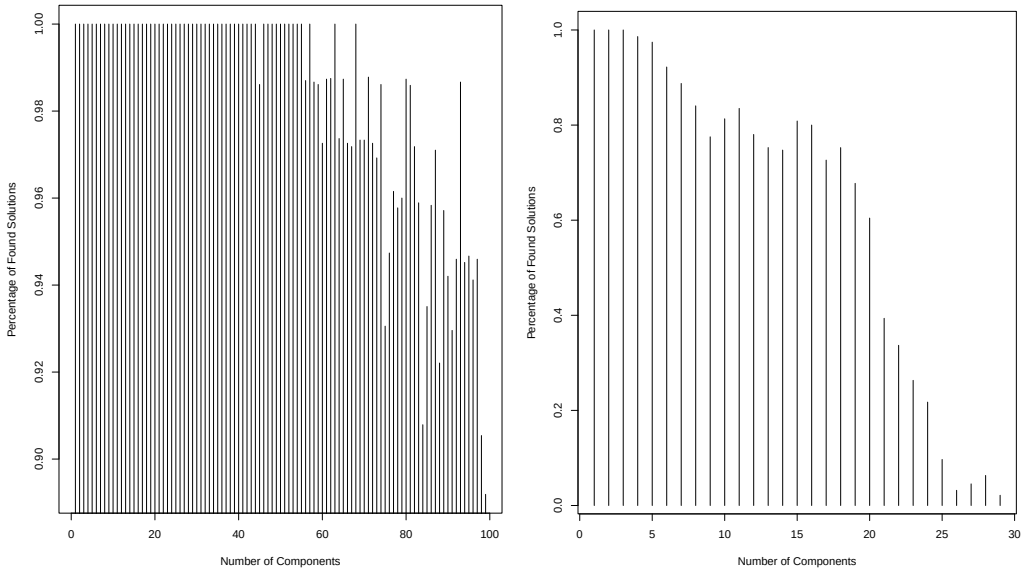


Figure 7.8.: PBO Solving Time in Relation to Number of Components.



(a) ILP approach

(b) PBO approach

Figure 7.9.: Percentage of Determined Solutions in 2 Minutes for ILP and PBO Approach.

8

An Approximate Approach for Multi-Quality Auto-Tuning

The approaches to contract negotiation presented in the last chapter ensure an exact solution to the optimization problem, i.e., ensure an optimal system configuration as a result. In this chapter, an approximate approach will be presented, which cannot ensure an optimal, but a near-optimal solution. The rational for investigating an approximate approach is the hypothesis that there is a trade-off between accuracy and runtime. That is possibly the approximate approach allows to trade the accuracy of the determined solution (i.e., how near the configuration is to the optimal configuration) and the required execution time.

In the following, first an approach to contract negotiation using the ant colony meta-heuristic [50] will be presented. Then, the scalability of this approach is analyzed and compared to the exact solutions presented in the previous chapter–showing that ACO performs much worse than both exact solutions–and rejecting the hypothesis of a runtime-accuracy trade-off.

The contributions given in this chapter are the following:

- An approximate runtime optimization approach using ACO. (Section 8.1)
- A scalability analysis of ACO for contract negotiation. (Section 8.2)

8.1. Contract Negotiation by Ant Colony Optimization

Ant Colony Optimization (ACO) is a well-known approximate optimization technique, which has been applied in many domains like vehicle networks [58] and theoretical problems as the quadratic assignment problem [85] or subset problems [83]. Notably, an approach to deploy software components on available resources using ACO has already been investigated by Aleti et al. [4]. In contrast to the approach presented in this thesis, the optimization problem is not generated, does not consider multiple implementations per component and only supports a fixed set of NFPs: memory, bandwidth, reliability and delay. In the following a realization of contract negotiation as ACO problem is discussed. Then, an analysis of the performance and accuracy of ACO for contract negotiation is shown.

8.1.1. Generating the Optimization Problem for Ant Colonies

The application of ACO to the shortest path problem as shown in Section 2.3.2 had the piece of luck that the problem is already expressed in terms of a graph. For contract negotiation this is not the case. Hence, in the following, first the translation of the optimization problem into a graph representation is shown. Then, the concretizations of the ants' and colonies task is presented.

Contract Negotiation Expressed as Graph Problem

To express the problem of contract negotiation in terms of a graph, the intended solution needs to be part of that graph. More concretely, a path through this graph should represent a solution of contract negotiation, i.e., a system configuration. As a system configuration denotes the selection of a set of implementations in a certain quality mode for a set of required software component types and their mapping to hardware resources, the graph has to comprise these concepts.

Figure 8.1 shows the basic structure of the problem graph for ACO for a single component type (i.e., a request which can be served by a single implementation). Each problem graph has a root node serving as a starting point for all ants. Then, for the software component type a separate node is created. The same applies to each existing implementation (represented by a QCL contract). Edges are created between the root node and all software component types and between the software component types and their implementations. For each quality mode, several nodes are created. This is because each mode has a different concretization for each server (cf. Chapter 5). Thus, for each pair of existing server and quality mode a separate node is introduced. An edge is created between the implementation node and all quality mode nodes. Finally, each existing server and its comprised resources get a corresponding node in the graph, too. Edges are introduced between the nodes representing a quality mode on a certain server

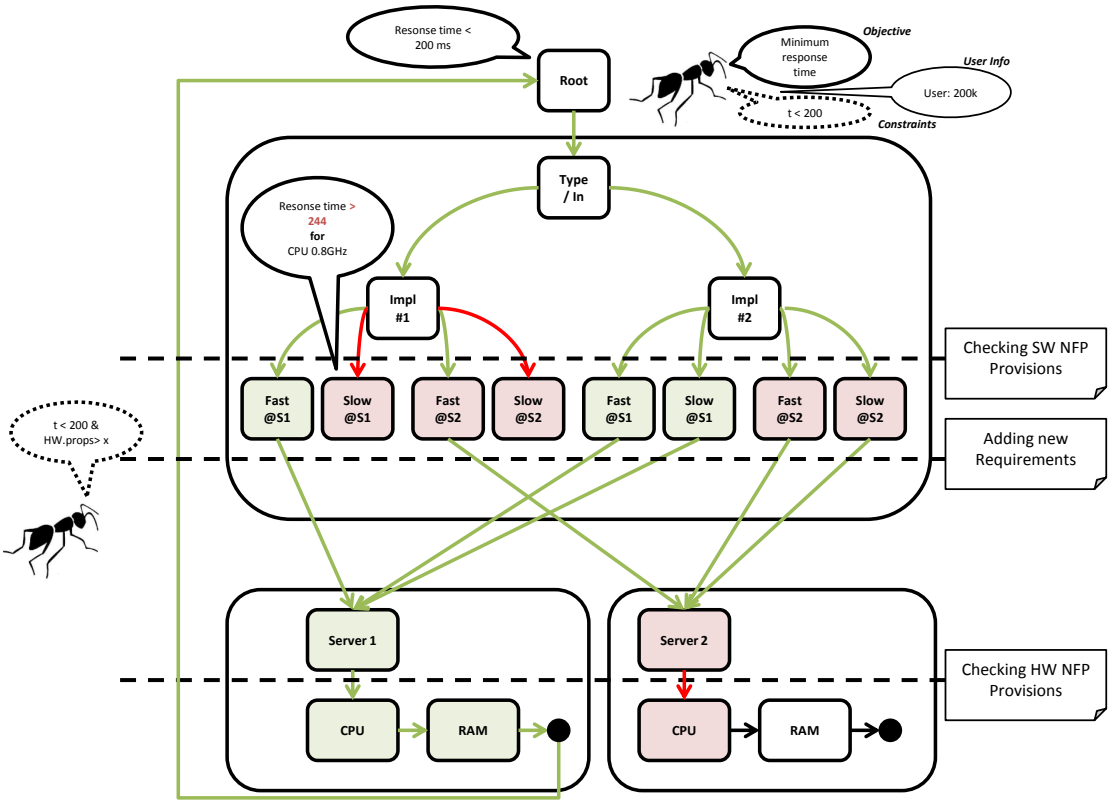


Figure 8.1.: Contract Negotiation by Ant Colony Optimization for a Single Component Type.

S and the node representing server S. The nodes of servers and their resources form a chain. At the end of each chain an edge to the root node is introduced, which is required for scenarios with multiple required software component types.

Each quality mode node is enriched with additional information, namely, the NFP and resource requirements as well as NFP provisions expressed in this quality mode. This information is required by the ants for proper path finding. In addition, each server and resource node comprises its provisions for the decision making process of the ants. Finally, the implied cost is annotated at each quality mode node.

Figure 8.2 depicts a scenario with two required software component types. In addition, servers are represented as reducible nodes having an input and output node. The input node maps to the server nodes in Figure 8.1, whereas the output node corresponds to the last node in the resource chain of that server. As illustrated in Figure 8.2, the number of quality mode nodes and edges to servers grows exponentially with the number of software component types and servers. For each additional server, an additional quality

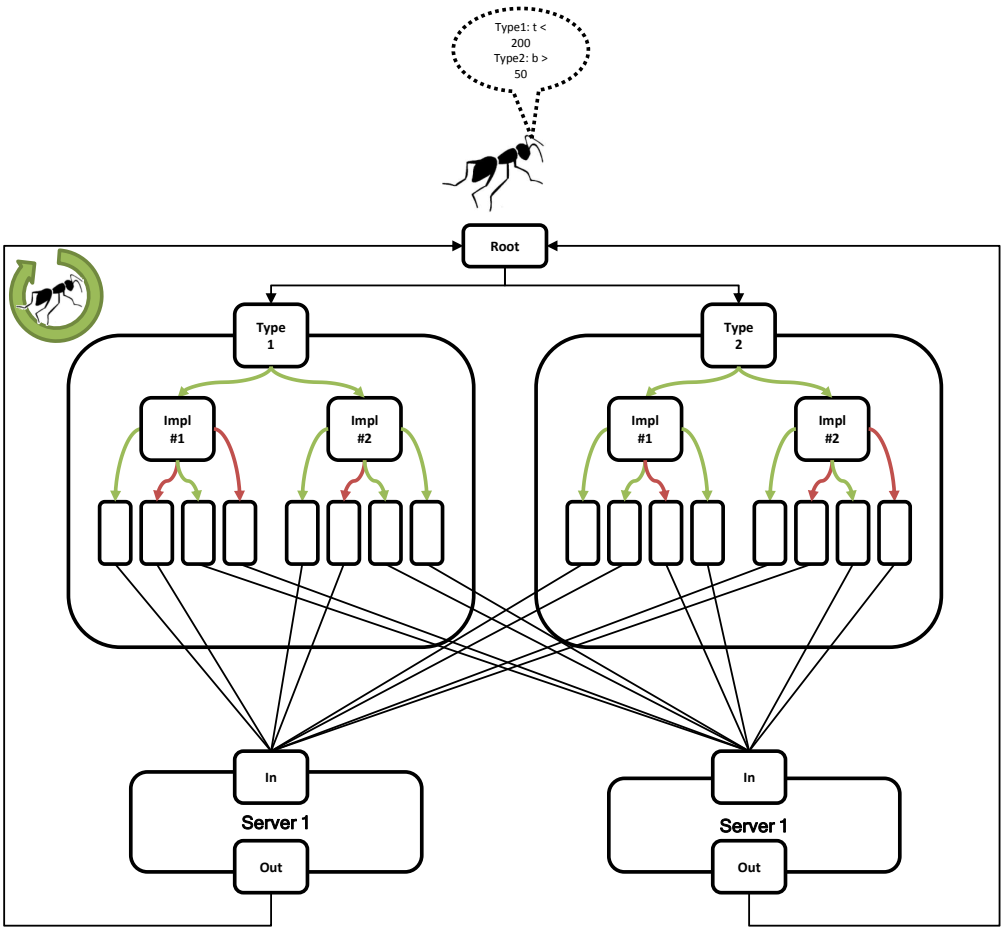


Figure 8.2.: Contract Negotiation by Ant Colony Optimization for Multiple Component Types.

mode node and edge is introduced for each quality mode.

In the following the path finding of ants through this graph and the tasks of the colony for contract negotiation are explained.

Task Concretizations of Ants and Colony

To perform contract negotiation, the path finding of the ants has to consider the information encoded in and annotated to the problem graph. Each ant starts its search at the root node of the graph. To perform contract negotiation ants need a local memory (i.e., metaphorically: a brain), which initially contains the information comprised by the user request: a required software component type and a set of NFP requirements. This

information is required for the first move of an ant–the move from the root node to a software component type node. An ant randomly selects a software component type node it currently requires.

The succeeding move is the decision for an implementation and made randomly taking the current pheromone distribution into account. This is done by interpreting the pheromone distribution as probabilities. Equation 8.1 shows the decision formally. For example, if two implementations exist, whereas the first has a pheromone value of 10 and the second a pheromone value of 15, the probability to decide for the first implementation is $10/(10+15) = 40\%$.

$$p(Impl_i) = pheromone(Impl_i) / \sum_{j=0}^{n} Impl_j \tag{8.1}$$

The path finding as described by equation 8.1 is termed an elitist strategy [51]: elitist ants, which found a solution, influence the path finding of all succeeding ants in an iteration. Alternatively, in a non-elitist strategy all ants randomly seek a path. Whether an elitist or non-elitist strategy performs better depends on the actual problem to be solved. For example, White et al. [129] have shown a non-elitist being superior to an elitist strategy for the traveling salesman problem. In the approach presented in this thesis, the elitist strategy is applied as it showed a better solution accuracy than the non-elitist strategy.

Once an implementation has been chosen, the ant has to decide for a server-specific quality mode node. This decision is influenced by the annotated cost in the same way as the pheromone influences the decision at other points in the graph. Furthermore, this decision leverages the annotated NFP provisions. First, the ant randomly chooses a server-specific quality mode considering the annotated cost. Then, the ant checks, whether the selected mode satisfies the requirements, which are currently in the memory of the ant. If they are met, the ant moves to the mode node. If not, the ant has failed in finding a path through the graph and is reset. In contrast to backtracking as a well known algorithmic approach, ants fail if the path they took is invalid and do not try an alternative path.

For the next move an ant does not require to make a decision as there is only one outgoing edge for each quality mode node by construction. But, leaving the quality mode node the ant collects all software, NFP and resource requirements annotated to this node. At the server node, the ant moves from resource to resource until all resource requirements it has collected are fulfilled. If the ant reaches the end of the resource chain and still has unmet resource requirements (because the server does not comprise the required resource), the ant failed and is reset. Else, the ant moves to the root node.

If an ant is at the root node and has no requirements in its local memory, it finished its search successfully and notifies the colony to mark the found path with pheromones. Thus, in summary, ants start with initial requirements taken from the user request and

circle the problem graph as long as all requirements are met. This way a successful path through the graph encodes a valid system configuration.

For each valid path found by an ant the colony increases the pheromone of this path by a certain percentage (e.g., 5%). Whenever an ant fails, the pheromone on the path taken by the respective ant is reduced by a certain percentage (e.g., 5%), too.

The task of the colony is to manage this pheromone distribution and, additionally, it's evaporation. That is, the colony takes care of evaporating the pheromones after each iteration to decrease the probability of the ants to get stuck in a local optimum. In addition, the colony manages (i.e., coordinates) the activities of all ants comprised by the colony and represents the optimization problem. The pheromone distribution used in our approach resembles the standard approach from literature [51] and is shown in Equation 8.2.

$$\tau_{ij} \leftarrow (1 - \rho) \times \tau_{ij} + \sum_{k=1}^{n} \Delta\tau_{ij}^{k} \tag{8.2}$$

Here, τ_{ij} denotes the edge connecting nodes i and j, ρ denotes the evaporation rate, n is the number of ants and $\Delta\tau_{ij}^{k}$ denotes the amount of pheromone put by ant k on the edge from node i to j. Each ant puts a fixed amount of pheromone on a path.

Ants with Multiple Objectives

Using ACO for contract negotiation has a central benefit in comparison to the presented exact approaches: it allows for *a posteriori* approaches to MOO. Two well-known approaches to apply ACO for MOO in literature are m-ACO [2], and Pareto ACO [49]. Existing alternatives to realize MOO with ACO divide by:

1. the way pheromone structures are used–either separate structure for each objective or a single, combined structure,

2. how the pheromone trails are rewarded–either only the best solution found during an iteration or the whole Pareto front is rewarded and

3. how the heuristic factor of path finding is realized–either it combines all objectives or separate factors are used for each objective.

In m-ACO for each objective a separate colony is created and an additional colony aiming at optimizing all objectives in combination is added. This multi-objective colony has multiple pheromone structures–one for each objective–and is used to represent the set of solutions without merging the objective values.

In Pareto-ACO only one colony having multiple pheromone structures, one per objective, is used. The heuristic factor used for path finding is an aggregation of the

satisfaction of all objectives. The trail rewards are computed using the best and second best solution found in an iteration.

As will be shown in Section 8.2, ACO is too slow to be applicable at runtime even in the presence of a single objective only.

8.1.2. ACO Generation by Example

To illustrate the generation and solving of the contract negotiation problem with ACO the same example as used in the last chapter for the ILP and PBO solution is used (cf. Figure 7.2 on page 117).

Figure 8.3 depicts the generated ACO graph. For each software component type (`ListGen`, `Sort` and `Filter`) a node is generated. The same is performed for the `Random` implementation of the `ListGen` component type, both implementations of `Sort` and the implementation of `Filter`. Furthermore, each implementation was defined to have two modes. For each mode, two nodes are introduced: one per server. In the example, two servers are considered to have a CPU, RAM and an HDD each. Each resource and the servers themselves are represented by a node.

The thickness of the directed edges shown in Figure 8.3 depicts the amount of pheromone distributed by the colony after several runs. Thus, according to the figure the solution proposed by ACO is to use the `Random` implementation of `ListGen` in its `fast` mode on `Server 2`, the `QuickSort` implementation in its `immediate` mode on `Server 1` and the `Bounded` implementation of `Filter` in its `fast` mode on `Server 2`.

In comparison to the exact approaches presented in the last chapter, the accuracy of this solution is hard to assess. In the general case it is unknown, whether this solution is optimal, near-optimal or even worse. Therefore, in the following section, we will present an approach to assess the accuracy of ACO and evaluate the performance and accuracy of ACO for contract negotiation.

8.2. Scalability Evaluation

To assess the accuracy of the solutions found by ACO, the same generated test systems already used to examine the scalability of the ILP and PBO solution are reused. As the ILP solution is able to deliver an optimal solution, this corresponding value of the objective function is taken as a reference obj^+ (i.e., best possible value). Moreover, the ILP solution can be used to determine the worst solution by changing the objective from minimization to maximization or vice versa. The corresponding value of the objective function is taken as a second reference obj^- (i.e., worst possible value). The objective value v found by ACO is necessarily between obj^+ and obj^-. This allows to compute the accuracy α as a value between 0 and 1 of an ACO solution using its objective value v by $\alpha(v) = 1 - \frac{(v - obj^+)}{(obj^- - obj^+)}$ where $obj^+ \neq obj^- \neq 0$. The values for obj^+ and obj^- have been

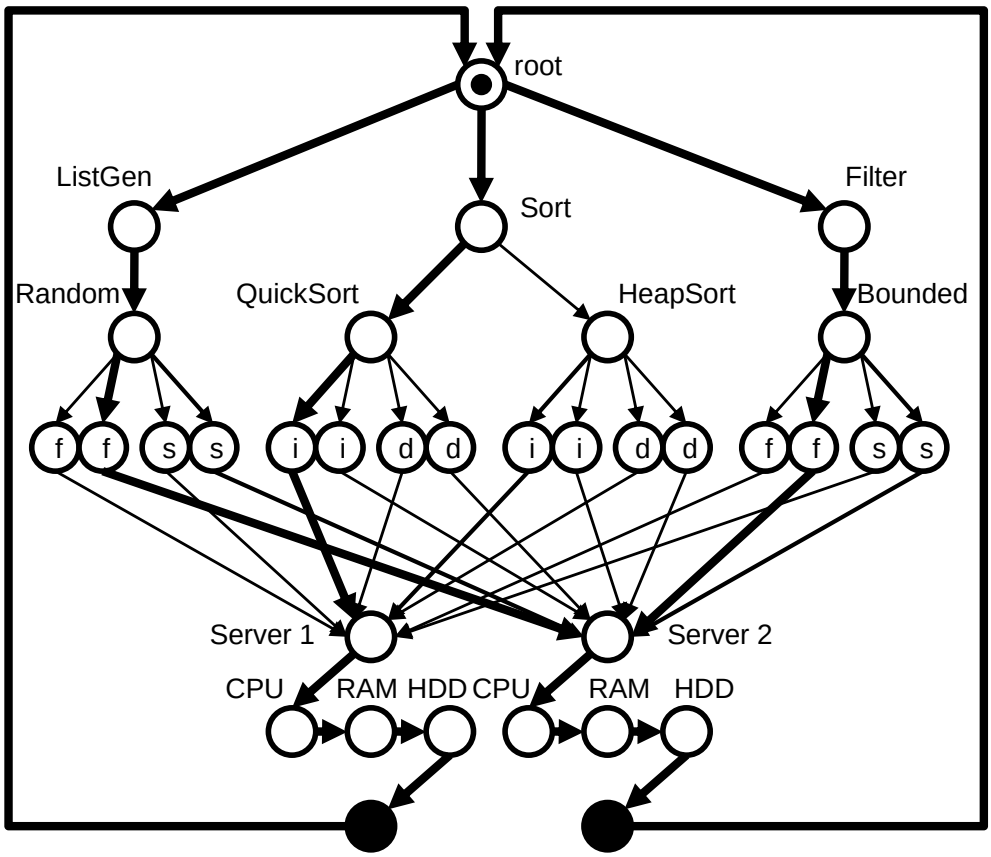


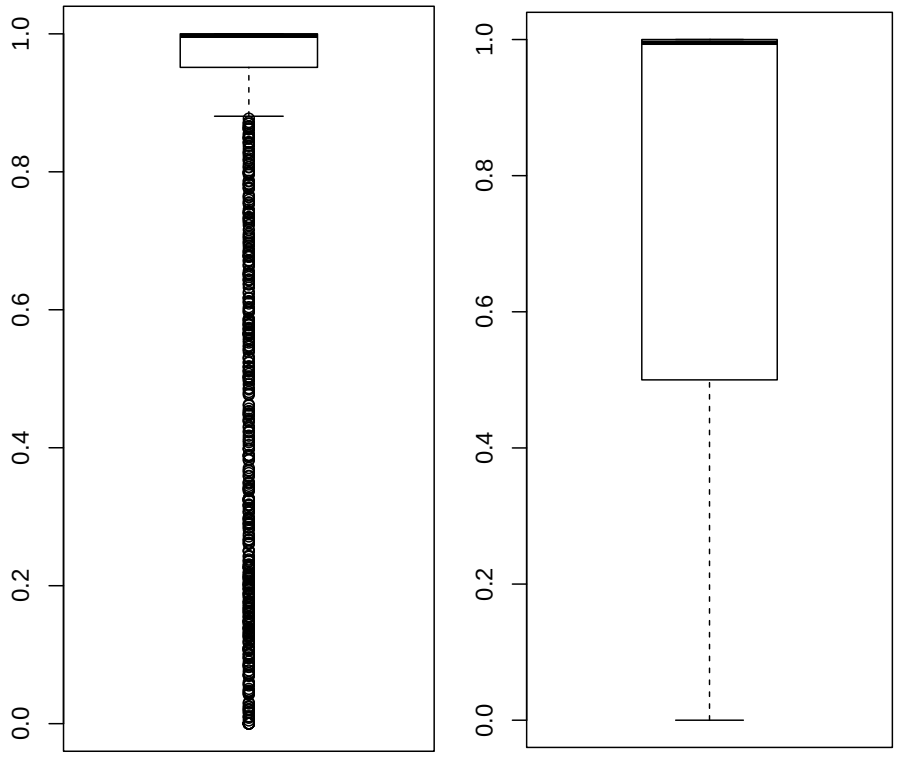
Figure 8.3.: Generated Ant Colony for List Generated/Sort/Filter Example.

collected for all generated systems from 2x2 to 100x100 component types per server.

To analyze the performance and accuracy of the presented ACO approach a set of parameters applicable to ACO have to be considered. The parameters considered in this thesis are the following:

1. Number of ants (n)
2. Number of iterations (it)
3. Evaporation rate (ρ)

All three parameters significantly influence the search. The number of ants increases the time required for a single iteration. The number of iterations in turn increases the total time required for the search. The evaporation rate decreases the probability to get stuck in local optima.



(a) 100 Ants x 100 Iterations (b) 10 Ants x 10 Iterations

Figure 8.4.: Accuracy of ACO Solutions with an Evaporation Rate $\rho = 0.1$.

In the following, measurements for generated systems of 2x2 to 30x100 component types per server (i.e., 2871 measurements) using 100 ants and 100 iterations and an evaporation rate $\rho = 0.1$ are shown. Detailed results of other parameter settings are omitted, but outlined instead.

Figure 8.4(a) depicts the accuracy reached for all systems from 2x2 to 30x30 component types using 100 ants, 100 iterations and an evaporation rate $\rho = 0.1$. It shows a very high accuracy with a mean value of 0.8667. Notably, even the first quantile is at 0.9514. Thus, using 100 ants and 100 iterations leads to an applicable accuracy. In comparison, Figure 8.4(b) depicts the accuracy reached using 10 ants and 10 iterations only. It shows a significantly worse accuracy with the first quantile at 0.5 only.

The runtime of the ACO meta-heuristic applied to contract negotiation is depicted in Figure 8.5 in correlation with the number of servers S and the number of component types C. It shows the expected behavior: the more components and the more servers, the longer the runtime. Indeed, this correlation can be statistically computed: $t_{solve}(S, C) = 0.065 \cdot C^2 S^3 + 1043.7[ms]$ with an adjusted R^2 of 0.9653.

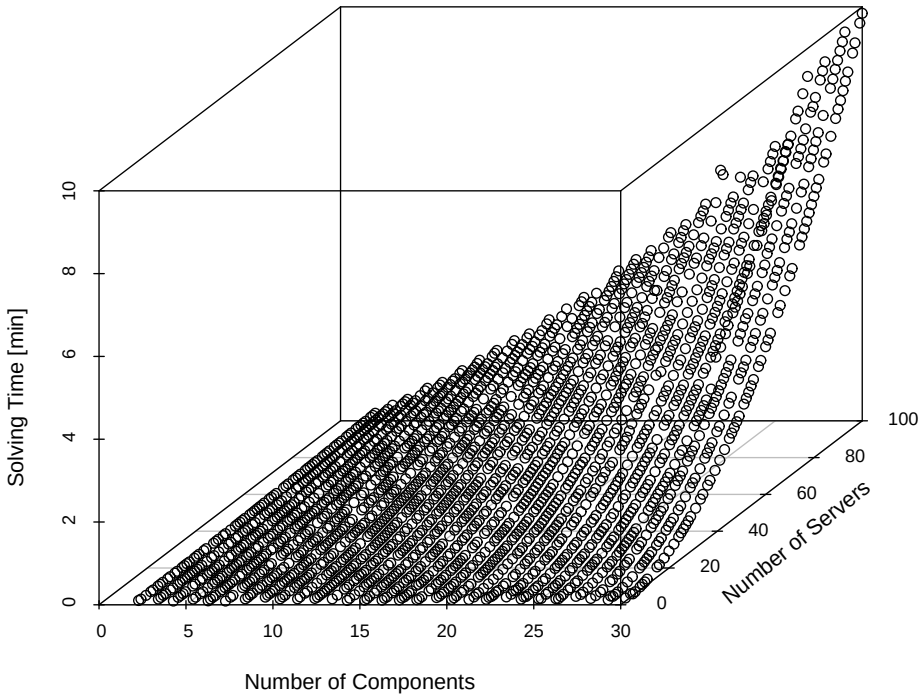


Figure 8.5.: Runtime of ACO Approach for 100 Ants, 100 Iterations and $\rho = 0.1$.

Another interesting finding is the correlation between accuracy and system size (i.e., number of servers and component types) as illustrated in Figure 8.6. Notably, solutions with a low accuracy only occur for systems with few servers. The reason is the combinatorial explosion of quality mode nodes with the increasing number of servers and, in consequence, the increasing probability of making bad decisions for the ants[1].

Despite the high accuracy reached by the ACO approach the runtime exceeds the boundary for practical applicability at runtime. Where the ILP and PBO approaches are able to deliver a solution in a few hundreds of milliseconds, ACO requires a few minutes. This runtime does not significantly decrease for fewer ants or iterations, because most of the time is required to build the graph and not to process it. Of course, the time required to build the graph grows with the numer of components and servers. Figure 8.7 depicts the boxplots of graph size (cf. Figure 8.7(a)), generation time (cf. Figure 8.7(b)) and graph processing time (cf. Figure 8.7(c)), showing that the graph generation takes much longer than graph processing. In the mean 92,58% of the time required for contract negotiation by ACO is required to generate the graph. Thus, tuning ACO by reducing the number of ants or iterations does not improve the overall performance.

[1]With an increasing amount of possible paths and ant can take, the probability window for a certain path (here the best) decreases.

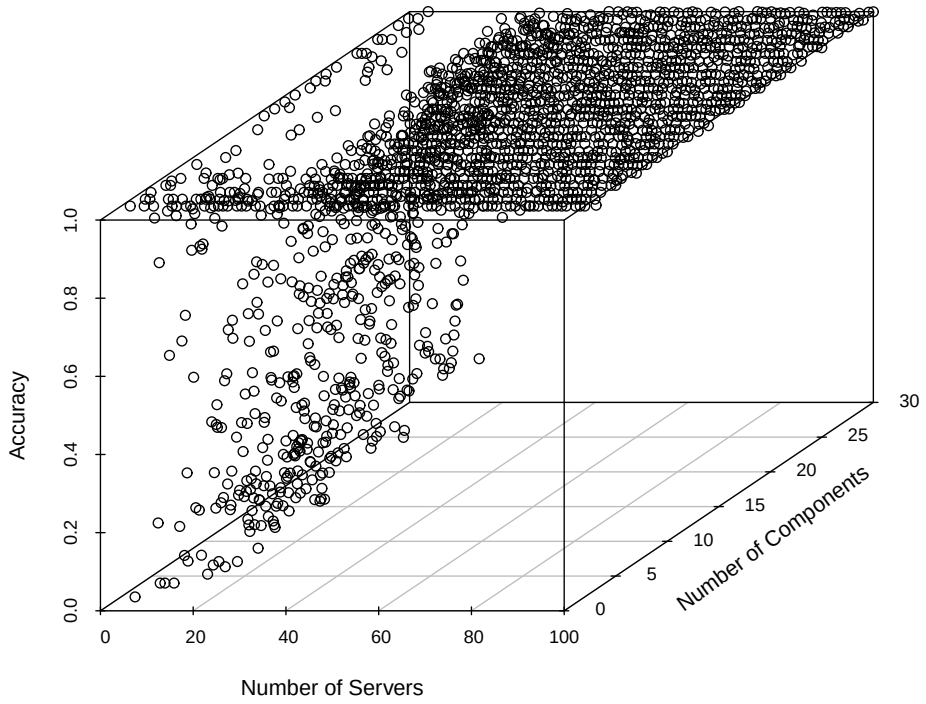


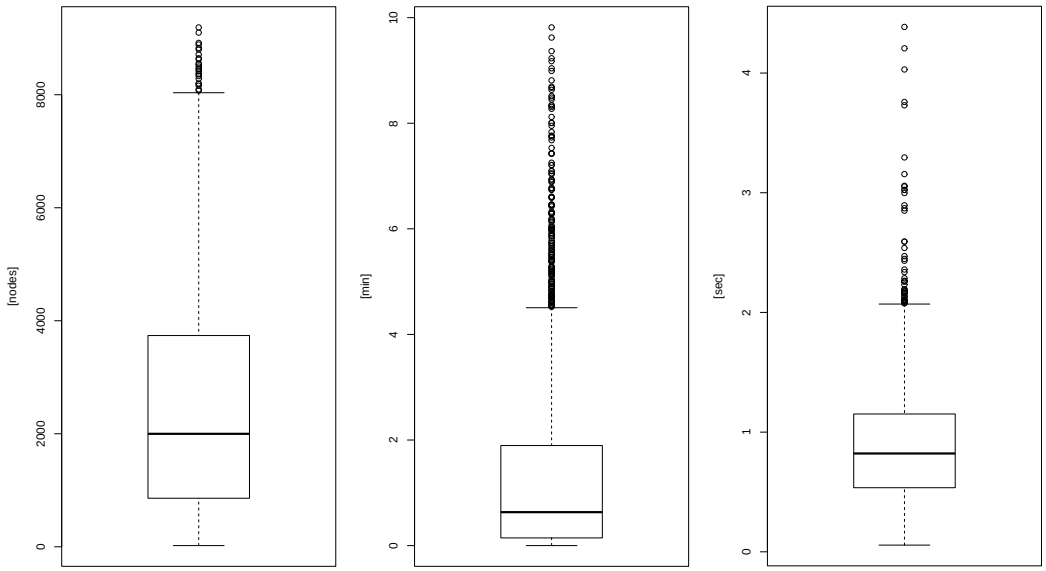
Figure 8.6.: Accuracy of ACO Approach in Correlation to System Size (for 100 Ants, 100 Iterations and $\rho = 0.1$).

8.3. Summary

In this chapter the application of ant colony optimization for contract negotiation has been presented. In contrast to the ILP and PBO approach the contract negotiation problem is transformed to a graph, which is processed by ants performing a shortest path search. A drawback of ACO and approximate approaches in general is their inability to assure the delivery of an optimal solution. Hence, in Section 8.2, the runtime and accuracy of ACO for contract negotiation has been evaluated against generated pipe-and-filter systems, where accuracy is measured as proximity to the optimal solution in relation to the worst solution. It has been shown that ACO can deliver very good solutions (the mean accuracy is at 0.8667), but requires much more time (several minutes) than the ILP and PBO approach.

The approach is applicable to any type of system, which can be modeled as data flow diagram using CCM and QCL. But, the conclusions drawn from investigating pipe-and-filter systems intensify for more complex structures, because they imply even bigger graphs to be build, which are the delimiting factor.

Hence, the hypothesis that ACO for contract negotiation allows to trade between



(a) Graph Size (b) Graph Generation (c) Graph Processing

Figure 8.7.: Boxplots of Graph Size and Generation as well as Processing Time.

accuracy and runtime with the ILP and PBO approaches is rejected, as both outperform the ACO approach. Nevertheless, the general applicability of ACO has been shown and the practical feasibility of ACO for contract negotiation has been refuted.

9 An A Posteriori Multi-Objective Optimization Approach for Multi-Quality Auto-Tuning

The approaches to contract negotiation presented in Chapter 7 ensure an exact solution to the optimization problem having a single objective function, i.e., do not separately consider each objective function but optimize a combination of them. In this chapter, exact approaches considering multiple objectives in parallel will be presented. The benefit of these approaches over single-objective approaches is the ability to compute the Pareto front [101] of the problem, which denotes a set of optimal solutions which are better in at least one objective than all other solutions. The approaches to be presented in this chapter, hence, realize *a posteriori* approaches to MOO.

In the following, an approach to multi-objective contract negotiation using the approach by Klein and Hannan [75] will be presented and analyzed.

The contributions given in this chapter are the following:

- An exact runtime multi-objective optimization approach using Klein and Hannan's algorithm. (Section 9.1)
- A scalability analysis of this approach. (Section 9.2)

9.1. Contract Negotiation by Multi-Objective Integer Linear Programming

This section covers an approach to identify a set of Pareto-optimal system configurations w.r.t. multiple objectives. This set of configurations is then presented to the user, who can select his preferred alternative. Each configuration presented to the user is the best in all but one objective. To determine this set a variety of approaches have been developed. Section 2.3 provides an overview on these approaches.

As shown in Chapter 7, integer linear programming (ILP) is applicable and feasible for contract negotiation, but limited to a single objective function. But, already in the eighties, various approaches have been developed, which allow to solve ILPs with multiple objectives. Bitran's approaches [19, 20] belong to the first published methods to solve 0-1 MOILPs (i.e., MOILPs with binary variables only) using implicit enumeration. Deckro and Winkofsky [45] as well as Klein and Hannan [75] provide further approaches using implicit enumeration. Finally, Kiziltan and Yucaoglu [74] provide a branch and bound algorithm to solve 0-1 MOILPs. Unfortunately, all these approaches, except for Klein and Hannan's algorithm, do not scale [109]. A further special type of MOILPs has been investigated by Burkhard, Krarup and Pruzan [80, 29]: 0-1 MOILPs whose first objective function is a sum function and all other objectives are bottleneck functions. 0-1 MOILPs with only two objective functions have been investigated by Pasternak and Passy [102]. Although these approaches perform much better than the more general approaches mentioned previously, they cannot be used for contract negotiation.

The restriction of 0-1 ILPs to use binary decision variables only does not pose a problem for contract negotiation as has been shown in Section 7.2. But, an appropriate MOILP approach for contract negotiation has to scale and has to support multiple objectives of free form. Hence, all previously mentioned approaches do not qualify, except for the approach introduced by Klein and Hannan in 1982 [75], which will be explained in more detail in the following.

9.1.1. Solving Multi-Objective Integer Linear Programs

Klein and Hannan developed an iterative approach to solve 0-1 MOILPs. First, an initial ILP is solved. Then, this initial ILP is extended with additional constraints and all resulting ILPs are solved, too. Each solution found by solving these single-objective ILPs is part of the Pareto front. In the following first a general definition of 0-1 MOILPs as known from literature is given as a basis for a succeeding in-depth explanation of Klein and Hannan's approach.

A 0-1 MOILP is defined as shown in Equation 9.1. C is a (p,n)-matrix covering p objective functions over n decision variables denoted by the vector x (of size n). A is a (m,n)-matrix covering the left hand side of m constraints over the n decision variables. The vector b of size m denotes the right hand side of these constraints.

$$min\{Cx : Ax \geq b, x \geq 0, x \in \mathbb{B}^n\} \tag{9.1}$$

For example, for $C = \begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix}, A = \begin{bmatrix} 1 & -3 \\ 2 & 5 \end{bmatrix}, b = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$, the following equations result.

$$\begin{aligned} min: \quad & 4x_1 + 2x_2 \\ min: \quad & x_1 + 6x_2 \\ 1x_1 - 3x_2 \quad & \geq 3 \\ 2x_1 + 5x_2 \quad & \geq 1 \end{aligned}$$

As mentioned, Klein and Hannan's approach is iterative. In each iteration an ILP is to be solved. The ILP of iteration i is denoted by P_i.

As a starting point P_0 is to be derived. Here one of the p objective functions is randomly selected denoted by $0 \leq s \leq p$. All other objective functions are omitted from P_0. This leads to a single-objective ILP, which can be handled by standard ILP solvers as shown in Chapter 7. The solution of P_0 is the first solution being part of the Pareto front and used to compute the succeeding ILPs.

In general the construction of ILPs of P_i for $i > 0$ depends on the solutions found until solving the last ILP (i.e., P_{i-1}). The variable r denotes the number of solutions found until P_{i-1}. Klein and Hannan formalized the construction as shown in Equation 9.2.

$$\begin{aligned} min \quad &: \quad z_s = c^s x \\ subject\ to \quad &: \quad Ax \geq b, x \geq 0, x \in \mathbb{B}^n \\ & \quad\ and \\ & \quad \bigcap_{i=1}^{r} \bigcup_{k=1 \wedge k \neq s}^{p} c^k x \leq c^k y^i - f_k \end{aligned} \tag{9.2}$$

Thus, all ILPs to be solved have a single objective function, namely, the objective function selected for P_0. The additional constraint added to the ILP depend on the number of solutions found until P_{i-1}, denoted by r, and the number of objective functions excluding objective function s selected in P_0. For each objective function except for s a set of alternative constraints, i.e., *or*-connected constraints, is added. Additionally, for each found solution these sets of alternative constraints are added to the ILP (i.e., *and*-connected). The constraint term $c^k x \leq c^k y^i - f_k$ describes that the k^{th} objective function has to have a value lower or equal to the value of the k^{th} objective function for solution y^i minus a constant f_k. If this constant is set to $f_k = 1$ the approach is guaranteed to provide the complete Pareto front [75][p. 380]. For bigger f_k a subset of the Pareto front is determined.

A crucial detail of the approach is the use of the boolean operators $\cup$ and $\cap$. Standard ILP solvers implicitly assume an *and*-connection between all constraints. The $\cup$-operator is not supported, but leads to multiple alternative ILPs instead. That is an ILP with two $\cup$-connected constraints cannot be solved directly by an ILP solver, but two ILPs have to be derived from the original ILP: one with the first constraint and another one with the second constraint. This leads to a combinatorial explosion of ILPs to be solved by Klein and Hannan's approach. For example, an ILP with three and-connected blocks of two or-connected constraints each (i.e., $((a,b),(c,d),(e,f))$), leads to 8 ILPs to be solved: $(ace, acf, ade, adf, bce, bcf, bde, bdf)$. As in Klein and Hannan's approach all or-blocks have the same size $(p-1)$, the number of ILPs to be solved in P_i can be computed as r^{p-1}. Despite this problem, the analysis shown in Section 9.2 reveals the applicability and feasibility of the approach for small systems.

Thus, the basic principle of Klein and Hannan's approach is to successively exclude solutions by enriching an initial ILP with more and more constraints. The original objective functions are used to iteratively restrict the feasible area of an initially selected single-objective ILP. In each iteration the solutions found are added to the Pareto front. The algorithm terminates when P_i is infeasible, i.e., no additional solutions are found.

The application to contract negotiation is straight forward, as the 0-1 ILP generation as described in Section 7.2 can be used without changes. The only difference is the support for multiple objective functions. To evaluate the scalability and performance of the approach the algorithm has been implemented, where the ILP solver from Chapter 7 has been reused.

9.1.2. Klein and Hannan by Example

To put the formal principle described in the last section into practice, in the following a small example is investigated. For clarity, this example is not the sort example shown in the previous chapters. The reason is that even for such a small system the generated ILPs are too large to showcase the basic principle of Klein and Hannan. Instead the smallest generated pipe-and-filter system–a 2x2 component types per server system–is used to illustrate the approach. The next subsection provides a practical example.

The MOILP for the generated 2x2 system is shown in Listing 9.1 with three randomly generated objective functions. Shortcut variables x_i are used instead of the long variable names as shown in the Chapter 7. The example comprises two servers and two component types having one implementation each, which in turn have one mode each. The decision variables x_i denote if an implementation a of component b in mode c is to be mapped to server d. In the example four variables result: x_0 denotes that the implementation of the first component is to be mapped to the first server. x_1 denotes that it is mapped to the second server. Analogously, x_2 and x_3 denote whether the implementation of the second component is to be mapped to the first or second server.

The architectural constraints on lines 6 and 7 specify that for both components a

```
1  /* Objective functions */
2  min: 15 x2 + 51 x3 + 17 x0 + 25 x1;
3  min: 28 x2 + 39 x3 + 19 x0 + 29 x1;
4  min: 24 x2 + 34 x3 + 76 x0 + 17 x1;
5  /* Architectural constraints */
6  x0 + x1  = 1;
7  x2 + x3  = 1;
8  /* Resource negotiation */
9  12 x3 + 22 x1 <= 84;
10 12 x2 + 22 x0 <= 48;
11 /* NFP negotiation */
12 97 x2 + 97 x3 >= 52 x0 + 52 x1;
13 /* Binary constraint */
14 bin x2, x3, x0, x1;
```

Listing 9.1: An Example MOILP for a Generated 2x2 System.

mapping decision is to be made. The resource negotiation constraints on lines 10 and 11 show that the first server offers a resource with 84 units, whereas the second server only offers 48 units. The implementation of the first component requires 22 units of this resource, whereas the implementation of the second component requires 22 units. Finally, the NFP negotiation constraint on line 13 denotes that the implementation of the second component offers a property with 97 units on both servers and the implementation of the first component requires 52 units of this property on both servers.

The objective functions on lines 2 to 4 are randomly generated. In practice they reflect NFPs of interest to the user like the response time, energy consumption or reliability. The value before each decision variable denotes the influence of the respective decision on the current objective. Imagine the first objective function (line 2) to represent response time. In this case, the mapping of the first implementation to the second server (x_1) creates a delay of 25 time units. Mapping this implementation to the first server (x_0) only implies a delay of 17 time units. The second objective could represent the noise level in an audio processing application. Minimizing the noise level conflicts with minimizing the response time, as more complex algorithm with longer execution times might be required to achieve a lower noise level.

Solving this MOILP results in a Parent front with two solutions:

- {x0=1, x1=0, x2=1, x3=0} and
- {x0=0, x1=1, x2=1, x3=0}

The first solution identifies the mapping of both implementations to the first server as best, whereas the second solution identifies the usage of both servers as best. An examination of the resulting objective values clarifies this result. For the first solution the objective values are: 32, 47 and 100. These are the best possible values for the first two objectives. The best value for the third objective function is 41, which is the case for the second solution in the Pareto front.

```
1  /* objective functions */
2  min: 15 x2 + 51 x3 + 17 x0 + 25 x1;
3  /* architectural constraints */
4  x0 + x1  = 1;
5  x2 + x3  = 1;
6  /* Resource negotiation */
7  12 x3 + 22 x1 <= 84;
8  12 x2 + 22 x0 <= 48;
9  /* NFP negotiation */
10 97 x2 + 97 x3 >= 52 x0 + 52 x1;
11 /* binary constraint */
12 bin x2, x3, x0, x1;
13 /* MOILP P_1 */
14 19 x0 + 29 x1 + 28 x2 + 39 x3 <= 46; //ORO
15 76 x0 + 17 x1 + 24 x2 + 34 x3 <= 99; //ORO
```

Listing 9.2: P_1 from Example MOILP for a Generated 2x2 System.

The steps performed by the algorithm to derive this Pareto front are the following. As a starting point, the first objective function has been selected for P_0. Solving P_0 returned the solution {x0=1, x1=0, x2=1, x3=0}. Now, P_1 is built resulting in the ILP shown in Listing 9.2. The added constraints are shown on lines 14 and 15. On the left hand side they represent the second and third objective function. On the right hand side the value of these objective functions for the solution derived in P_0 minus $f_k = 1$ is used (i.e., $47 - 1 = 46$ and $100 - 1 = 99$). Solving this ILP actually means solving two ILPs: one with the first added constraint and another one with the second added constraint. They return one solution: {x0=0, x1=1, x2=1, x3=0}, which is added to the Pareto front.

Now, P_2 is derived using the two solutions found until P_1, which leads to the ILP shown in Listing 9.3. Here, two further constraints are added, which again represent the second and third objective function on the left hand side and the value of these functions for the second found solution on the right hand side. This ILP does not provide a new solution. Hence, the algorithm terminates.

9.1.3. The Confidential Sort Example

In the following, the approach presented in this chapter is applied to a confidential sort example. A confidential sort is to be performed whenever the list subject to sorting contains confidential data. This data, in consequence, has to be encrypted. A typical approach to encryption is the use of a pair of public and private keys (e.g., SHA-2 [53]). The bigger the keys used for encryption are, the more effort is required for an attacker to decrypt the data. But, the bigger the keys, the more time is required to encrypt the data. Typically, users intend to get their lists sorted as fast as possible, but encrypted as much as possible. Listing 9.4 shows such a user request for a list with 200.000 elements.

The request is formulated against the structure model shown in Figure 9.1(b). The

```
/* objective functions */
min: 15 x2 + 51 x3 + 17 x0 + 25 x1;
/* architectural constraints */
x0 + x1  = 1;
x2 + x3  = 1;
/* Resource negotiation */
12 x3 + 22 x1 <= 84;
12 x2 + 22 x0 <= 48;
/* NFP negotiation */
97 x2 + 97 x3 >= 52 x0 + 52 x1;
/* binary constraint */
bin x2, x3, x0, x1;
/* MOILP P_1 */
19 x0 + 29 x1 + 28 x2 + 39 x3 <= 46; //OR0
76 x0 + 17 x1 + 24 x2 + 34 x3 <= 99; //OR0
/* MOILP P_2 */
19 x0 + 29 x1 + 28 x2 + 39 x3 <= 56; //OR1
76 x0 + 17 x1 + 24 x2 + 34 x3 <= 40; //OR1
```

Listing 9.3: P_2 from Example MOILP for a Generated 2x2 System.

list, subject to sorting, is encrypted prior to sorting. This way the sort algorithm can be placed on any available machine, even if it is not considered safe (due to its encryption).

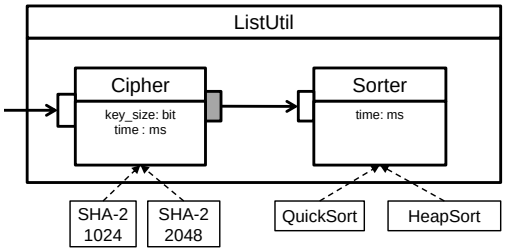
The resulting MOILP is shown in Listing 9.5. The actual values are part of the quality contracts, which are not shown here to keep the example concise. Constraint and variable generation do not differ from the standard ILP approach. The only difference is the presence of two objective functions, which are derived from the two objectives formulated in the user request. Both functions represent the effect of each decision on the respective NFP (response time and size of encryption key used). The variables x_0 and x_1 represent the decision for the SHA-2 algorithm with a 1024 bit key or a 2048 bit key respectively. The variables x_2 and x_3 represent the decision for `QuickSort` or `HeapSort`. To keep the number of variables low, the example comprises only one server.

Listing 9.4: Confidential Sort Request.

```
import ccm [./sort.structure]
target platform [./current.variant]

call ListUtil.sort expecting {
        list_size = 200000
        response_time minimize
        encryption_key_size maximize
}
```

(a)



(b) Referenced Structure Model

Figure 9.1.: Request and Model for the Confidential Sort Example.

```
/* objective functions */
min: 25 x2 + 35 x3 + 180 x0 + 240 x1; //response time
min: -0 x2 - 0 x3  -1024 x0 -2048 x1; //encryption key size
/* architectural constraints */
x0 + x1  = 1;
x2 + x3  = 1;
/* Resource negotiation */
12 x3 + 22 x1 <= 84;
12 x2 + 22 x0 <= 48;
/* NFP negotiation */
97 x2 + 97 x3 >= 52 x0 + 52 x1;
/* binary constraint */
bin x2, x3, x0, x1;
```

Listing 9.5: Generated MOILP for Confidential Sort Example.

The first objective function describes the effect of each decision on the overall response time (w.r.t. a known list, as the MOILP is generated for a specific request). It resamples the fact that `QuickSort` is faster than `HeapSort` and encryption with a 1024 bit key is faster than with a 2048 bit key. The second objective function shows, explicitly, the effect of using a 1024 or a 2048 bit key and, by this, allows to consider the key size as separate dimension of the Pareto front.

9.2. Scalability Evaluation

This section covers a scalability analysis of using Klein and Hannan's approach to MOILP for contract negotiation. As in the previous chapters, generated pipe-and-filter systems of $C \times S$ component types on servers have been used as a basis for measurements. Due to the combinatorial explosion of ILPs to be solved only 2x2 to 30x30 systems have been measured for systems with more than 2 objectives. The bi-objective case has been evaluated with the same 2x2 to 100x100 systems as used for the ILP approach.

Measurements were taken for these 841 systems with 3 and 4 objective functions to analyze the impact of a growing number of objectives. As these objectives are NFPs of interest to the user, 4 is a realistic number of objectives, although in special cases more NFPs might be of interest to the user (e.g., for an expert in audio processing it is likely that more than four NFPs regarding the desired audio file are of his interest). A notable fact about the special case of two objective functions is the lack of *or-constraint*-blocks in the generated ILPs of P_i where $i > 0$. This is because the number of constraints per or-block equals the number of objective functions minus one, i.e., one in the case of two objective functions. In consequence, there is no combinatorial explosion of ILPs to be solved. Instead, as many ILPs are to be solved as solutions are found (and a final ILP, which is infeasible).

The measurements taken for MOILPs were limited to 2 minutes (as for the ILP mea-

surements in Chapter 7). Due to the combinatorial explosion of ILPs to be solved for MOILPs with more than 2 objective functions, a second reason for failing is to be considered: the lack of memory. For all measurements 4 GB of main memory are reserved. But, as will be shown in the following, this does not suffice for MOILPs having a big Pareto front. Thus, a MOILP can fail by timeout (the 2 minute limit) or by running out of memory.

The generation time of MOILPs is similar to the time required to generate single-objective ILPs, because the generation process is exactly the same, except for the generation of multiple objective functions. Hence, we do not separately analyze the generation time, but focus on the overall runtime of the approach.

Figure 9.2 shows the runtime for 2x2 to 100x100 systems with 2 objective functions. As can be seen, the number of servers has a stronger impact than the number of components. The runtime grows very fast per server (approximately 60 seconds already at 5 servers), but slower per additional component (below 1 second until 10 components). Only 27,12% of all MOILPs timed out (2658 of 9801). Notably, no MOILP run out of memory. The mean runtime showed up to be at 36,54 s, the median runtime at 60,12 s only. The 3rd quantile is at 62,92 s. Figure 9.7(a) depicts the runtime as boxplot.

Two further properties are of interest to interpret the behavior of the MOILP solver: the number of ILPs generated for each MOILP and the size of their Pareto front. Fig-

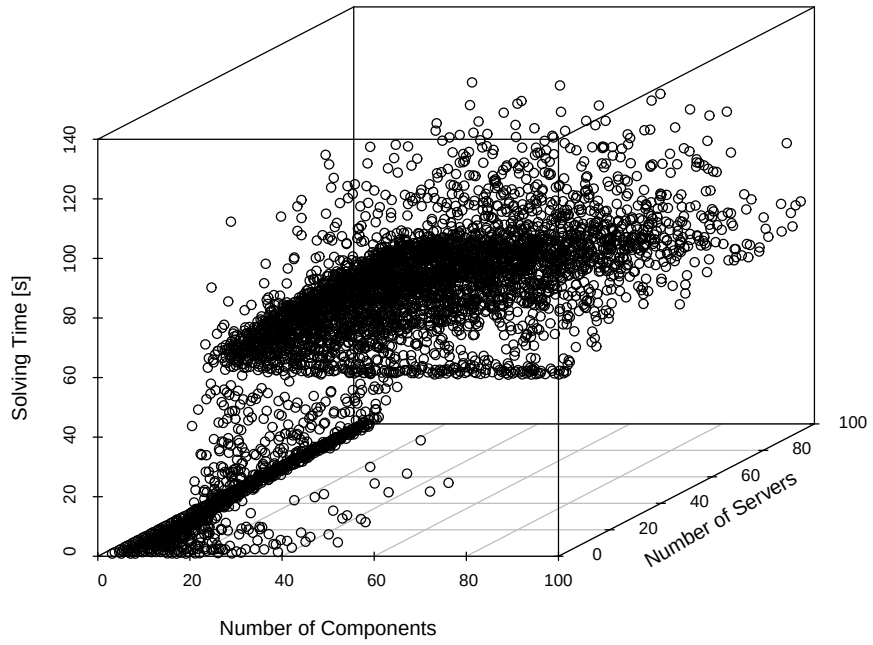
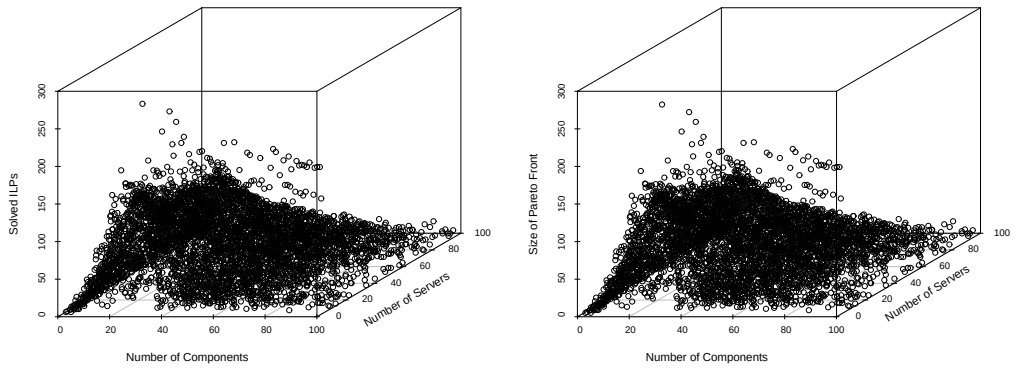
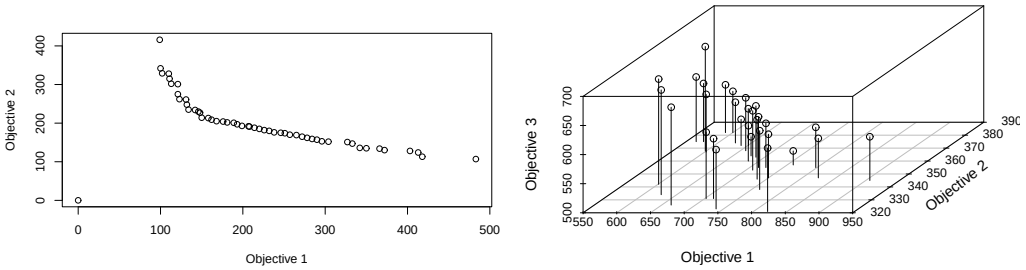


Figure 9.2.: Runtime of MOILP with 2 Objective Functions for 2x2 to 100x100 Systems.



(a) Number of Solved ILPs (b) Size of Pareto Front

Figure 9.3.: Number of Solved ILPs and Size of Pareto Front for 2 Objective MOILPs on 2x2 to 100x100 Systems.



(a) 2-Dimensional Pareto Front of a Generated 10x10 System (b) 3-Dimensional Pareto Front of a Generated 14x5 System

Figure 9.4.: Pareto Front Examples for 2 and 3 Objective MOILPs.

ure 9.3(a) depicts the correlation between the number of components and servers to the number of ILPs to be solved. Figure 9.3(b) in turn depicts the correlation between the number of components and servers to the size of the Pareto front. Obviously both are almost equal, because each solved ILP results in an additional solution in the Pareto front. Small differences occur, as not every ILP could be solved and for each MOILP the last solved ILP was necessarily infeasible.

Another insight from Figure 9.3 is the high number of solutions in the Pareto front for relatively small systems. For systems as small as 10x10 more than 50 solutions are part of the Pareto front. This poses a challenge to user interaction. The question is how to present this high amount of solutions to the user so he can make an educated decision. For 2 objective functions a 2-dimensional representation can be used, which results in a graph as presented in Figure 9.4(a). An example Pareto front for a MOILP with 3

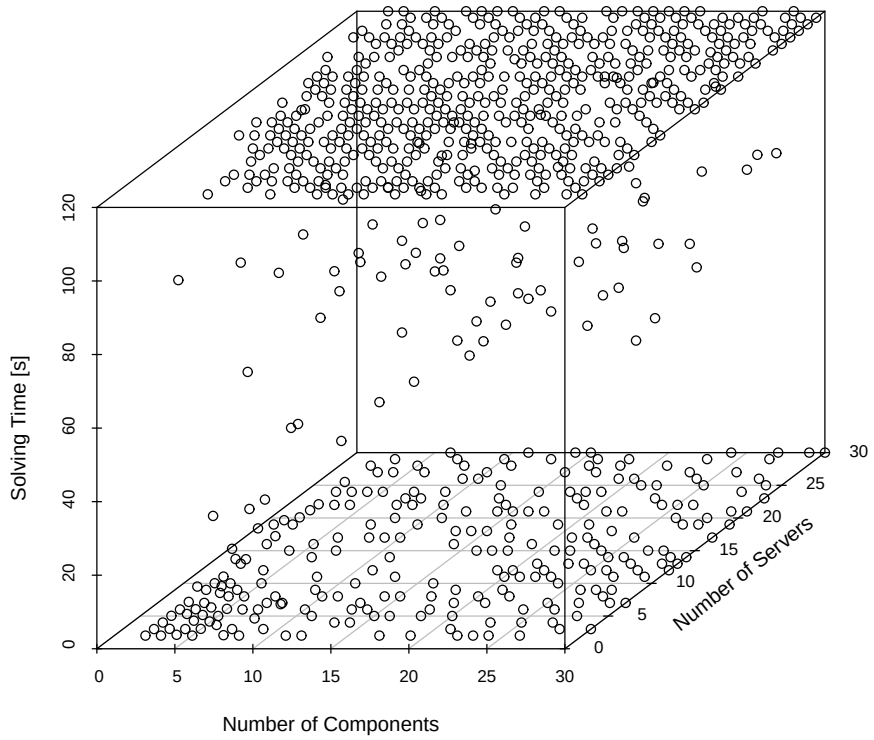
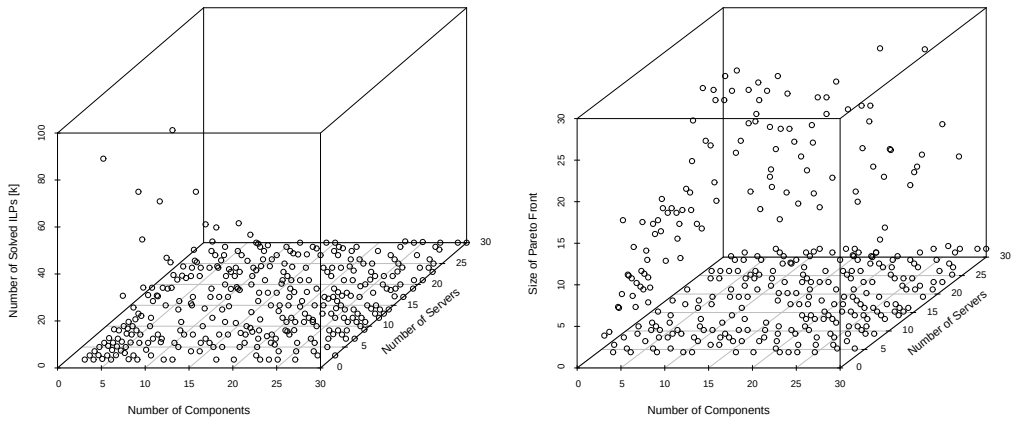


Figure 9.5.: Runtime of MOILP With 3 Objective Functions for 2x2 to 30x30 Systems.

objective functions is shown in Figure 9.4(b). For more than 3 dimensions this graphical representation cannot be used anymore. Hence, another approach is required, but this is out of this thesis' scope.

An investigation of 3 objective MOILPs shows, as to expect, considerably worse performance. In each iteration an additional or block of two constraints is added, leading to a quadratic increase in ILPs to be solved. The boxplot of the runtime is shown in Figure 9.7(b) at the end of this section. With a mean runtime of 79,391 s and its third quantile at the timeout limit of 2 minutes its more than 4 times worse the bi-objective measurements. In contrast to the bi-objective case many more MOILPs timed out and ran out of memory. In total 305 MOILPs (36,27%) timed out and 195 MOILPs (23,19%) ran out of memory, where both sets do not overlap. Hence, for 500 MOILPs (59.45%) no solution could be found. Interestingly, these approx. 60% of failed MOILPs are not concentrated on bigger systems, but range from small to big systems as can be seen in Figure 9.5.

Moreover, Figure 9.5 shows that most MOILPs, which do not timeout or run out of memory are solved very fast as there are more points on the bottom plane than between the bottom and top plane. An investigation of the successful MOILPs shows a



(a) Number of Solved ILPs (b) Size of Pareto Front

Figure 9.6.: Number of Solved ILPs and Pareto Front Size for MOILPs With 3 Objective Functions on 2x2 to 30x30 Systems.

mean runtime of 19,55 s (compared to 20,49 s in the bi-objective scenario) and a third quantile of just 8.3 s. Thus, if the MOILPs do not fail, they perform comparably good. Unfortunately, it is impossible to predict whether a MOILP will fail without solving it. Thus, the high probability of failure revealed in this analysis is a drawback of the approach.

The reason for MOILPs running out of memory is the combinatorial explosion of ILPs to be solved in the presence of more than 2 objective functions. Figure 9.6(a) shows the number of ILPs to be solved for the successful 3-objective MOILPs. In comparison to the bi-objective case, where up to 200 ILPs were to be solved per MOILP, in the 3-objective case up to 82.000 ILPs have to be solved. The mean number of ILPs is at 2.366, the third quantile at 305 ILPs. This drastic increase in ILPs to be solved swiftly exceeds the available memory[1].

Notably, the size of the Pareto fronts in the 3-objective case is very small as Figure 9.6(b) depicts. It comprises only up to 30 solutions. For example, the Pareto front shown in Figure 9.4(b) for a generated 14x5 system comprised only 29 solutions. The reason why the number of ILPs solved and the number of solutions in the Pareto front do not correlate with each other in the 3-objective case are the or-blocks, which lead to many infeasible ILPs per iteration P_i. In the 14x5 system example, the 29 solutions resulted from 5 iterations, but in total 16.423 ILPs were solved to identify these 29 solutions. Again, the size of the systems does not correspond to the number of solutions. Small systems can have comparably large Pareto fronts (e.g., a generated 3x4 system

[1]It does not matter whether they are solved in parallel or not. Nevertheless, parallel execution obviously shortens the total solving time.

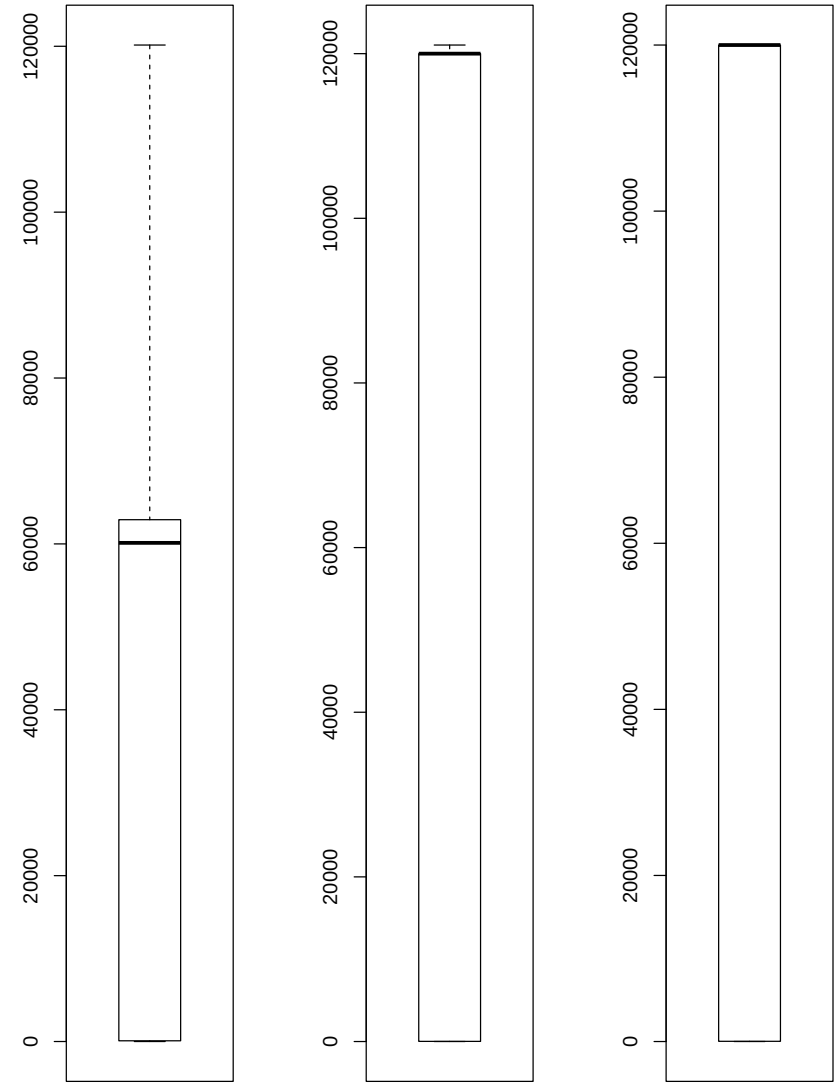
had a Pareto front with 16 solutions) and large systems can have small Pareto fronts (e.g., a generated 29x24 system had a Pareto front with 4 solutions only). An interesting contrast is the larger size of Pareto fronts in the 2-objective case (approx. 200) compared to the 3-objective case (approx. 30). A possible explanation is that the more objective functions exist, the constraint qualifying a solution as being non-dominated (i.e., being part of the Pareto front) is stressed, as such a solution has to be best in all but one objective. The maximum number of solutions in a Pareto front corresponds to the largest objective function (i.e., the objective function comprising the most different values).

Investigating 4-objective MOILPs reveals the expected further decline in performance. Here 399 MOILPs (47,44%) run out of memory and 168 MOILPs (19,98%) timed out, where both sets do not intersect. Thus, in total 567 MOILPs (67,42%) failed. The runtime is depicted as boxplot in Figure 9.7(c). The mean runtime is at 85,174 s, which is only slightly worse than for the 3-objective case (79,391 s). Investigating only successful 4-objective MOILPs shows a mean runtime of only 13,11 s and the third quantile of the runtime is also just at 25,75 s. Nevertheless, as shown, 4-objective MOILPs are more probable to fail than to succeed.

In conclusion, bi-objective MOILP are applicable and feasible for contract negotiation up to 30x30 systems. Unfortunately, using MOILPs with more than two objective functions is more likely to fail than to succeed (the probability of failing is approx. 60% for 3 objectives and approx. 70% for 4 objectives, but only approx. 2% for 2 objectives). Thus, MOILPs with more than two objective functions are applicable, but not feasible for contract negotiation. In consequence, as each objective function represents an NFP of interest to the user, the presented approach allows to feasibly optimize two NFPs concurrently. More NFPs are theoretically possible, but impractical.

9.3. Summary

In this chapter, an a posteriori approach to MOO for contract negotiation has been presented. As the ILP approach presented in Chapter 7 showed good performance, an approach to solve ILPs with multiple objectives was chosen for closer examination. Various such approaches have been developed since the eighties, but only one approach– Klein and Hannan's approach [75] to MOILP–protruded as being promising [109]. The application of Klein and Hannan's approach to contract negotiation has been described in Section 9.1 and was evaluated in Section 9.2. The evaluation showed the applicability of MOILP to contract negotiation and revealed its feasibility for bi-objective MOILPs, but also the infeasibility for MOILPs with more than two objectives. Notably, this assessment is based on the used measurement environment. More powerful resources could render 3- and 4-objective MOILPs feasible in the future.



(a) 2 Objective Functions (b) 3 Objective Functions (c) 4 Objective Functions

Figure 9.7.: Runtime Boxplot of 2 Objective MOILP for 2x2 to 100x100 Systems and of 3 and 4 Objective MOILPs for 2x2 to 30x30 Systems.

10
Conclusion and Future Work

The aim of this thesis is to provide new design and operation principles, namely MQuAT, to software developers for self-optimizing software systems. The central question such a system is meant to answer is:

> How to provide the best possible quality of experience to the user for the least possible cost?

Thus, MQuAT enables software developers to build and run software systems, which automatically adjust themselves to always provide the possible user utility for the least possible cost. To do so, such systems are comprised of multiple variants differing in their non-functional behavior, and are able to reason about the trade offs between the respective NFPs. A key characteristic of MQuAT is the use of QoS contracts to cover the interdependencies between NFPs, the distinction between hard- and software and the use of behavioral models to simulate complex non-functional behavior like energy consumption.

10.1. General Conclusion

In the following, each contribution of the thesis is summarized and pointers are given to the sections covering the respective contribution.

The following three major contributions are comprised in this thesis:

1. A **novel taxonomy of self-optimizing software systems**, highlighting current challenges in this field of research. The taxonomy spans five categories: (1) time of adaptation reasoning, (2) coverage of NFP dependencies, (3) whether an optimal, near-optimal or valid configuration is intended, (4) the number of objectives and (5) adaptivity of reasoning. A literature analysis covering 9 research projects revealed two, yet unaddressed, central challenges. First, the need to cover NFP dependencies using QoS contracts. Second, the need to consider multiple objectives concurrently, i.e., to apply multi-objective optimization (MOO) at runtime.

2. A **development methodology for self-optimizing software systems**, which addresses the first challenge. Using the quality contract language (QCL) and behavioral models, dependencies between NFPs are covered. Additionally, the system's structure is covered and a runtime model, which is kept in synch with the running system, enables predictive reflection on the system.

3. Four evaluated **multi-quality (multi-objective) runtime software optimization approaches**, which address the second challenge. Two a priori methods to MOO are presented in Chapter 7: integer linear programming (ILP) and pseudo-boolean optimization (PBO). Chapter 8 covers an approximate approach, which allows for a priori as well as a posteriori MOO: ant colony optimization (ACO). Finally, Chapter 9 covers an exact a posteriori approach to MOO: multi-objective integer linear programming (MOILP) according to Klein and Hannan [75].

The second and third major contribution are subdivided into more specific minor contributions. In the following, each minor contribution is elaborated and pointers to the sections covering them are given.

The development methodology covers four minor contributions:

- In Section 4.1, the Cool Component Model (CCM) is introduced as a novel self- and context-aware, component-based software architecture for self-optimizing software systems. The CCM comprises structural models to cover the system's architecture, variant models to cover the systems runtime state and behavioral models to cover complex non-functional behavior like energy consumption. The CCM exceeds state of the art in its flexible extensibility, which is due to its 2D metamodelling characteristic.

- In Section 4.2, the Quality Contract Language (QCL) is introduced as an extended contract language covering the complex interdependencies between NFPs as well as their relation to cost and utility. It exceeds state of the art by the novel concept of *quality modes*, which represent levels of user utility.

- In Chapter 5, a process to specify QoS contracts at design time and to approximate their concrete instantiations at deployment time using linear regression is intro-

duced. To the best of my knowledge, no comparable approach for self-optimizing software systems exist.

- Chapter 6 provides an argumentation on how contract negotiation using QoS contracts advances over state of the art and discusses different levels of contract negotiation from plain single-objective contract negotiation to multi-quality, economic contract negotiation. Additionally, a new approach to merge multiple incomparable objectives by discrete event simulation to be used by a priori MOO methods is introduced.

The contributed runtime methods divide into three classes: exact a priori methods to MOO, an approximate method to MOO and an exact a posteriori method to MOO (cf. Section 2.3 for an overview on MOO methods).

- In Section 7.1, an exact runtime a priori MOO approach using integer linear programming is presented.

- In Section 7.2, a second exact runtime a priori MOO approach using pseudo boolean optimization is presented.

- In Section 7.3, both approaches are compared with each other and a scalability analysis of both approaches is performed. The analysis revealed the applicability of ILP for systems of up to 100 component types and servers. Additionally, PBO was shown to be outperformed by ILP.

- In Section 8.1, an approximate runtime MOO approach using ant colony optimization is presented.

- In Section 8.2, a scalability and accuracy analysis of this approach in relation to the previous approaches is performed. The analysis revealed ACO to perform much worse than ILP and PBO and showed the infeasibility of ACO for contract negotiation. The problem of ACO was shown to be the time required to generate the problem graph. Notably, the accuracy of ACO was shown to be very high (mean accuracy of 86,67%).

- In Section 9.1 an exact runtime a posteriori MOO approach using multi-objective ILP is presented.

- Finally, in Section 9.2 a scalability analysis of this approach is performed showing that MOILP performs good for 2 objectives, where it is feasible for systems of up to 80 components and servers. For more than 2 objectives only small systems (up to 10x10) can be handled by MOILP in a feasible time.

10.2. Limitations and Future Work

The thesis at hand provides a broad basis for future research. In the following a set of follow up research questions, based on the limitations of the approaches presented in this thesis, are elaborated.

First, the optimization problem considered in this thesis does only consider a single user. That is, systems are meant to automatically adjust themselves to optimally serve a single user for the least possible cost. For future work, multi-user scenarios have to be investigated. The question is to be adjusted to "how to optimally serve all users over time for the least possible cost". To answer this new question, adjustments are required at both, the design and operation principles, presented in this thesis. The reason is resource contention. Multiple users could be served by the same component and, hence, indirectly share resources. This effect has to be considered by the contract language, the process to create contracts and all optimization approaches.

A further limitation of MQuAT is the need for measurable NFPs. Each NFP need to be quantifiable and measurable. For some NFPs, these properties are hard to realize. For example, usability is a very subjective NFP, which is hard to quantify. Future research could investigate how to quantify such problematic NFPs and how their measurability harmonizes with the MQuAT approach.

For each optimization approach shown in this thesis, a scalability analysis was performed. All these analyses were performed on generated systems, which strictly follow the pipe-and-filter architectural style, because such systems are most likely to benefit from MQuAT. Nevertheless, in the future more complex systems can be treated, which have a different architectural style. Hence, a research question for future work is, how does MQuAT scale for other types of systems.

Yet another limitation of MQuAT is the restriction of the optimization problem to a selection and mapping problem, i.e., which implementations to select and where to map them. Fur future work, an extension to scheduling problems, is valuable. Please note that the omission of scheduling from the optimization problem delivers a strong benefit: the applicability of linear optimization approaches. This is, because a snapshot of the system can be used as basis for the optimization. This snapshot is, consequently, independent from time, whereas scheduling problems obviously are not. The property of linearity is lost as soon as the development of the system of time is to be considered. Nevertheless, the applicability of non-linear MOO approaches to contract negotiation is yet another possible direction for future research.

Finally, further approximate MOO approaches, besides ACO, can be investigated. For example, simulated annealing [73], genetic programming approaches like NSGA-II [44] or evolutionary algorithms like SMEA [34]. A promising approach is to investigate the realization of contract negotiation using a general framework for approximate optimization like Opt4J [87]. Such a framework enables the application of various approximate optimization approaches to a unified problem description.

Appendix

A1. Concrete Syntax of the Quality Contract Language

```
1  SYNTAXDEF qcl
2  FOR <http://www.cool-software.org/qcl> <qcl.genmodel>
3  START QclFile
4
5  IMPORTS {
6          exp : <http://www.cool-software.org/dimm/expressions>
7                  <org.coolsoftware.coolcomponents.expressions/metamodel/expressions.genmodel>
8                  WITH SYNTAX exp <../../org.coolsoftware.coolcomponents.expressions/metamodel/
                       expressions.cs>
9  }
10
11 OPTIONS {
12         reloadGeneratorModel = "true";
13         //overrideManifest = "false";
14         disableDebugSupport = "true";
15         disableLaunchSupport = "true";
16         useClassicPrinter = "true";
17 }
18
19 RULES {
20
21         // Top level container
22         QclFile ::=
23                 imports* !0 contracts* !0;
24
25
26         // Import Statements
27         CcmImport ::=
28                 "import" #1 "ccm" #1 ccmStructuralModel['[', ']'] !0;
29
30
31         // Contracts
32         SWComponentContract ::=
33                 "contract" #1 name[TEXT] #1 "implements" #1 "software" #1 componentType[TEXT] "."
                        port[TEXT] "{" !1
34                         ("metaparameter:" (metaparams #1)*)? !1
35                         (modes !0)+
36                 "}" !0;
37
38         HWComponentContract ::=
39                 "contract" #1 name[TEXT] #1 "implements" #1 "hardware" #1 componentType[TEXT] #1 "{"
                         !1
40                         ("metaparameter:" (metaparams #1)*)? !1
```

```
41                          (modes !0)+
42                  "}";
43
44
45          // Contract Modes
46          SWContractMode ::=
47                  "mode" #1 name[TEXT] #1 "{" !1
48                          (clauses !0)+
49                  "}";
50
51          HWContractMode ::=
52                  "mode" #1 name[TEXT] #1 "{" !1
53                          (clauses !0)+
54                  "}";
55
56
57          // Provision Clauses
58          ProvisionClause ::=
59                  "provides" #1 providedProperty[TEXT] #1
60                          ("min:" #1 minValue:exp.Expression #1)? ("max:" #1 maxValue:exp.Expression
                                #1)? formula?;
61
62
63          // Requirement Clauses
64          SWComponentRequirementClause ::=
65                  "requires" #1 "component" #1 requiredComponentType[TEXT] #1 (("{" !1 (
                        requiredProperties !0)* "}") | ";");
66
67          HWComponentRequirementClause ::=
68                  "requires" #1 "resource" #1 requiredResourceType[TEXT] #1 "{" !1
69                          (requiredProperties !0)+ !1
70                          "energyRate: " energyRate[REAL_LITERAL] !1
71                  "}";
72
73          SelfRequirementClause ::=
74                  "requires" #1 requiredProperty;
75
76          PropertyRequirementClause ::=
77                  requiredProperty[TEXT] #1
78                          ("min:" #1 minValue:exp.Expression #1)? ("max:" #1 maxValue:exp.Expression
                                #1)? formula?;
79
80          FormulaTemplate ::= "<" name[TEXT] "(" (metaparameter[] #1)* ")>";
81
82          Metaparameter ::= name[TEXT];
83  }
```

Listing A1.1: Concrete Syntax of the Quality Contract Language.

List of Figures

1.1. MAPE-K Loop [98]: Monitor - Analyze - Plan - Execute - Knowledge. . . 2
1.2. The Layers of Multi-Quality Auto-Tuning Systems. 7
1.3. Global Picture of MQuAT. 9
1.4. Combined Design- and Runtime Perspective of the Approach. 10
1.5. Structural Overview of the Thesis. 13

2.1. Incarnations of The Feedback Loop for Self-Adaptive Systems. 19
2.2. Hierarchy of Self-* Properties. Redrawn from [112, p.5]. 20
2.3. Graphical Representation of an Exemplary Linear Program and the Applied Simplex Approach. 27
2.4. Approaches to Solve Integer Linear Programs. 28
2.5. Concepts of an Ant Colony Optimization Framework. 30
2.6. The Shortest Path Problem in Ant Colony Optimization. 31

3.1. Comparison Criteria for Related Work. 36

4.1. Constituents of the Multi-Quality-aware Software Architecture. 56
4.2. Concepts and their Relations in the Structure Package of the CCM. . . . 58
4.3. Concepts and their Relations in the Variant Package of the CCM. 61
4.4. Example of Structure and Variant Model. 62
4.5. Exemplary Energy State Chart for CPUs. 63
4.6. Concepts and their Relations in the Behavior Package of the CCM. 64
4.7. Concepts and their Relations in the Expressions Package of the CCM. . . 66
4.8. Concepts and their Relations in the Datatypes Package of the CCM. . . . 67
4.9. Concepts and their Relations in the Units Package of the CCM. 68
4.10. Concepts and their Relations of the Request Language of the CCM. . . . 70
4.11. Concepts and their Relations of the Reconfiguration Language of the CCM. 71
4.12. Concepts and their Relations of the Workload Language of the CCM. . . 72
4.13. Layering of NFPs. From Quality to User Utility. 74
4.14. Concepts and their Relationships of the QCL. 77
4.15. General Architecture for Multi-Objective Optimization in MQuAT. 79

5.1. Process of Contract Creation. 83

5.2. Approximated Function for Data from Table 5.1. 85
5.3. Measurements (Dots) and Regression Function (Line) for HeapSort. . . . 88

6.1. From Contract Checking to Economic Multi-QoS Contract Negotiation. . 94
6.2. Example Request to Sort a List. 95
6.3. Architecture of Non-intrusive Alzheimer Disease Detection Application. . 97
6.4. Annotation of Sediments in Brain Slices Indicating Alzheimer's Disease. . 98
6.5. Valid Contract Paths for a Sample User Request. Depicted as Flow of Demands (Left) and as Directed Graph (Right). 100

7.1. Overview of ILP Generation. 110
7.2. List Generation, Sorting and Filter Example. 117
7.3. Structural Model Specifying Types of a Server Landscape. 127
7.4. Parameters of Generated Pipe-and-Filter Architectures. 129
7.5. Boxplots of Generation and Solving Time for the ILP Solution. 131
7.6. ILP Solving Time in Relation to Number of Components. 133
7.7. Boxplots of Generation and Solving Time for the PBO Solution. 134
7.8. PBO Solving Time in Relation to Number of Components. 137
7.9. Percentage of Determined Solutions in 2 Minutes for ILP and PBO Approach. 138

8.1. Contract Negotiation by Ant Colony Optimization for a Single Component Type. 141
8.2. Contract Negotiation by Ant Colony Optimization for Multiple Component Types. 142
8.3. Generated Ant Colony for List Generated/Sort/Filter Example. 146
8.4. Accuracy of ACO Solutions with an Evaporation Rate $\rho = 0.1$. 147
8.5. Runtime of ACO Approach for 100 Ants, 100 Iterations and $\rho = 0.1$. . . . 148
8.6. Accuracy of ACO Approach in Correlation to System Size (for 100 Ants, 100 Iterations and $\rho = 0.1$). 149
8.7. Boxplots of Graph Size and Generation as well as Processing Time. 150

9.1. Request and Model for the Confidential Sort Example. 157
9.2. Runtime of MOILP with 2 Objective Functions for 2x2 to 100x100 Systems. 159
9.3. Number of Solved ILPs and Size of Pareto Front for 2 Objective MOILPs on 2x2 to 100x100 Systems. 160
9.4. Pareto Front Examples for 2 and 3 Objective MOILPs. 160
9.5. Runtime of MOILP With 3 Objective Functions for 2x2 to 30x30 Systems. 161
9.6. Number of Solved ILPs and Pareto Front Size for MOILPs With 3 Objective Functions on 2x2 to 30x30 Systems. 162
9.7. Runtime Boxplot of 2 Objective MOILP for 2x2 to 100x100 Systems and of 3 and 4 Objective MOILPs for 2x2 to 30x30 Systems. 164

List of Tables

2.1. Comparison of Self-adaptive Systems and Auto-Tuning. 24

3.1. Comparison with Related Work. 39

5.1. CPU data (mean ($\overline{cpu\_time}$), standard deviation (σ_{cpu_time}) and standard error of the mean ($SE_{\overline{cpu\_time}}$)) for HeapSort example. 86

6.1. All Possible Mappings for Quality Path (PNG.full_size, Hagen.full_size, ConstCorrection.fast) onto Two Servers. Invalid Mappings are Shadowed. 101

List of Listings

4.1. Example Unit Specification. 69
4.2. Example Request to Sort a List. 70
4.3. Example Reconfiguration Script. 71
4.4. Example Contract for an "ImageViewer" Implementation of Presenter Component. 75
4.5. Example Request with Multiple Objectives. 78

5.1. QCL Contract Template for a Sort Component. 84
5.2. Example QCL Contract for a Sort Component. 87
5.3. User Request-Specific QCL Contract. 89

6.1. Request to Sort a List. 95
6.2. Example Request to Alzheimer Detection Application. 98
6.3. QCL Contract of PNG Output Postprocessor. 99

7.1. User Request on Filter Component. 117
7.2. Contract for "Random" Implementation of List Generator Component. . . 118
7.3. Contract for "QuickSort" and "HeapSort" Implementation of Sort Component. 119
7.4. Contract for "BoundedFilter" Implementation of Filter Component. 120
7.5. Generated ILP for List Generator/Sort/Filter Example. 121
7.6. Generated PBO for List Generator/Sort/Filter Example. 125

9.1. An Example MOILP for a Generated 2x2 System. 155
9.2. P_1 from Example MOILP for a Generated 2x2 System. 156
9.3. P_2 from Example MOILP for a Generated 2x2 System. 157
9.4. Confidential Sort Request. 157
9.5. Generated MOILP for Confidential Sort Example. 158

A1.1. Concrete Syntax of the Quality Contract Language. 169

List of Abbreviations

ACO	Ant Colony Optimization.
ACPI	Advanced Configuration and Power Interface.
AEOS	Automated Empirical Optimization of Software.
ATLAS	Automatically Tuned Linear Algebra Software.
CBSD	Component-Based Software Development.
CCM	Cool Component Model.
CESAR	Cost-Efficient methods and processes for SAfety Relevant embedded systems.
COMQUAD	COMponents with QUantitative properties and ADaptivity.
CPU	central processing unit.
CSL	Contract Specification Language.
CSM	Common System Meta-Model.
DAG	Direct Acyclic Graph.
DARPA	Defense Advanced Research Projects Agency.
DES	Discrete Event Simulator.
DICOM	Digital Imaging and Communications in Medicine.
DIVA	DynamIc Variability in complex, Adaptive systems.
ESC	Energy State Chart.
GQM	Global Quality Manager.
GRM	Global Resource Manager.
GUM	Global User Manager.
HDD	hard disk drive.
HPC	High-Performance Computing.

HRC	Heterogeneous Rich Components.
ILP	Integer Linear Programming.
ITK	Insight Segmentation and Registration Toolkit.
LQM	Local Quality Manager.
LRM	Local Resource Manager.
M@RT	Models @ Runtime.
MADAM	Mobility and ADAptation enabling Middleware.
MDA	Model-Driven Architecture.
MDSD	Model-Driven Software Development.
MILP	Mixed Integer Linear Programming.
MOILP	Multi-Objective Integer Linear Programming.
MOO	Multi-objective Optimization.
MQuAT	Multi-Quality Auto Tuning.
MRT	Magnetic Resonance Tomography.
MUSIC	Self-Adapting Applications for Mobile USers In Ubiquitous Computing Environments.
NFP	Non-functional Property.
OCL	Object Constraint Language.
OS	Operating System.
PBO	Pseudo-Boolean Optimization.
PEPPHER	Performance Portability and Programmability for Heterogeneous Many-core Architectures.
QCL	Quality Contract Language.
QoS	Quality of Service.
QoSMOS	QoS Management and Optimisation in Service-Based Systems.
RAM	random access memory.

SAS	Self-adaptive Software.
SPEEDS	SPEculative and Exploratory Design in Systems Engineering.
THEATRE	The Auto-Tuning Runtime Environment.
WCET	Worst Case Execution Time.
WLAN	Wireless Local Area Network.
XML	Extensible Markup Language.

Bibliography

[1] J. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[2] I. Alaya, C. Solnon, and K. Ghedira. Ant colony optimization for multi-objective optimization problems. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence - Volume 01*, pages 450–457. IEEE Computer Society, 2007.

[3] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *Proceedings of Model-based Methodologies for Pervasive and Embedded Software (MOMPES), Workshop at ICSE*, pages 61–71. ACM and IEEE Digital Libraries, 2009.

[4] A. Aleti, L. Grunske, I. Meedeniya, and I. Moser. Let the ants deploy your software - an ACO based deployment optimisation strategy. In *Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 505–509. ACM and IEEE Digital Libraries, 2009.

[5] M. Alia, G. Horn, F. Eliassen, M. Khan, R. Fricke, and R. Reichle. A component-based planning framework for adaptive systems. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *LNCS*, pages 1686–1704. Springer, 2006.

[6] U. Aßmann, S. Götz, J.-M. Jézéquel, B. Morin, and M. Trapp. *State-of-the-Art Survey Volume on Models@run.time*, chapter Uses and Purposes of M@RT Systems. Springer LNCS, 2013.

[7] C. Atkinson and T. Kühne. The essence of multilevel metamodeling. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 19–33. Springer, 2001.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[9] J. Banks and J. Carson. *Discrete event system simulation*. Prentice Hall, Englewood Cliffs, NJ 07632, 1984.

[10] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[11] P. Barth. *Logic-Based 0-1 Constraint Programming*. Springer, 1996.

[12] A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, and R. Weber. A model-based design methodology with contracts to enhance the development process of safety-critical systems. In S. Min, R. Pettit, P. Puschner, and T. Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *LNCS*, pages 59–70. Springer, 2011.

[13] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[14] L. Benini, R. Hodgson, and P. Siegel. System-level power estimation and optimization. In *Proceedings of the 1998 international symposium on low power electronics and design*, ISLPED '98, pages 173–178. ACM, 1998.

[15] S. Benkner, S. Pllana, J. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, Sept./Oct. 2011.

[16] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 5382 of *LNCS*, pages 200–225. Springer, 2008.

[17] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Contract aware components, 10 years after. In C. Cámara and Salaün, editors, *Electronic Proceedings of Theoretical Computer Science*, volume 37, 2010.

[18] J. Bilmes, K. Asanovicy, C.-W. Chinz, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Super Computing*, pages 340–347. ACM, 1997.

[19] G. Bitran. Linear multiple objective programs with zero-one variables . *Mathematical Programming*, 13:121–139, 1977.

[20] G. Bitran. Theory and algorithms for linear multiple objective programs with zero-one variables. *Mathematical Programming*, 17:362–390, 1979.

[21] E. Boros and P. L. Hammer. Pseudo-boolean optimization. *Discrete Applied Mathematics*, 123(1–3):155–225, November 2002.

[22] C. Boutilier, R. Das, J. O. Kephart, and W. E. Walsh. Towards cooperative negotiation for decentralized resource allocation in autonomic computing systems. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1458–1459. Morgan Kaufmann Publishers Inc., 2003.

[23] P. Bratskas, N. Paspallis, K. Kakousis, and G. A. Papadopoulos. Applying utility functions to adaptation planning for home automation applications. In G. A. Papadopoulos, W. Wojtkowski, G. Wojtkowski, S. Wrycza, and J. Zupancic, editors, *Information Systems Development*, pages 529–537. Springer, 2010.

[24] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report, W3C, November 2008.

[25] D. Brodowski. CPU frequency and voltage scaling code in the linux(tm) kernel. http://www.kernel.org/doc/Documentation/cpu-freq/.

[26] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12):1257–1284, 2003.

[27] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *Software Maintenance and Evolution: Research and Practice*, 17:309–332, September 2005.

[28] C. Bürger, S. Karol, C. Wende, and U. Aßmann. Reference attribute grammars for metamodel semantics. In B. Malloy, S. Staab, and M. Brand, editors, *Software Language Engineering*, volume 6563 of *LNCS*, pages 22–41. Springer, 2011.

[29] R. Burkard, J. Krarup, and P. Pruza. Efficiency and optimality in minisum, minimax 0-1 programming problems. *Journal of the Operational Research Society*, 33:137–151, 1982.

[30] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Transactions on Software Engineering*, 37(3):387–409, May–June 2011.

[31] J. M. Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008.

[32] H. Chang and P. Collet. Fine-grained contract negotiation for hierarchical software components. In *Proceedings of 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 28–35. IEEE, 2005.

[33] H. Chang, P. Collet, A. Ozanne, and N. Rivierre. From components to autonomic elements using negotiable contracts. In L. Yang, H. Jin, J. Ma, and T. Ungerer, editors, *Autonomic and Trusted Computing*, volume 4158 of *LNCS*, pages 78–89. Springer, 2006.

[34] J. Chen, K. Lin, and C. Zhou. The strength mutation evolutionary algorithm and its application in multi-object optimization. In *Proceedings of 4th International Conference on Natural Computation*, volume 1, pages 681–685. IEEE, 2008.

[35] S.-W. Cheng, D. Garlan, and B. R. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2006.

[36] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[37] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.

[38] P. Collet. Taming complexity of large software systems: Contracting, self-adaptation and feature modeling. Habilitation, December 2011.

[39] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[40] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32. ACM, 2011.

[41] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[42] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[43] A. de Roo, H. Sözer, and M. Aksit. An architectural style for optimizing system qualities in adaptive embedded systems using multi-objective optimization. In *Proceedings of Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA)*, pages 349–352. IEEE, 2009.

[44] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[45] R. Deckro and E. Winkofsky. Solving zero-one multiple objective programs through implicit enumeration. *European Journal of Operational Research*, 12(4):362–374, 1983.

[46] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5:4–7, 2001.

[47] E. W. Dijkstra. *On the Role of Scientific Thought*, pages 60–66. Springer, 1982.

[48] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, 2006.

[49] K. Doerner, W. J. Gutjahr, R. F. Hartl, C. Strauss, and C. Stummer. Pareto ant colony optimization: A metaheuristic approach to multiobjective portfolio selection. *Annals of Operations Research*, 131:79–99, Oct. 2004.

[50] M. Dorigo and G. Di Caro. *New ideas in optimization*, chapter The ant colony optimization meta-heuristic, pages 11–32. McGraw-Hill Ltd., 1999.

[51] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(1):1–13, 1996.

[52] B. L. Duc, P. Collet, J. Malenfant, and N. Rivierre. A QoI-aware framework for adaptive monitoring. In *Proceedings of 2nd International Conference on Adaptive and Self-adaptive Systems and Applications, ADAPTIVE 2010*, pages 133–141. IEEE, 2010.

[53] D. Eastlake and T. Hansen. RFC 6234: US secure hash algorithms (SHA and SHA-based HMAC and HKDF), 2011.

[54] K. Eikland and P. Notebaert. LP Solve 5.5 reference guide. http://lpsolve.sourceforge.net/5.5/ (access on 26.11.2012).

[55] M. Ericsson, C. Kessler, W. Löwe, and J. Andersson. Composition and optimization. *Proceedings of the conference on component-based high performance computing (CBHPC'08)*, 45(4):591–610, 2008.

[56] A. Finkelstein and J. Kramer. Software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 3–22. ACM, 2000.

[57] F. Fleurey and A. Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *Model Driven Engineering Languages and Systems (MODELS2009)*, volume 5795 of *LNCS*, pages 606–621. Springer, 2009.

[58] M. Förster, B. Bickel, B. Hardung, and G. Kokai. Self-adaptive ant colony optimisation applied to function allocation in vehicle networks. In *Proceedings of 9th annual conference on Genetic and evolutionary computation*, pages 1991–1998. ACM, 2007.

[59] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceddings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[60] S. Götz, C. Wilke, S. Cech, and U. Aßmann. Runtime variability management for energy-efficient software by contract negotiation. In *Proceedings of the 6th International Workshop Models@run.time, MRT'11*, pages 61–72, 2011.

[61] S. Götz, C. Wilke, S. Cech, and U. Aßmann. *Sustainable ICTs and Management Systems for Green Computing*, chapter Architecture and Mechanisms for Energy Auto Tuning, pages 45–73. IGI Global, June 2012.

[62] S. Götz, C. Wilke, S. Richly, and U. Aßmann. Approximating quality contracts for energy auto-tuning software. In *Proceedings of First International Workshop on Green and Sustainable Software (GREENS 2012)*, pages 8–14. IEEE, 2012.

[63] S. Götz, C. Wilke, M. Schmidt, and S. Cech. THEATRE resource manager interface specification. Technical Report TUD-FI10-08, Technische Universität Dreden, Dresden, Germany, 2010.

[64] S. Götz, C. Wilke, M. Schmidt, S. Cech, and U. Aßmann. Towards energy auto tuning. In *Proceedings of First Annual International Conference on Green Information Technology (GREEN IT)*, pages 122–129. GSTF, 2010.

[65] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[66] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning (2nd edition)*. Springer, Feb. 2009.

[67] N. Heinen. Alzheimer früher erkennen. *Technology Review*, 12:52–55, Dec. 2012.

[68] D. Jackson. *Software Abstractions. Logic, Language, and Analysis.* MIT Press, revised edition, 2012.

[69] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, The Object Management Group, June 2003.

[70] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[71] M. U. Khan. *Unanticipated Dynamic Adaptation of Mobile Applications*. PhD thesis, Universität Kassel, March 2010.

[72] M. U. Khan, R. Reichle, M. Wagner, K. Geihs, U. Scholz, C. Kakousis, and G. A. Papadopoulos. An adaptation reasoning approach for large scale component-based applications. *Electronic Communications of the EASST*, 19:1–13, 2009.

[73] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[74] G. Kiziltan and E. Yucaoglu. An algorithm for multiobjective zero-one linear programming. *Management Science*, 29(12):1444–1453., 1983.

[75] D. Klein and E. Hannan. An algorithm for the multiple objective integer linear programming problem. *European Journal of Operational Research*, 9(4):378–385, 1982.

[76] R. Klein, M. Buchheit, and W. Nutt. Configuration as model construction: The constructive problem solving approach. In *Proceedings of the 3rd International Conference on Artificial Intelligence in Design*, pages 201–218. Springer, 1994.

[77] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[78] A. Koziolek. *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. PhD thesis, Karlsruher Institut für Technologie, 2011.

[79] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Proceedings of Future of Software Engineering*, pages 259–268. IEEE, 2007.

[80] J. Krarup and P. Pruzan. Reducibility of minimax to minisum 0-1 programming problems. *European Journal of Operational Research*, 6(2):125–132., 1981.

[81] R. Laddaga. Darpa self adaptive software broad agency announcement (baa) 98-12 proposer information pamphlet - excerpt. http://people.csail.mit.edu/rladdaga/BAA98-12excerpt.html, Dec 1998.

[82] S. M. Lee. *Goal programming for decision analysis*. Auerbach Publishers, 1972.

[83] G. Leguizamon and Z. Michalewicz. A new version of ant system for subset problems. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1458–1464. IEEE, 1999.

[84] L. Liberti. Linear programming. http://www.lix.polytechnique.fr/ liberti/ teaching/dix/inf572-09/linear_programming.pdf, October 2010.

[85] M. Lopez-Ibanez, L. Paquete, and T. Stützle. On the design of ACO for the biobjective quadratic assignment problem. In *Proceedings of ANTS'04*, volume 3172 of LNCS, pages 214–225. Springer, 2004.

[86] C. Lucas. Multidimensional economics, March 1999. http://www.calresco.org/lucas/economic.htm (accessed 12.02.2013).

[87] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, pages 1723–1730, Dublin, Ireland, 2011.

[88] R. Marler and J. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[89] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[90] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. MATE: Monitoring, analysis and tuning environment for parallel/distributed applications. *Concurrency and Computation: Practice and Experience*, 19(11):1517–1531, 2007.

[91] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[92] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jézéquel. On executable meta-languages applied to model transformations. In *Proceedings of Model Transformations In Practice Workshop*, 2005.

[93] M. Mulugeta. *QoS Contract Negotation in Distributed Component-Based Software*. PhD thesis, Technische Universität Dresden, 2007.

[94] M. Mulugeta and A. Schill. Component QoS contract negotiation in multiple containers. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *LNCS*, pages 1650–1667. Springer, 2006.

[95] M. Mulugeta and A. Schill. A framework for QoS contract negotiation in component-based applications. In B. Meyer, J. Nawrocki, and B. Walter, editors, *Balancing Agility and Formalism in Software Engineering*, volume 5082 of *LNCS*, pages 238–251. Springer, 2008.

[96] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley Interscience, 1999.

[97] T. Nowicki, M. Squillante, and C. Wu. Fundamentals of dynamic decentralized optimization in autonomic computing systems. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *LNCS*, pages 366–366. Springer, 2005.

[98] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62, May 1999.

[99] J. Ostergaard. Discrete optimization of the sparse QR factorization. http://unthought.net/OptimQR/OptimQR/report.html, Oct 1998.

[100] J. Pan, S. Staab, U. Aßmann, J. Ebert, and Y. Zhao. *Ontology-Driven Software Development*. Springer, 2013.

[101] V. Pareto. *Manuale di Economica Politica, Societa Editrice Libraria. Milan; translated into English by A.S. Schwier as Manual of Political Economy in 1971*. A.M. Kelley, New York, 1906.

[102] C. Pasternak and U. Passy. *Bicriterion Mathematical Programs with Boolean Variables*. Operations research, statistics and economics mimeograph series. Technion - Israel Institute of Technology, Faculty of Industrial and Management Engineering, 1972.

[103] R. Patrascu, C. Boutilier, R. Das, J. O. Kephart, G. Tesauro, and W. E. Walsh. New approaches to optimization and utility elicitation in autonomic computing. In *Proceedings of the National Conference on Artificial Intelligence*, pages 140–145. AAAI Press, 2005.

[104] V. Poladian, S. Butler, M. Shaw, and D. Garlan. Time is not money: The case for multi-dimensional accounting in value-based software engineering. In *Proceedings of Fifth Workshop on Economics-Driven Software Engineering Research (EDSER-5)*, 2003.

[105] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 604–613. IEEE Computer Society, 2004.

[106] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[107] S. Quinton, S. Graf, and R. Passerone. Contract-based reasoning for component systems with complex interactions. Technical Report TR-2010-12, Verimag Research Laboratory, Grenoble, France, May 2010.

[108] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[109] L. M. Rasmussen. Zero-one programming with multiple criteria. *European Journal of Operational Research*, 26:83–95, 1986.

[110] R. Ribler, J. Vetter, H. Simitci, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, pages 172–179, 1998.

[111] S. Röttger and S. Zschaler. CQML+: Enhancements to CQML. In *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56. Cépadués-Éditions, 2003.

[112] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM TAAS*, 4:14:1–14:42, May 2009.

[113] M. Salukvadze. *Vector-Valued Optimization Problems in Control Theory*. Academic Press, 1979.

[114] M. Sandrieser, S. Benkner, and S. Pllana. Improving programmability of heterogeneous many-core systems via explicit platform descriptions. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 17–24. ACM, 2011.

[115] M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[116] C. Seo, S. Malek, and N. Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the 22nd IEEE/ACM international conference on automated software engineering*, ASE '07, pages 421–424. ACM, 2007.

[117] C. Szyperski. *Component Software: Beyond Object-Oriented Programming (ACM Press)*. Addison-Wesley Professional, December 1997.

[118] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 57–69. IEEE, 2005.

[119] A. Tiwari and J. Hollingsworth. Online adaptive code generation and tuning. In *Proceedings of 2011 International Symposium on Parallel Distributed Processing (IPDPS)*, pages 879–892, 2011.

[120] H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar. 2002.

[121] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-Driven Software Development*. John Wiley & Sons, 2006.

[122] M. J. Voss and R. Eigemann. High-level adaptive program optimization with adapt. In *Proceedings of the eighth ACM SIGPLAN symposium on principles and practices of parallel programming*, PPoPP '01, pages 93–102. ACM, 2001.

[123] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. 16:521–530, 2005.

[124] W. Walsh, G. Tesauro, J. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of the International Conference on Autonomic Computing*, pages 70–77. IEEE, 2004.

[125] J. Warmer and A. Kleppe. *Object Constraint Language 2.0*. Addison Wesley Professional, 2004.

[126] G. Weiss, K. Becker, B. Kamphausen, A. Radermacher, and S. Gerard. Model-driven development of self-describing components for self-adaptive distributed embedded systems. In *Proceedings of Euromicro Conference on Software Engineering and Advanced Applications*, pages 477–484. IEEE, 2011.

[127] G. Weiss, K. Becker, A. Radermacher, and S. Gerard. Rt-describe: Self-describing components for self-adaptive distributed embedded systems. In *Proceedings of 3rd Workshop on Adaptive and Reconfiurable Embedded Systems (APRES)*, pages 1–4. Fraunhofer ESK, 2011.

[128] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[129] T. White, S. Kaegi, and T. Oda. Revisiting elitism in ant colony optimization. In *Proceedings of Genetic and Evolutionary Computation Conference*, volume 2723 of *LNCS*, pages 122–133. Springer, 2003.

[130] P. L. Yu. A class of solutions for group decision problems. *Management Science*, 19(8):936–946, Apr. 1973.

[131] M. Zeller, C. Prehofer, G. Weiss, D. Eilers, and R. Knorr. Towards self-adaptation in real-time, networked systems: Efficient solving of system constraints for automotive embedded systems. In *Proceedings of Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 79–88. IEEE, 2011.

[132] A. Zimmermann, A. Lorenz, and R. Oppermann. An operational definition of context. In B. Kokinov, D. Richardson, T. Roth-Berghofer, and L. Vieu, editors, *Modeling and Using Context*, volume 4635 of *LNCS*, pages 558–571. Springer, 2007.