Reusability Evaluation of Component-Based Embedded Automotive Software Systems

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: Dipl.-Inf.(FH) Martin Hobelsberger
geb. am 02.08.1981 in Freyung

Gutachter:
Prof. Dr. habil. Reiner Dumke
Prof. Dr. Christian Wolff
Prof. Dr. Jürgen Mottok

Ort und Datum des Promotionskolloquiums: Magdeburg, den 15.05.2012

Abstract

The most important drivers of innovation in modern cars are embedded systems. The design and development of these systems have become, as technology progresses, more and more complex and challenging. In order to meet these challenges, component-based architectures were introduced to automotive embedded systems. Reusability is an important aspect in the development of these systems and is frequently seen as a powerful approach to develop high quality systems while reducing complexity.

This thesis presents novel and efficient techniques for the evaluation and optimization of the reuse of component-based embedded automotive systems. Two problems are addressed: component reuse evaluation and component package generation with reuse optimization. With component reuse evaluation, individual components are measured by their contribution to reuse. This technique applies network analysis methods, derived from the research field of social network analysis and graph theory. Focusing on the connections and relationships between components and component groups, important and central individual components are identified and clustered to reusable packages.

For large project repositories, the feasibility of the network analysis concepts might be impeded due to very dense graphs. In such cases, the system architect has to resort to an optimization heuristic, the second component reuse clustering technique, proposed in this thesis. This optimization approach, based on simulated annealing, selects and groups components in order to optimize reuse in the component-based software architecture.

Both techniques work with information on system components which is automatically collected from a company's project repository. This information represents the knowledge, experience and design decisions of system architects of past and current projects and is transformed to a graph based structure.

To integrate the evaluation and clustering techniques in a company's development process and to support the handling (e.g. storing and retrieval) of individual components a model-based architecture framework has been proposed.

Experimental results on real-life project repositories have proven and demonstrated the feasibility of the proposed techniques.

Kurzfassung

Eingebettete Software Systeme sind derzeit eine treibende Kraft von Produktinnovationen in der Automobilindustrie. Der Entwurf und die Entwicklung dieser Systeme werden mit bestehendem technologischem Fortschritt immer komplexer und führen zu neuen Herausforderungen. Um sich diesen Herausforderungen in der Automobilindustrie stellen zu können wurden komponenten-basierte Softwarearchitekturen eingeführt. Die Wiederverwendbarkeit ist ein wichtiger Aspekt bei der Entwicklung dieser Systeme und wird als leistungsfähiger Ansatz zur Entwicklung von hochqualitativen Systemen bei gleichzeitiger Reduzierung der Komplexität und der Entwicklungskosten gesehen.

In dieser Arbeit werden neue und effiziente Methoden zur Evaluation und Optimierung der Wiederverwendbarkeit dieser Systeme vorgestellt. Bei der Evaluierung werden individuelle Komponenten hinsichtlich ihres Beitrags zur Wiederverwendung gemessen. Hierzu werden Methoden aus der Netzwerkanalyse, abgeleitet aus dem Forschungsgebiet der sozialen Netzwerkanalyse und der Graphen Theorie, angewendet. Durch den Fokus auf die Beziehungen zwischen Komponenten und Komponentengruppen können zentrale Komponenten identifiziert und zu wiederverwendbaren Komponentenpaketen gruppiert werden.

Bei sehr großen Projektdatenbanken kann die Netzwerkanalyse, aufgrund sehr dichter Graphen, nur eingeschränkt angewendet werden. In diesem Fall kann der Systemarchitekt auf eine, in dieser Arbeit vorgestellte, Optimierungsheuristik zurückgreifen. Dieser Optimierungsansatz, basierend auf dem Prinzip von *Simulated Annealing*, wählt Komponenten aus und gruppiert diese mit dem Ziel, die Wiederverwendbarkeit der Architektur zu optimieren.

Als Datenquelle für beide Methoden dient eine Projektdatenbank aus der automatisiert Informationen erhoben werden. Diese Informationen repräsentieren das Wissen, die Erfahrung und die Entwurfsentscheidungen von Systemarchitekten über abgeschlossene und aktuelle Projekte und werden in die Struktur eines Graphen transformiert. Um die vorgestellten Methoden in den Entwicklungsprozess eines Unternehmens zu integrieren und um die Anwendung (z.B. das Ablegen und Wiederfinden von Komponenten) zu unterstützen wurde ein Modellbasiertes Architektur-Framework vorgeschlagen. Experimentelle Ergebnisse auf Basis von Projektdatenbanken aus der Industrie demonstrieren eine erfolgreiche Anwendbarkeit der vorgestellten Techniken und Methoden.

Acknowledgements

This work is part of the BMBF founded research project DynaS^3 ("Dynamic software architectures in electronic control units in automotive systems with consideration of requirements for functional safety") and is supported by the FHprofUnd program of the German Federal Ministry of Education and Research (FKZ 1752X07). The main target of this work is to support the process of design and evaluation of efficient dynamic software architectures in the automotive domain. It is part of a cooperation between the University of Applied Sciences Regensburg, the Otto-von-Guericke University Magdeburg and the Continental Automotive GmbH - Powertrain Division.

Many people have either directly and indirectly contributed to this thesis. I would like to take the opportunity to thank all of them for their support. First and foremost, I would like to sincerely thank my supervisors Prof. Dr. Jürgen Mottok and Prof. Dr. Reiner Dumke for their guidance, invaluable suggestions, continued encouragement and support during these years. I am also very grateful to Prof. Dr. Christian Wolff for his review and expert opinions on the thesis at hand. I am deeply grateful to my advisors at the Continental Automotive GmbH, Prof. Dr. Michael Niemetz and Stefan Kuntz. I will particularly remember our valuable, inspiring and lively discussions at our regular meetings. Special thanks go to my former colleagues at the University of Applied Sciences Regensburg, Dr. Michael Deubzer, Michael Schorer and Michael Steindl for their friendship, encouragement and helpful advice over the years. I will never forget the time we spent on joint research and conference travels, as well as the fruitful discussions and the social activities in the evenings.

Finally, my very special thanks belong to Fritz, Gerlinde, and Stefan Hobelsberger and Beata Kling for their warm support and for always believing in me all these years.

Contents

List of Tables viii

List of Figures x

List of Algorithms xii

Abbrevations xv

I. Preliminaries 1

1. Introduction 3
1.1. Motivation 3
1.2. Research Problem 4
1.2.1. Evaluate Component-Based Systems 5
1.2.2. Generate Reusable Packages 6
1.2.3. Framework for Component Reuse 6
1.3. Contributions 6
1.4. Dissertation Outline 7

2. Background 9
2.1. Software Reuse 9
2.1.1. Benefits of Software Reuse 11
2.1.2. Software Reuse Techniques 13
2.1.3. Component Repositories 16
2.2. Component-Based Development 17
2.3. Component-Based Software Engineering 19
2.3.1. Software Architectures 22
2.3.2. Architecture Model 24
2.4. Automotive Software Architecture 25
2.5. Network Analysis 28
2.6. Simulated Annealing 28
2.7. Expert Systems 30
2.8. SAE AADL 31

3. Related Work 33
3.1. Component Selection Problem 33

3.2. Software Architecture Analysis . . . 34
3.3. Evaluation using Metrics . . . 35
3.4. Evaluation using Human Expertise . . . 36
3.5. Summary: Measurement Tasks . . . 36

II. Reusability Evaluation 39

4. System Reuse Modelling 41
4.1. Definition of the Solution Space . . . 41
4.1.1. Data Collection . . . 42
4.1.2. Generating the Component-Graph . . . 44
4.2. Summary: An Experienced-Based Component-Graph . . . 46

5. Network Analysis Based Reuse Measurement 47
5.1. Centrality Measures . . . 47
5.2. Cohesive Subgroups . . . 49
5.3. Application of Network Analysis . . . 51
5.4. Reuse Graph Metrics . . . 52
5.4.1. Preliminaries . . . 52
5.4.2. Reuse Popularity . . . 53
5.4.3. Component Rank . . . 54
5.4.4. Component Package Complexity . . . 54
5.5. Summary: Network Analysis Reuse-Metrics . . . 54

6. Reuse-Oriented Component Package Generation 57
6.1. Simulated Annealing for Component Clustering . . . 58
6.1.1. Initializing the Solution and Parameters . . . 58
6.2. A Recombinative Population-Oriented SA Algorithm . . . 64
6.2.1. The Algorithm (RPOSA) . . . 66
6.2.2. Candidate Recombination . . . 68
6.2.3. Neighbor Recombination . . . 70
6.3. Application of RPOSA . . . 72
6.4. Summary: Reuse-Oriented Component Packaging . . . 77

III. Evaluation Process and Framework 79

7. Model-Based Architecture Evaluation Framework 81
7.1. Motivation . . . 81
7.2. Functional Partitioning to Reuse Partitioning . . . 82
7.3. Reuse-Oriented Development Process . . . 83
7.3.1. Model-Based Reuse Framework . . . 85
7.3.2. High Level System Architecture Specification . . . 85

7.3.3. Reusable Component Package Import 87
7.3.4. Architecture Refinement 87
7.3.5. Architecture Implementation and Instrumentation 87
7.3.6. Generation of Software Artefacts 88
7.4. Multicore Support 88
7.5. Summary: Reuse-Oriented Development Process 90

IV. Case Study: Engine Management System 93

8. Evaluation of the Engine Management System 95
8.1. EMS Reuse: Status Quo 95
8.2. Data Collection And Extraction 96
8.3. Application of Network Analysis Measures 97
8.4. Application of the RPOSA Algorithm 100
8.5. Summary: Case Study 104

V. Conclusions and Future Work 107

9. Conclusions and Future Work 109
9.1. Conclusions 109
9.2. Future Work 111

VI. Appendix 113

A. Example Project Repository 115

List of Tables

2.1. Comparison of strategies for optimization problems. 29

3.1. Example metrics for the assessment of software reuse ([Dumke2003]). 38

4.1. Project repository example. For the full list of components of the example project repository used in Section 5.3, see Appendix A. . . 42

6.1. Tested values for the control parameters and the final decisions for the example repository. 74

6.2. Final clustering to reusable packages of the components from the example repository (sorted in ascending order). 76

8.1. Tested values for the control parameters and the final decisions for the full network. 103

A.1. Example of a companies possible project repository were individual components are associated to a particular project. 115

List of Figures

1.1. Evolution of engine management systems [Claraz et al.2004]. Each dot represents a finished customer project. While time-to-market has roughly been cut to a half and the ECU price to one third, the need for ROM is increasing drastically. This is mostly due the change from 16-Bit to 32-Bit systems (in gasoline (GS) as well as in diesel (DS) projects) and the increasing number of functionalities. . 4

2.1. Potential reusable artefacts 11
2.2. Reusable components (assets) characterized by description and body after [Dumke2003] and [Ezran et al.2002]. 16
2.3. Overview about the component-based development process. 18
2.4. Aspects of Component-based software engineering from [Dumke2003]. 19
2.5. The main goals for architectural decisions and their main dependencies. Squares are indicating a target conflict, while the circle-style connections indicate goals that combine well. 21
2.6. The Engine Management System (EMS2) partitioning according to blocks of functionality. 25
2.7. Figure (a) shows a high level model of the reuse structure of the EMS2 architecture for powertrain control units. In Figure (b) the configuration of an Aggregate, split into a fixed core, a configurable part and a project-specific part, is shown. The different parts are connected via an internal interface. 26

3.1. Potential measurement approaches within the context of a component architecture after [Dumke et al.2002]. 36

4.1. Set A of component-based system architectures including candidate architectures a composed of candidate components c out of the data set D. 42
4.2. Schematic view on the generation of the component-graph. The data for the network analysis is generated by the cumulated adjacency graphs (1) of the projects ($a = \{c_1, c_2, \ldots, c_n\}$, D is the collection of all possible modules). The final weighted component-graph (2) over all projects is analyzed with network analysis measures. 43

4.3. Exemplary graphical representation of a system, generated with the methodology shown in Figure 4.2, as a component-graph G. The components v_2 and v_3 were both used in the candidate architectures a_1, a_2, a_3 represented by an connecting edge with weight $w = 3$ and a label with the corresponding candidate architectures. Each vertex has a node label class C_i representing the cluster a particular vertex is assigned to. 44

5.1. Graphical representation of patterns in the reuse network. Graph {1} represents a vertex (v_1) with high degree (5) but low weighted degree. It corresponds to a component which was used often but in different contexts (systems) and therefore has a low reusability. Graph {2} and {3} show vertices with high reusability while Graph {3} represents a pattern which would lead to a high eigenvector centrality of vertex v_1. Graph {4} shows the concept of gatekeepers (v_3, v_6). 49

5.2. The schematic representation of a network with community structure (as presented in [Girvan and Newman2002]). It shows three communities (with densely connected vertices) with much lower density connections (gray lines) between them. 50

5.3. Figure (a) shows the graph of the example repository. The layouting of the graph shows found clusters (e.g. IDs 0,1,2,3). Figure (b) shows the corresponding dendrogram. 51

5.4. The key component analysis. The IDs of the most important components in the network analyzed by eigenvector centrality and betweenness centrality. 52

6.1. Example of possible neighbors and the connected path of a candidate component. In this example, the neighborhood $\Gamma(v_1)$ of the component v_1 is formed by the adjacent components $\{v_2, v_3, v_5, v_7\}$ while the thick line marks an example of a connected path > 1 through the network starting with the vertex v_1 to the vertex v_{10}. 60

6.2. The recombinative simulated annealing algorithm starts with initializing the solution set as a graph $G = (V, E)$, setting the initial parameters for the specific run (depending on the structure of the problem) and choosing randomly one candidate solution from the generated initial population P (which is evaluated according the objective function). On this candidate solution a *candidate recombination* and a *neighbor recombination* is performed a number l times and the next candiate solution is choosen from the current population. After the operations are performed on the current population P_i it is updated with the new candidate solutions and the temperature is decreased according the cooling schedule. The final population is evaluated according the objective function to measure the quality of the new solutions. 65

6.3. Every candidate solution (each different color represents a different solution) in the population explores the solution space and contributes to the fitness of the packaging. Due to the two recombination methods the generation of new solutions as well as the discarding of solutions is possible. In this example the algorithm starts with a population containing only four candidate solutions. During the run new solutions are generated. 67

6.4. Performing a candidate recombination operation. From the solution in image {1} the candidate components v_6 and v_4 are selected for recombination. Before the recombination the packages in image {1} have a fitness value of 0 as they share no common candidate architecture. In image {2} the new formed solutions are pictured. While the packages with one component each have a fitness of 0 the new formed package with the candidate components v_8, v_9, v_{10} have a fitness of 6 (ignoring the component rank in this example). 70

6.5. Performing a neighbor recombination operation. From the solution p_1 in image {1} the candidate component v_6 selects the components v_8 and v_9 from the neighbor solution p'_1 for a recombination operation. After the operation in image {2}, the new neighbor solution $p{*}'$ would not be connected and the new solutions $p{*}'_i$ and $p{*}'_j$ are created. The fitness of solution p_2 has an unchanged value of 3 (ignoring component ranks in this example) while the fitness values of the new solutions also remain 0 after the recombination 73

6.6. RPOSA algorithm applied to the example repository. The dotted line represents the best solution found in over 150 runs while the black line pictures the fitness of the population over the iterations. The grey line shows the development of the number of clusters in the population. 75

7.1. The current reuse process based on functional partitioning. The system architect defines the functionality and the authority (the function representative) for each function implements and supports it. Parts of the function are composed together to an aggregate which is used in system projects. 82

7.2. The proposed reuse process based on reuse partitioning. Like in the current process the system architect defines the functionality and the authority (the function representative) for each function implements and supports it. Parts of each function are composed together in reusable packages which are used in system projects. 83

7.3. The proposed process the framework is wrappend in. After step five (a component defines not only code but also documentation and specification) the component repository is updated with the new components. 84

7.4. Proposed architecture framework with an experience-based component repository (5), the software architecture (1) specified by an architecture description language and the architecture implementation (3) which is instrumented with various non-functional requirements (2) for further analysis (e.g. real-time simulation) and contains the functional parts for generation of code and documentation (4). . . . 86

8.1. This figure shows the reuse of modules over different projects. . . . 96

8.2. This figure shows the reuse of modules over all projects. One bin represents 20 projects. 97

8.3. Generation of the maximum weighted spanning tree. One component of the graph (1) is chosen as starting point. The components with the maximum weight are chosen (2). The final graph (3) contains all components with a reduced set of edges. 98

8.4. This figure shows the key component analysis. The IDs of the most important components in the network analyzed by eigenvector centrality and betweenness centrality. Scaled by eigenvector centrality. 99

8.5. This figure shows the degree distribution over the graph. Most of the components have a degree between one and four. 99

8.6. The resulting graph after the edge betweenness community clustering 101

8.7. Figure (a) shows the distribution of different aggregates in a selected cluster. Two specific clusters have a very high share in this cluster. In total, 20 aggregates have been grouped to one cluster. Figure (b) shows a cluster with four different aggregates grouped together. Components, which act as gatekeepers between the different aggregates, could be identified. 102

8.8. RPOSA algorithm applied to the full network. The dotted line represents the best solution found (19208) in various test runs while the black line pictures the fitness of the population over the iterations (with a final value of 18661.82). The grey line shows the development of the number of solutions in the population (with an initial value of 104 and an final value of 972). 105
8.9. This figure shows the first 100000 iterations of the RPOSA algorithm applied to the full network. This image pictures the exploration phase of the algorithm at the first 20000 iterations. 106

List of Algorithms

1. Simulated Annealing . . . 30

2. RPOSA($G, F, \alpha, T_0, T_{min}, X_0, Y_0, L_{max}$) . . . 69
3. Candidate Recombination . . . 71
4. Neighbor Recombination . . . 72

Abbrevations

AADL SAE Architecture Analysis and Design Language

ADL Architecture Description Language

CAME Computer Assisted Software Measurement and Evaluation tools

CBD Component-Based Development

CBSE Component-Based Software Engineering

COTS Commercial of the Shelf

EMS Enginge Management System

OO Object Oriented

OOD Object-Oriented Development

RPOSA Recombinative Population-Oriented Simulated Annealing

SA Simulated Annealing

Part I.

Preliminaries

1. Introduction

Reusability is an important aspect in the development of component-based software for embedded automotive systems. This thesis adresses reuse in this systems with an emphasis on component reuse and in particular the generation of reusable component packages. Although the thesis concentrates on algorithms for the generation of this packages, it also covers analysis methods for the evaluation of individual components of component-based embedded software systems. This introductory chapter presents the motivation behind this work, the research problem, contributions and an overview of the thesis.

1.1. Motivation

The increasing complexity of new-generation embedded systems raises major concerns in various critical application domains, like for example, the automotive industry. Modern cars contain a huge amount of features regarding passenger safety, environment protection or comfort. Most of them would not be possible without the support of electronic devices, which are managed by controllers performing complex control strategies. In recent years, the complexity of these algorithms grew enormously (see Figure 1.1). Within twelve years, an increase in terms of memory consumption and calculation power by the factor sixteen was observed. In the same time, the software engineering methods changed from assembler programming to the introduction of C as programming language and finally to the use of model based development or even floating point based algorithms.

In the development of complex control strategies the increasing effort, that is required to manage state of the art combustion engines, as well as the cost pressure and the short development cycles made it necessary to introduce a concept for reusing solutions for the automotive powertrain domain.

During the last decade, engineering approaches have emerged that aimed at mastering this complexity during the development process. One of these approaches is the use of component-based software development which has proven to support the development of complex software solutions [Rombach2003, Weisbecker2002]. Component-based engineering and, more recently, model-driven engineering address the problem of complexity by promoting reuse and partial or total automation of certain phases of the development process. For the successful implementation

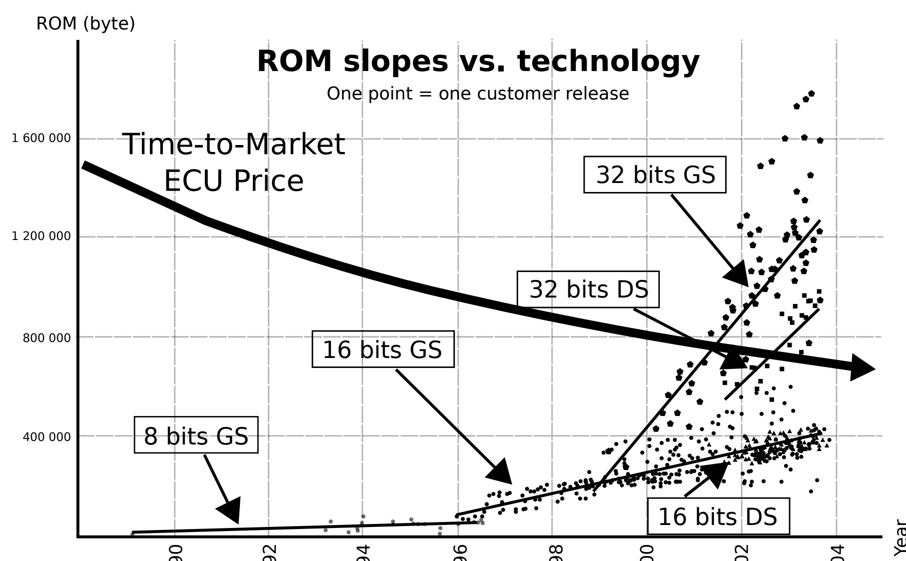


Figure 1.1.: Evolution of engine management systems [Claraz et al.2004]. Each dot represents a finished customer project. While time-to-market has roughly been cut to a half and the ECU price to one third, the need for ROM is increasing drastically. This is mostly due the change from 16-Bit to 32-Bit systems (in gasoline (GS) as well as in diesel (DS) projects) and the increasing number of functionalities.

of component-based software engineering in the development process the reuse of existing components is an essential requirement [Morisio et al.2002].

Frequently, reuse is seen as a powerful approach to reduce the number of software errors in the final product. Components, while having been used in different contexts, typically contain less undiscovered inactive bugs and because the testing effort is only spent once, reuse also reduces the cost pressure on the testing phase. So, if an application is built from such reused components, the likelihood of failure is lower than in the case of building software from new components [Gaffney et al.1989, Lim1994].

These engineering approaches must be supported by languages and tools that provide means to ensure that the implemented system complies with its specifications. In particular, it is necessary to integrate analyses of quality attributes (such as dependability and performance or reusability) in the development process [Bachmann et al.2000].

1.2. Research Problem

The previous section stated that the embedded domain, and in particular the automotive industry, is facing the challange of continously growing complexity in their

systems. To meet the challenge, in the year 2004 a component-based architecture was developed for automotive engine management systems at Siemens VDO. They were grouping the complete control unit into functionalities (e.g. injection control, idle speed control), which then were structured into reusable parts, the so called *modules*.

Over the years, the increasing complexity of algorithms resulted in an increasing number of modules, making the management of the reuse more and more difficult and inefficient. This led to an clustering of modules, to the so called *aggregates*. The introduction of this additional level of reuse in the architecture was not only aiming at simplifying the reuse (e.g. by reducing the integration effort of existing solutions into new project environments), but also at structuring the development to achieve a reduced number of variants.

As it is not clear, if the decisions that have been made when creating the partitioning, will last for the lifetime of the architecture, a measure for the quality of the defined architecture is required. Furthermore a solution to evaluate the reuse desicions, made while introducing the concept of aggregates, is required in order to determine to which extent the expected reuse benefits have materialized after the adoption of the defined architecture. The following three main tasks where identified:

- Evaluate component-based systems and in particular identify components which qualify for reuse.
- Group components of existing solutions to reusable packages to reduce the integration effort for new solutions.
- Provide a framework, supported by a defined process, for the handling of components in component-based architectures.

This thesis solves problems related to all three of the tasks. In the following subsections the problem, which have been adressed by this thesis, are shortly presented.

1.2.1. Evaluate Component-Based Systems

Due to the increasing number of modules that result from new functionalities, the increasing complexity of algorithms and the support for a product line approach, the management of components has become difficult and inefficient. To choose which individual component qualifies best to be reused for a specific task is almost impossible due to the vast number of versions and revisions of a component. Therefore a solution for evaluating and measuring the importance and impact of individual components is needed.

1.2.2. Generate Reusable Packages

The way of grouping modules (the components of the system) into aggregates was driven by several structuring concepts. First of all, functional coherence was considered. The functionality of a combustion engine management system can be divided hierarchically into units, thus creating a partitioning of the functionality according to the engine management physics and the supporting functions (e.g. comfort or legal requirements). Additionally, the complexity of interfaces and the encapsulation of components was a driving factor for the partitioning definition.

Furthermore, it was ensured that the organizational responsibility matches the borders of the defined components to simplify cooperation and minimize communication problems between teams, especially in the communication with the customer. To measure the maximum impact of reusability and maximize reuse efficiency, a new solution for the grouping of components to reusable packages has to be evaluated.

1.2.3. Framework for Component Reuse

To build new solutions from existing components, the systems have to have a common architecture framework. Also the management of components, e.g. search for specific components or component groups and retrieve them from the repository which is integrated into the framework, could lead to a higher reuse efficiency.

1.3. Contributions

This thesis deals with issues related to reusability in component-based embedded systems. The main contributions are summarized in the following:

Component reuse network. A component-based system architecture is transformed into a graph. Graph theoretical measures are applied in the context of reusability in component-based systems. Graph metrics are defined to analyse the network for central components whereas clustering algorithms are then applied to generate meaningfull groups of components.

Automatic reuse optimized clustering of components. A combinatorial optimization algorithm has been developed which, given a component reuse network, clusters components to reusable packages. The algorithm adapts the simulated annealing methodology to optimize clusters of components for reuse.

Evaluation Framework. A model-based framework serving as expert system has been proposed to support system architects and integration engineers in the development process.

Although these items are contributions by themselves and presented in Part II (reusability analysis) and Part III (theoretical framework) respectively, they can also be considered as part of one single proposed methodology for the improvement of reuse in component-based embedded systems. The components are first evaluated for centrality and importance to help system experts with the selection of individual components. In a second step the components are grouped to reusable packages to separate project specific implementations from reusable parts and help integration engineers with the construction of new solutions. These steps are combined in a model-based framework.

1.4. Dissertation Outline

The thesis is divided into five main parts. Part I introduces the motivation behind the use of component-based systems in the embedded domain. It furthermore presents the background needed to understand the thesis and a high-level overview of the used concepts. Part II introduces methodologies for evaluating component-based systems and cluster components to reusable packages. Then Part III demonstrates the proposed methods on a industrial case study. Next Part IV presents a model-based framework wrapped in a defined process. Finally Part V concludes the thesis and points out a few areas for future work.

The five parts are divided into chapters as follows:

Part I: Preliminaries

- Chapter 1 shortly motivates the importance of reuse in the area of component-based automotive systems. Furthermore it summarizes the problems discussed, states the contribution of this thesis and gives an overview of the structure of this thesis.
- Chapter 2 provides background of the research area and introduces prerequisite concepts and methodologies.
- Chapter 3 adresses related work.

Part II: Reusability Evaluation

- Chapter 4 introduces the concept for collecting the data from project repositories and the representation of a component-based system as a graph.

- Chapter 5 introduces network analysis methods, discusses metrics to evaluate the graph and the application to an example repository.
- Chapter 6 presents an optimization algorithm based on simulated annealing for the generation of reusable component packages.

Part III: Evaluation Framework

- Chapter 7 presents the theoretical model-based framework, based on an architecture description language and application scenarios.

Part IV: Case Study: Engine Management System

- Chapter 8 applies the discussed network analysis methods and the RPOSA algorithm to the engine management system.

Part V: Conclusions and Future Work

- Chapter 9 concludes the thesis and discusses possible issues for future work.

In the Appendix at the end of the thesis, supplementary material (e.g. additional results of experiments) and a summary of abbrevations and notations have been included.

2. Background

This chapter explores the background to this research and is devoted to the introduction of the basic concepts and terminologies. In Section 2.1 software reuse is described in general and the boundaries of this research are definded. This work aims to develop a methodology that supports the reuse management of software components in embedded automotive systems.

As this work is based on component-based software systems the notion of component-based development is discussed in Section 2.2. Reuse packages discussed in this work are composed of many small reusable software components which are typically stored in a component library or database. Such components can have simple or complex functionality, may be static or adaptable and can be built in-house or bought off-the-shelf. These component characteristics are described in Section 2.2 and important considerations as well as an introduction to the target system are discussed in Section 2.4.

Component reuse in embedded automotive systems discussed in Section 2.4 differs significantly from static black-box component reuse. Several barriers to component-based development, such as component retrieval techniques and the lack of support for understanding and integrating components are identified in this section. To measure the reusability of the system or system parts and evaluate individual components for their impact on the overall reuse an approach based on network analysis was used which is introduced in Section 2.5.

To provide means for the automatic clustering of components to reusable packages which intend to optimize the packaging regarding reuse an algorithm, based on an optimization heuristic, is presented. The background to this optimization approach which uses simulated annealing as technique is presented in Section 2.6.

2.1. Software Reuse

The 1968 NATO Software Engineering Conference is generally considered the birthplace of the software engineering field of reuse. The conference focused on the software crisis — the problem of building large, reliable software systems in a controled, cost-effective way. Software reuse by McIlroy [McIlroy et al.1969] — Mass Produced Software Components — was an invited paper at the conference. Therefore the idea of systematic reuse (the planned development and widespread use of

software components) was first proposed in 1968 by Doug McIlroy. Since then, many attempts at improving the software process by using reusable software components have been proposed and tried, with varying degrees of success. One reason for this may be the different viewpoints about what software reuse is.

Basili and Rombach [Basili and Rombach1988] define software reuse as the use of everything associated with a software project, including knowledge. Braun [Braun] defines reuse as the use of existing software components in a new context, either elsewhere in the same system or in another system. An important aspect is whether software to be reused may be modified. Cooper defines software reuse as the capability of a previosly developed software component to be used again or used repeatedly, in part or in its entirety, with or without modification [Cooper1994]. A more general view of software reuse was defined by Krueger as follows [Krueger1992]:

> Software reuse is the process of creating software systems from existing software rather than building them from scratch.

Ezran et. al. [Ezran et al.2002] complemented this definition with the addition of goals as follows:

> Software reuse is the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance.

In [Dumke2003] Dumke established the following basic characteristics for software reuse from the definition provided by Ezran et. al.:

- An exclusive reference to the software development while the maintenance is, in a certain manner, included in the further development.
- The characterization of reuse components as a “stock of building blocks” includes, refering to requirements and architecture components, both kinds of components.
- Reuse is characterized as a systematic approach.
- The goals of reuse are higher productivity, quality and business performance.

In software development, the idea of reusability is generally associated with code reuse. However, reusability is a much wider notion that goes beyond APIs and class libraries to include many types of artefacts of the software development project, e.g., requirements specification, design patterns, architectural description, design rationale, document templates, test scripts and so forth [Wasserman1996]. A overview about those potential reusable artefacts is illustrated in Figure 2.1.

McCarey [McCarey et al.2008] characterised the reuse lifecycle for reusable artefacts generally into five steps:

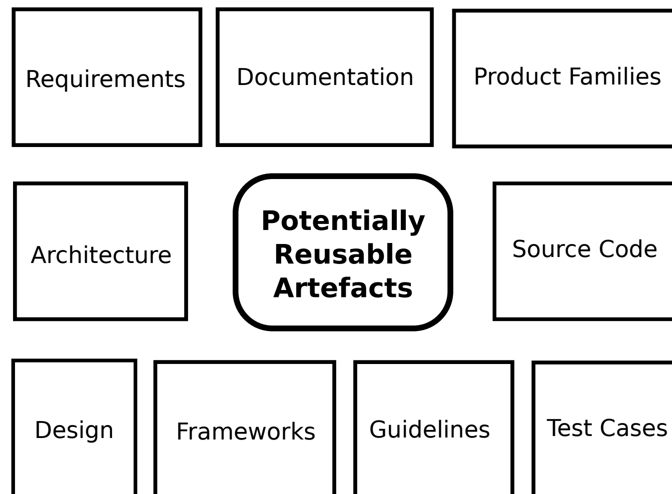


Figure 2.1.: Potential reusable artefacts

1. Specify requirements in terms that can be matched with the description of the artefacts.
2. Search for relevant artefacts.
3. Evaluate the retrieved artefacts reuse potential.
4. Select the most suitable artefact and adapt as appropriate.
5. Integrate the artefact in the current project.

The goal of this research is to develop methodologies to automate the points two and three of this reuse lifecycle. These artefacts can be loosely classified into two categories, namely *knowledge reuse* and *code reuse* [McCarey et al.2008]. Knowledge reuse describes the use of high-level abstract software artefacts which can be modeled at different abstraction levels. These are, for example, design patterns or software architectures. Code reuse refers to the instantiation of this knowledge via source code. Unlike the classification of [McCarey et al.2008], component-based reuse is classified as a form of knowledge reuse in this work. This is motivated by the architecture of the case study which is discussed in Chapter 8. This component-based system architecture defines a component as a composition of code, documentation and test cases. A more detailed definition of this particular system architecture is provided in Section 2.4. In the following sections the benefits of software reuse, varying techniques and implications of reuse are discussed in more detail.

2.1.1. Benefits of Software Reuse

The benefits of software reuse have been treaten many times in literature e.g. the Encyclopedia of Software Engineering [Braun], the NATO Standards for Software

Reuse [Braun1992] and in various overviews of software reuse in journals and books [Sametinger1997, Mohagheghi and Conradi2007]. In summary the main benefits of reuse result in quality improvements and effort reduction as stated by Sametinger [Sametinger1997] as follows :

Quality improvements:

Quality: Error fixes accumulate from reuse to reuse through a higher testing coverage. This yields higher quality for a reused component than would be the case for a component that is developed and used only once.

Productivity: A productivity gain is achieved due to less code that has to be developed. This results in less testing efforts and also saves analysis and design labor, yielding overall savings in cost.

Reliability: Using well-tested components increases the reliability of a software system. Furthermore, the use of a component in several systems increases the chance of errors to be detected and strengthens confidence in that component.

Interoperability: Various systems can work better together if their interfaces are implemented consistently. This is the case when they use the same components for these interfaces.

Effort reduction:

Redundant work and development time: Developing every system from scratch means redundant development of many parts. This can be avoided when these parts are available as reusable components.

Time-to-Market: The success or failure of a software product is very often determined by its time to market. Using reusable components will result in a reduction of that time [Lim1998].

Documentaton: Reusing software components reduces the amount of documentation to be written.

Maintenance costs: Fewer defects can be expected to occur when proven components have been used, and less of the software system must be maintained.

Training costs: Over time, software engineers become familiar with the reusable components available for their development efforts. So they have a good working knowledge of many components of theses systems when they are starting to design and develop new systems.

Team size: If many components ca be reused, then software systems can be developed with smaller teams.

Among all the powerful software technologies abailable today, software reuse is the best way to acelerate the production of high quality software [McClure and McClure2001].

Of course reuse has not only benefits. As Sametinger stated [Sametinger1997] there are several obstacles to software reuse. There are many factors that directly or indirectly influence the success or failure of reuse. These factors can be of conceptual, technical, managerial, organizational, psychological, economic or legal nature. In this work we concentrate mainly on the conceptual and technical obstacles of software reuse. One open problem is the difficulty of finding reusable software components. Software can not be reused unless it can be found. Reuse is unlikely to happen when a repository does not have sufficient information about components or when the components are poorly classified. In complex component-based systems the reusable parts have to be well-organized to maximize the reuse.

2.1.2. Software Reuse Techniques

Various techniques or approaches can be used in order to achieve software reuse. One technique is *Compositional reuse* which supports bottom-up development of systems from a repository of available lower-level components. The classification and retrieval of components is very important in this context and is subject to the approach presented in this thesis. Another approach is *generative reuse* which is often domain-specific, adopting standard system structures, e.g. reference architectures or generic architectures, and standard interfaces for components. An extensive overview about the techniques are provided by Dumke in [Dumke2003] and Sametinger in [Sametinger1997]. In the following various reuse techniques are summarized:

Ad hoc Reuse: Ad hoc reuse is the reuse of software for a defined goal, a defined point in time or a special requirement and is, in general, realized only once [Dumke2003] .

Abstraction: Wegner [Wegner] stated that abstraction and reusability are two sides of the same coin and is therefore essential in any software reuse technique. In software engineering it is a major challenge to raise the abstraction level and to find concise abstractions for components is a difficult task.

Black-Box Reuse: Using black-box reuse means to use a component without seeing, knowing or modifying any of its internals. The component provides an interface that contains all the information necessary for its utilization. Object-oriented techniques allow modifications of black boxes by making modifications and extensions to a component without knowing its internals [Sametinger1997].

Compositional Reuse: The idea of compositional reuse is based on reusable components that remain unmodified in their reuse. Complex or higher-level components are build by combining lower-level or simpler components. New components are only build if a needed component is not available and cannot

be created by modification of exiting components. The components intended for reuse are collected in repositories (e.g. function libraries).

Defensive Reuse: In the case of defensive reuse software requirements are in general regarded as predefined and constant. Based on this a possible reuse is designed and realized [Dumke2003].

Explicit Reuse: The explicit reuse includes the posibility of an exact identification of the reuse component in the reuse process [Dumke2003].

External Reuse: The external software reuse uses components from parts outside of the realised software development. This form of reuse implies, after Dumke [Dumke2003], the so called "Boundary Problem" which expresses the complexity of the determination of external and internal borders.

Generative Reuse: Generative Reuse is based on the reuse of a generation process rather than the reuse of components. Large structures are used as invarians, i.e., reused without change.

Glass-Box Reuse: Goldberg and Rubin [Goldberg and Rubin1995] used the term glass-box reuse if components are used as-is like black boxes, but their internals can be seen from the outside. It gives the software engineer information about how the component works without the ability to change it. But this may lead to dependencies on certain implementation details which become fatal when the internals of the component are changed.

Grey-Box Reuse: In case of a grey-box reuse the components are not "Black Box" to the extend that they can be parameterized (e.g. to interfaces).

Horizontal Reuse: The horizontal reuse refers to the application area and therefore is problem-/domainspecific oriented (to a certain extend regarded as top-down reuse).

Implicit Reuse: In case of implicit reuse the reusable components are included (or embedded via special techniques) in the product parts [Dumke2003].

Inside Reuse: In case of inside reuse the reused components are part of the final product [Dumke2003].

Internal Reuse: Internal reuse is defined by solely use of components from the software development area itself [Dumke2003].

Offensive Reuse: In case of offensive reuse the software requirements are, as far as possible, adapted to the already existing components [Dumke2003].

Outside Reuse: In case of outside reuse the reused components, which are used during the software development, are not part of the final product [Dumke2003].

Systematic Reuse: The systematic software reuse is a reuse form that consists of a (pre)defined process, specific techniques and a personnel structure which is comprehnsible and therefore rateable [Dumke2003].

White-Box Reuse: White-box reuse means reuse of components by adaptation. The internals of the component are changed for the purpose of reuse. Unlike black-box reuse, a new component derived by modifications to an existing component must be regarded as a new component and thoroughly tested. It requires separate maintenance and in the case of the existance of many copies of a component with slight modifications, it becomes burdensome to fix errors that affect all of them [Dumke2003].

These techniques inted to maximize the reuse maturity in the development process of a company. Reuse maturity can be seen as the range of expected results in reuse efficiency, reuse proficiency and reuse effectiveness that can be achieved in an organization by following a reuse process [Davis1992].

Reuse Efficiency: Reuse efficiency measures how much of the intended reuse opportunities have actually been exploited by an organization.

Reuse Proficiency: Reuse proficiency is the ratio of actual reuse to potential reuse.

Reuse Effectiveness: Reuse effectivenewss is the ratio of reuse benefits to reuse costs.

Within the scope of software reuse the term **asset** is used for a software component. Assets describe what can be reused during the software development or maintenance. Software assets are defined by Ezran et. al. [Ezran et al.2002] as follows:

> Software Assets are composed of a collection of related software work products that may be reused from one application to another.

For the description of reusable components (assets) different approaches can be found in the literature. In [Dumke2003] and [Ezran et al.2002] an asset is characterized by a *description* and a *body* as shown in figure 2.2.

The description of an asset consists of meta information about the asset and the body includes a number of actual products. The individual products often represents the very same piece of software on different levels of abstraction (concept, analysis, design, implementation, test) [Dumke2003]. A more extensive description on assets and a proposal for a defined structure is provided in [Dumke and Schmietendorf2000]. In this work however the more general term *component* is used.

Basili et. al. stated in [Basili et al.1992] that reuse is a simple concept, using the same thing more than once. Although a simple concept, reuse is a powerful software

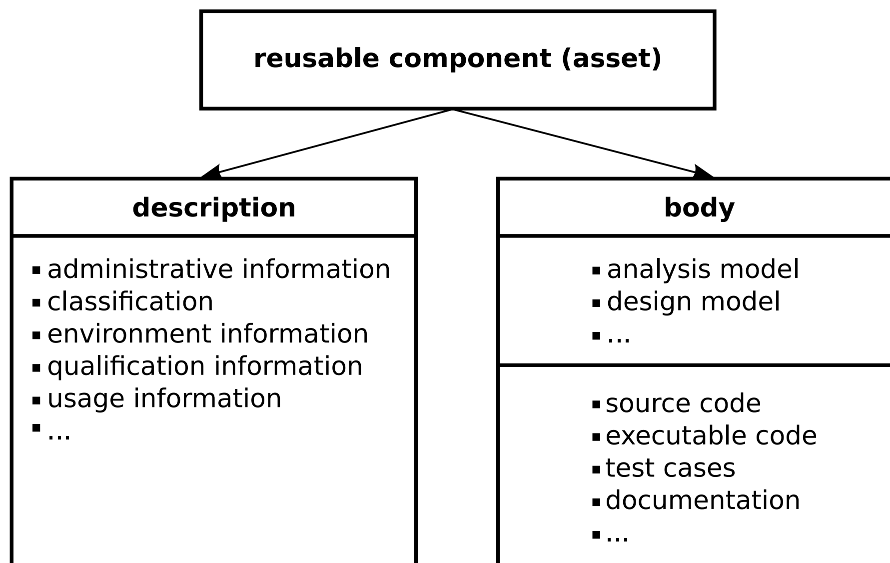


Figure 2.2.: Reusable components (assets) characterized by description and body after [Dumke2003] and [Ezran et al.2002].

practice that can deliver significant improvements in software productivity and quality, as well as substantially lower software development and maintenance costs. However, as experiences from industry prove [Morisio et al.2002], it is nothing but simple in practice.

2.1.3. Component Repositories

The main requirement for successful reuse is the availability of a wide variety of high-quality components. But the existance of those components alone does not guarantee successfull reuse. Proper classification and retrieval mechanisms, sufficient and proper documentation of components, flexible means for combining components, and means of adapting components to specific needs [Sametinger1997] are critical factors for success. In an ideal scenario reused components are largely atomic and remain unmodified. However, often this ideal cannot be achieved and the components have to be modified and changed in order to fit the special purpose (as defined as white-box reuse). In this process different versions and revisions of components are constructed which lead to a higher complexity in repositories. During composition, compontents are combined by predefined principles which are crucial for systems being built from existing components. This principles could be for example partitions because of functional coherences or the demand for higher reuse in projects.

As libraries of reusable software components continue to grow, the issue of retrieving components from software component repositories has captured the attention of

the software reuse community [Burton et al.1987, Devanbu et al.1991, Esteva1995, Fischer1998, Ostertag et al.1992, Frakes and Gandel1990, Frakes and Nejmeh1986, Maarek et al., Prieto-Diaz and Freeman1987] and is still an open research topic.

According to Ostertag and Hendler [Ostertag et al.1992], the ability to reuse existing software requires four steps: definition, retrieval, adaptation, and incorporation. The definition step describes the component which needs to be constructed in terms of its functionality and relation to the rest of the environment. The retrieval step takes this description and retrieves from the software library a list of components with similar characteristics. One of these components is then selected. During the adaptation step the needed component is generated, usually by modifying the component that was selected from the library. The needed component is then incorporated in a new software package during development. Finally, new reusable software components are derived from the current software development project, and inserted into the software reuse library. The classification of a software component for easy later retrieval from a software library is an important part of the reuse procedure.

Despite the various obstacles, systematic reuse is generally recognized as a key technology for improving software productivity and quality [Mili et al.1995]. Software reuse is practiced to save time and money, and to improve quality as discussed by McClure in [McClure and McClure2001]. The popularity of object-oriented development (OOD) brought the notion of reuse to the forefront [Dumke et al.1996] but the software community was disappointed because less object reuse was achieved than expected. Reuse, it seems, is not an automatic byproduct of OOD. The next form of reuse centers on components and component-based development which is introduced in the following section.

2.2. Component-Based Development

Component-Based Development (CBD) is a still emerging software development paradigm that promises many benefits including reduced development and maintenance costs, and increased productivity. The main features of CBD originate from business requirements: Short time-to-market. Large savings in time can be achieved by constructing applications from already existing parts as well as the distribution of work among dedicated experts for development of components. Since components are developed independently of the products the experts in particular domains can develop them. The latest trends show that different component technologies are being developed for different domains. Similarly to the object-oriented (OO) paradigm that is exploited in different OO languages, a component-based paradigm based on certain common principles is slowly built and used in different component technologies as discussed in [Bouyssounouse and Sifakis2005, Crnkovic and Larsson2002].

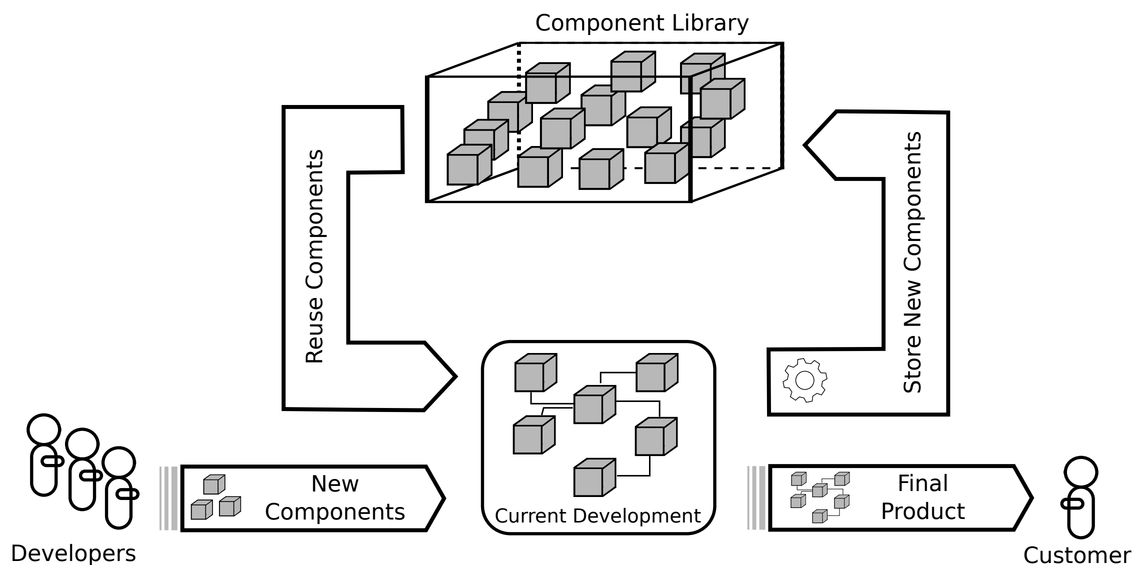


Figure 2.3.: Overview about the component-based development process.

In [Dumke and Winkler1997] Dumke et. al. describe Component-based software development as a new way for a more flexible software generation, composition and integration while the components have the following main characteristics [Barry1996]:

- Components general do something useful
- Components are a small related set of functions or services
- Real OO programs are component based
- Classes are not components
- Components are composable
- Frameworks often define component families

Component-Based Development involves the technical process of designing and implementing reusable components, and assembling applications from existing components as is illustrated in Figure 2.3. Bachmann et. al [Bachmann et al.2000] refer to the practice necessary to perform CBD in a repeatable way to build systems that have predictable properties.

From the engineering point of view the advantages of CBD are based on standardisation and reusability. Standardisation plays a crucial role as it enables independent development and seamless integration of components. By reusing the same entities the confidence of their behaviour and properties increases. Similar to other engineering domains, CBD targets complexity: By reusing existing solution not only on the component level but also on the system structure level CBD enables a better understanding of complexity; the implementation details of components are hid-

den and only component services that are exposed through component interfaces are visible. In this way the abstraction level is increased which is a key factor in managing complexity.

Component-based systems result from adopting a component-based design strategy, and software component technology includes the products and concepts that support this design strategy. By design strategy (something very close to architectural style) a high-level design pattern is meant, described by the types of components in a system and their patterns of interaction [Clements and Kazman2003]. Component-based systems are the result of structuring a system according to a particular design pattern. Dumke et. al [Dumke and Winkler1997] argue that the general idea of component-based development leads to a “component-based” software measurement.

2.3. Component-Based Software Engineering

In literature, the terms Component-Based Development and Component-Based Software Engineering (CBSE) are often used indistinguishably. In a technical report by Bachmann et. al. [Bachmann et al.2000] the concept of CBSE is stated as:

> Component-based software engineering is concerned with the rapid assembly of systems from components where components and frameworks have certified properties and these certified properties provide the basis for predicting the properties of systems built from components.

In [Dumke2003] Dumke illustrated the fundamental software principles of component-based technologies as shown in Figure 2.4.

But the rapid assembly of systems from components is not the only goal of component-based system engineering. Clements et. al. [Clements and Kazman2003] summarized the goals as follows:

Cost reduction: Costs are a main driving factor for all decisions in industrial software development (see Figure 1.1). CBSE reduces the development steps for creating and assembling software systems.

Ease of assembly: Due to the well defined interfaces of the components they facilitate a quick and easy tool to support a subsequent assembly process. This satisfies the demand for a reduced time-to-market.

Reusability: Designing software for reuse in different applications is supported due to the partitioning of the complete software system into smaller parts (the components).

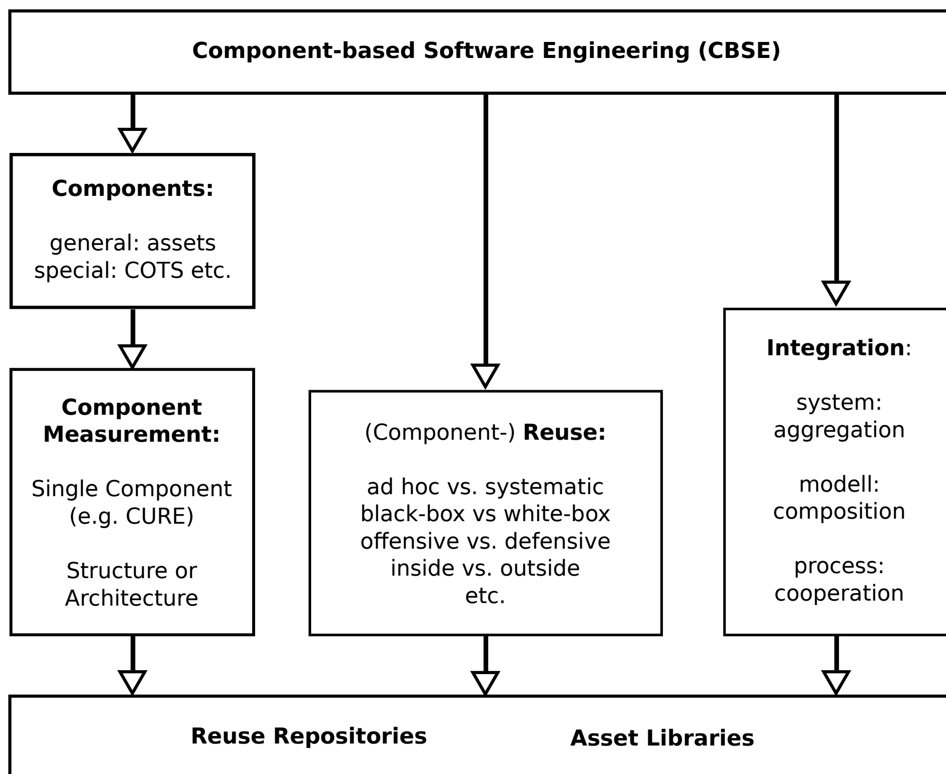


Figure 2.4.: Aspects of Component-based software engineering from [Dumke2003].

Customization and flexibility: In component-based systems, each component supports a clearly defined range of different configurations regarding its behavior and its interfaces. This allows the customization of the component and makes component design flexible.

Maintainability: The effect of changes can be restricted to a clearly defined set of components.

These properties of CBSE help to improve the quality of the developed software systems. In [Hobelsberger and Mottok2009] different software reliability models have been applied to measure the improvement of the software quality. Especially reuse and maintainability are playing a key role due to the higher test depth and the local effect of changes. In Figure 2.5 the main goals for architectural decisions and their dependencies to each other are shown.

Exploiting component-based development can provide very significant software productivity, quality, and cost improvements to an organization which are summarized by McClure [McClure and McClure2001] as follows:

- Deploy critical software applications more quickly
- Simplify large-scale software development

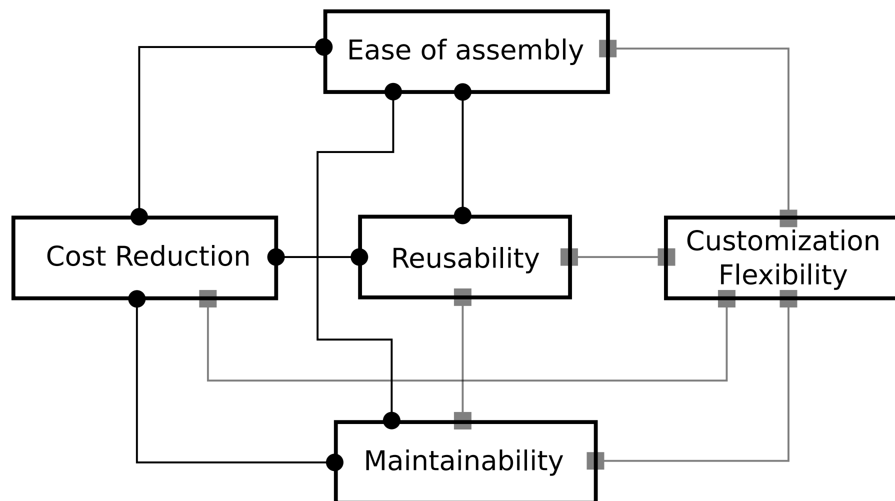


Figure 2.5.: The main goals for architectural decisions and their main dependencies. Squares are indicating a target conflict, while the circle-style connections indicate goals that combine well.

- Encapsulate business services into reusable application logic
- Shorten software development cycles
- Reduce the amount of new code to write
- Allow software applications to share functionality
- Make software applications more adaptable and easier to change
- Decrease software complexity
- Increase software reliability and overall quality
- Increase software productivity by reducing costs

The key consideration of component-based development is reusability and adaptability. The component models must have a mean by which components can be reused and adapted to the requirements. The work presented in this thesis focuses on the **reusability** of components in the component-based systems engineering process and particulary with the handling of the continously growing number of components (manifesting in different versions and revisions). In the domain of embedded automotive systems this has led to a number of implications which will be discussed in more detail in Section 2.4.

One of the major issues in software systems development today is quality as stated in [Dobrica and Niemel2002]. The idea of predicting the quality of a software product from a higher-level design description is not a new one. In 1942, Parnas [Parnas1972] described the use of modularization and information hiding as a means of high-level system decomposition to improve flexibility and comprehensibility. In

1974, Stevens et al. [Stevens et al.1974] introduced the notions of module coupling and cohesion to evaluate alternatives for program decomposition. During recent years, the notion of software architecture has emerged as the appropriate level for dealing with software quality.

This is because the scientific and industrial communities have recognized that software architectures sets the boundaries for the software qualities of the resulting system [Clements and Kazman2003]. In the following section an introduction to the notion of software architectures is provided.

2.3.1. Software Architectures

There are several definitions and understandings of software architectures and the basic concepts behind it in the field of computer science. At its essence, a software architecture is defined quite simply, by Taylor [Taylor et al.2009] as follows :

> **Definition:** A software systems architecture is the set of **principal design decisions** made about the system.

He emphasizes that the notion of *design decision* is central to software architecture and to all of the concepts based on it. The IEEE Standard 1471-2000 [Hilliard2000] defines a software architecture as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution while Paul Clements et. al. [Clements and Kazman2003] defines the software architecture of a system as the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Another useful definition of architecture, which adresses system evolution and sumarizes the definitions stated above is that, provided by the ANSI/IEEE Standard 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems:

> **Definition:** The architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

The individual parts of the software architecture as stated above are described in more detail in the following subsections.

2.3.1.1. Components

As stated, a software architecture describes the components of a software and the relationships between these components. Now it is considered what can be thought of as a component, and what as a relationship.

In [Taylor et al.2009] a software component is defined as follows:

> **Definition:** A *software component* is an architectural entity that (1) encapsulates a subset of the systems functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.

Another, widely cited, definition of software component is provided by Clemens Szyperski [Szyperski1998]:

> **Definition:** A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This definition does not tell what a component is, but rather how it is to be structured and used, both by software developers (by composing the component into a system and deploying it) and by other components (by interacting with it through explicit interfaces). This definition is very similar to several other definitions (for example see Heineman and Councills Book [Heineman and Councill2001] or Biggerstaff et. al. [Biggerstaff and Perlis1989]) as it does not separate data from computation in software systems architectures similar to the object-oriented paradigma in which all system elements are treated as objects, regardless of their purpose.

A comprehensive overview about component architecture, requirements and a classification of different kinds of components is presented by Schmietendorf et. al in [Schmietendorf et al.2002].

A component can be as simple as a single operation or as complex as an entire system, depending on the architecture, the perspective taken by the designers, and the needs of the given system. The key aspect of any component is that it can be "seen" by its users, wheter human or software, from the outside only, and only via the interface it (or, rather, its developer) has chosen to make public. Otherwise it apperars as a "black box". Software components are thus embodiments of the software engineering principles of *encapsulation, abstraction* and *modularity.* In turn, this has a number of positive implications on a components *composability, reusability,* and *evolvability.*

2.3.1.2. Connectors

Components are in charge of processing or data, or both simultaneously. Another fundamental aspect of software systems is interaction among the systems building blocks. Most modern systems are build from large numbers of complex components dynamically updated over long time periods. In such systems, ensuring

appropriate interactions among the components may become even more important and challenging to developers than the functionality of the individual components [Taylor et al.2009]. In other words, the interactions in a system become a principal (architectural) concern. Software connectors are the architectural abstraction tasked with managing component interactions.

> **Definition:** A software connector is an architectural element tasked with effecting and regulating interactions among components [Taylor et al.2009].

2.3.1.3. Configuration

Components and connectors are composed in a specific way in a given systems architecture to accomplish that systems objective. That composition represents the systems configuration, also referred to as **topology** which will become very important to the clustering approach presented in section 6.2. Configuration is definded as follows [Taylor et al.2009]:

> **Definition:** An architectural configuration is a set of specific associations between the components and connectors of a software systems architecture.

2.3.2. Architecture Model

A software systems architecture is captured in an architectural model using a particular modeling notation. Taylor [Taylor et al.2009] defined an architectural model as follows:

> **Definition:** An architectural model is an artifact that camptures some or all of the design decisions that comprise a systems architecture. Architectural modeling is the specification and documentation of those design decisions.

One system may have many distinct models associated with it as, for example, one model for every level of abstraction. Models may vary in the amount of detail they capture, the specific architectural perspective they capture (for instance, structural versus behavioral, static versus dynamic, entire sytem versus a particualar component or subsystem), the type of notation they use, and so forth. In Chapter 4 a specific model for the capturing of reusability of a component-based software system is presented.

> **Definition:** An architectural modeling notation is a language or means of capturing design decisions [Taylor et al.2009].

The notation for modeling software architectures are referred to as architecture description languages (ADLs) . ADLs can be textual or graphical, informal, formal,

domain-specific or general-purpose, proprietary or standardized. The ADL, used for architectural modeling in our approach, is the SAE AADL [Aerospace2004] which will be introduced in more detail in section 2.8. This choice for the SAE AADL as modeling language is based on comprehensive comparisons on the use of differend architecture description languages in the embedded domain presented in [Hobelsberger2007], [Hobelsberger et al.2007] and [Hobelsberger et al.2008].

Architectural models are used as the foundation for most of the other usual activities in architecture-based software development processes, such as analysis, system implementation (the automatic generation of code), deployment, and dynamic adaption.

2.4. Automotive Software Architecture

In this section an automotive software architecture, which will serve as a case study in Section 8.1 is presented: the **engine management system** of the Continental Automotive GmbH - Powertrain Division.

The engine management system (EMS2) architecture [Claraz et al.2004] was introduced in the year 2003 as a new, corporate wide standard in the powertrain domain at SiemensVDO and is still the standard EMS architecture at Continental Automotive Engine Systems. It is organized as a layer architecture which is used as a basis for systematical abstraction of hardware dependencies. It allows e.g. to exchange a used micro controller with a limited impact on the upper software layers. This hardware dependent software within the system is called *Infrastructure*. The other part of the software is hardware independent. Both layers feature a functional partitioning, which enables the reuse of individual parts following both functional and interface dependencies.

The elements of this partitioning cover the complete functionality of the system (see Figure 2.6). As these groups (hardware dependent and independent software) are too big to be reusable, they have been split into subparts, called *aggregates*.

In total, around 100 aggregates are defined. They are the basis for reuse and are themselves split into modules, which are the smallest units managed in the configuration (see Figure 2.7(a)). Each aggregate contains a complete functionality, allowing good encapsulation and weak binding between the aggregates.

Every aggregate is split into a fixed core, a configurable part, and a project-specific part (see Figure 2.7(b)). The two first parts are intended for broad reuse. The fixed core is strictly the same for all projects reusing the aggregate and is completely independent of the system configuration. The configurable part can be adapted to the requirements of the reusing project, by choosing one of the predefined configuration options. Finally the specific part has to fulfill predefined interface constraints but is considered as project integration code.

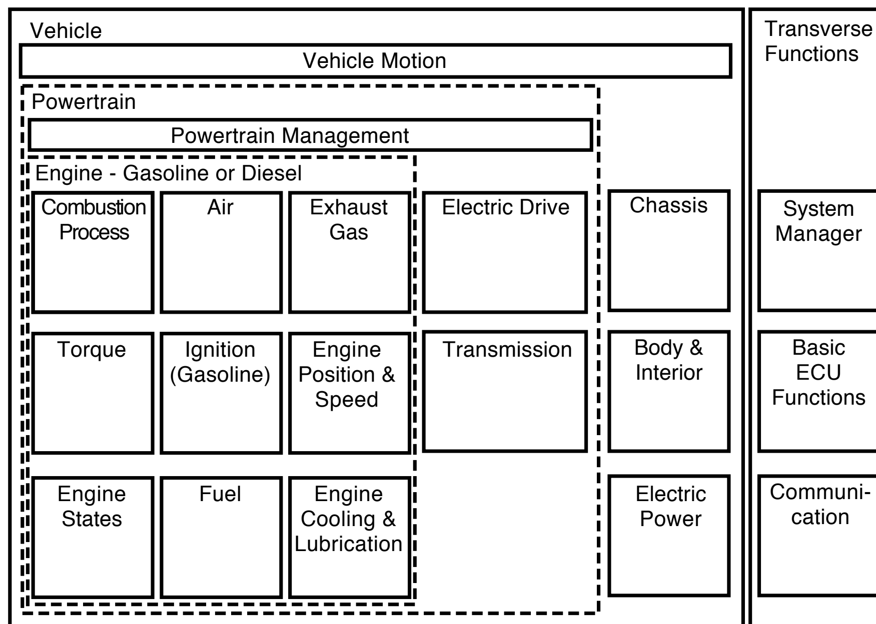
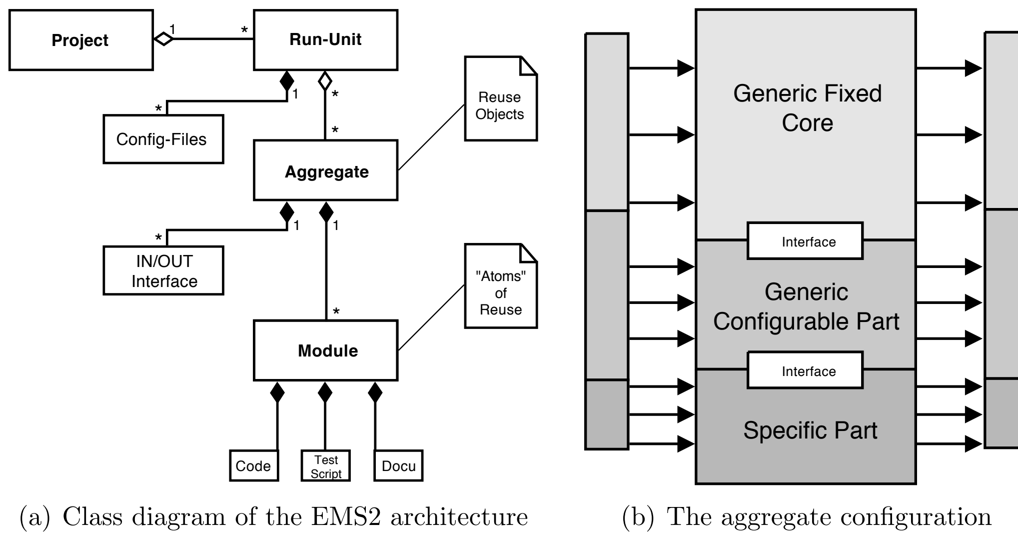


Figure 2.6.: The Engine Management System (EMS2) partitioning according to blocks of functionality.

The way of grouping modules into aggregates was driven by several structuring concepts. First of all, functional coherence was considered. The functionality of a combustion engine management system can be divided hierarchically into units, thus creating a partitioning of the functionality according to the engine management physics and the supporting functio (e.g. comfort or legal requirements). Additionally, the complexity of interfaces and the encapsulation of components was a driving factor for the partitioning definition. Furthermore, it was ensured that the organizational responsibility matches the borders of the defined components to simplify cooperation and minimize communication problems between teams, especially in the communication with the customer.

Due the several structuring concepts, reuse efficiency was not the only goal behind the construction of aggregates and therefore different properties of CBSE (see Section 2.2), had to be considered. As shown in Figure 2.5, target conflicts between reusability (e.g. grouping by functional coherence), maintainability (e.g. organizational responsibilites) as well as flexibility (e.g. encapsulation of components) exist and affect the impact on the reusability. To measure the maximum impact of reusability and maximize reuse efficiency a new grouping of components has to be evaluated. In order to include the concept of aggregates in the measurement



(a) Class diagram of the EMS2 architecture (b) The aggregate configuration

Figure 2.7.: Figure (a) shows a high level model of the reuse structure of the EMS2 architecture for powertrain control units. In Figure (b) the configuration of an Aggregate, split into a fixed core, a configurable part and a project-specific part, is shown. The different parts are connected via an internal interface.

of reuse, a new reuse metric was defined as follows (J. Feder, G. Wirrer; Internal Technical Report 2009; Continental Automotive GmbH):

$$r_p = \left(\sum_{a \in A_{core}} |M_a| + 2 \left| \bigcup_{b \in A_{prj}} M_b \right| + 4 \sum_{p \in P} n_p \right) \frac{|P|}{\sum_{p \in P} m_p}, \tag{2.1}$$

P = set of selected projects
r_p = reuse indicator using P as basis, the smaller the better, 1 is best
A_{core} = set of aggregate core part module versions used in P
A_{prj} = set of aggregate project part module versions used in P
M_x = set of modules contained in aggregate part x
n_p = number of modules not inside aggregates in project p
m_p = number of modules in project p

The first summand of Equation 2.1 is the quantity of modules in the core part of the aggregate (see Figure 2.7(b) Generic Fixed Core), the second summand (weighted double) is the quantity of modules, which are in the aggregate but not in the core part (see Figure 2.7(b) Generic Configurable Part) and the third summand (quadruple weighted) is the number of modules not assigned to an aggregate. The last part of the equation is a suitable normalization factor. The weighting indicates

that modules, which are part of aggregates lead to a higher reuse level (which is desired) than project specific implementations.

The defined metric provides means to measure the reuse level while taking into account the architectural concept of the system. Nevertheless it assumes that the configuration of the architecture is optimal and that the use of modules from the generic part has a higher impact on the reuse than modules from the generic configurable or specific parts. As stated before the motivations for structuring the components to component groups vary and it is not guaranteed that the selected partitioning is optimal as constraints are changing over time.

Also, the metric tends to enforce the selected partitioning by the imposed weighting without giving hints to improvements of the architecture. The drawbacks of this metric are subject to the solutions provided in this thesis. One of the solutions uses network analysis methods which is introduced in the following section.

2.5. Network Analysis

Section 5.1 discusses an approach which provides means to measure the reusability of past and current systems and to evaluate individual components. This approach uses methods from network analysis which became very popular in recent decades in social network analysis. Wasserman stated in [Wasserman and Faust1994] that the application of network analysis in behavioral and social science focuses on the relationships among entities and on the patterns and implications of these relationships.

Knoke described network analysis in [Knoke and Yang2008] as follows:

> Central to the theoretical and methodological agenda of networks analysis is identifying, measuring, and testing hypotheses about the structural forms and substantive contents or relations among entities. The Network perspective emphasizes structural relations as its key orienting principle. Entities may be individual natural persons, small groups, organizations or even nation-states. The central objectives of network analysis are to measure and represent these structural relations accurately, and to explain both why they occur and what are their consequences.

The network perspective allows, similar to the application in social science, the answering of research questions in the software domain by giving a precise formal definition to the aspects of the component-based system and its structural environment. From the network analysis view in the embedded software domain, the component structure can be expressed as patterns or regularities in relationships among interacting units.

Many practitioners describe a software system playful as a living organism, and it may be not that far from reality. Each software component may evolve over time, become distinct, is merged with other components and has strong and weak ties and relationships to other components in the environment which alike can change over time too. The component environment becomes an ecosystem to which many of the key structural measures and notions of social network analysis are applicable. Because of the vast amount of data and information on component-based architectures in industrial systems network analysis methods may not be efficient enough. To overcome this challange heuristics will be discussed in later sections and are introduced in the following.

2.6. Simulated Annealing

Simulated annealing (SA) was proposed as an appraoch to the approximate solution of difficult combinatorial optimization problems by Kirkpatrick, Gelatt and Vecchi [Kirkpatrick and Gelatt1983] in 1983 and independently by Cerny [Černỳ1985] in 1985 based on the work of Metropolis et. al. [2]. This approach is based on ideas from statistical mechanics and motivated by an analogy to the physical process of annealing. It uses the notions of a probabilistic acceptance rule, exploration in the neighborhood of the current solution, and the Boltzmann distribution and thermal equilibrium to guarantee asymptotic convergence to global optima in combinatorial optimization problems [Aarts and Korst1988]. It can be viewed as an enhanced version of the technique of local optimization or iterative improvement, in which an initial solution is repeatedly improved by making small local alterations until no such alteration yields a better solution. Simulated Annealing contains three basic elements:

1. probabilistic acceptance (Metropolis or logistic forms),
2. neighborhood exploration, and
3. a cooling schedule that respects thermal equilibrium.

SA has been demonstrated to be robust and capable of dealing with noisy and incomplete real-world data [Nolle et al.1999, Nolle et al.2002] and when adapted efficiently to optimization problems, SA is often characterised by fast convergence and ease of implementation. These characteristics motivate the choice of SA for NP-hard combinatorical optimization problems in general and for the optimization problem discussed in this thesis in particular. In Table 2.6 a comparison of strategies for optimization problems is presented.

Simulated annealing randomizes this procedure in a way that allows for occasional *downhill moves* (changes that worsen the solution), in an attempt to reduce the probability of becoming stuck in a poor but locally optimal solution. As with lo-

	Accuracy	Complexity	Advantages	Disadvantages
Exhaustive	Always finds the optimal solution	Exponential	High Accuracy	High complexity
Sequential	Good if no backtracking needed	Quadratic $O(N_{EX}^2)$	Simple and fast	Cannot backtrack
Randomized	Good with proper control parameters	Generally low	Designed to escape local minima	Difficult to choose good parameters

Table 2.1.: Comparison of strategies for optimization problems.

cal search, simulated annealing can be adapted readily [Ingber1993] to old (e.g. the traveling salesman problem [Kirkpatrick and Gelatt1983]) and new problems in diverse areas such as neural networks as stated by Ackley [Ackley et al.1985], Israel [Israel and Koutsougeras2002] and Bilbro [Bilbro et al.], pattern recognition discussed by Duda in [Duda et al.2001], circuit design presented by Wu in [Wu and Sloane2002] or exploratory data analysis and clustering problems such as the approach presented in Chapter 6.

SA, as described in [Metropolis et al.1953] (called the Metropolis algorithm), starts with a high temperature T and any initial state. A neighborhood operator is applied to the current state i (having energy E_i) to yield state j (having energy E_j). If $E_j < E_i$, j becomes the current state. Otherwise j becomes the current state with probability, $e^{(E_i - E_j)/T}$ (if j is not accepted, i remains the current state). The application of the neighborhood operator and the probabilistic acceptance of the newly generated state are repeated either a fixed number of iterations or until a quasi-equilibrium is reached. The entire procedure is performed repeatedly, each time starting from the current i and from a lower T. At high T, almost any change is accepted and the algorithm visits a very large neighborhood of the current state. At lower T, transitions to higher energy states become less frequent and the solution stabilizes.

2.7. Expert Systems

As discussed in the previous Section 2.1.3 crucial for the use of component repositories are proper classification and retrieval mechanisms and flexible means for combining components. In the context of the component-based automotive system discussed in Section 2.4, combining components means to partition components (so called modules) to component groups (so called aggregates) which are the basis for reuse. The way of grouping was driven not only by functional coherence but

Algorithm 1 Simulated Annealing

Determine an annealing schedule T_i
Create an initial solution Y_0
while $T_i > T_{min}$ **do**
  Generate a new solution Y_{i+1} which is a neighbor of $Y(_i)$
  Compute $\Delta E = -[J(Y_{i+1}) - J(Y_i)]$
  if $\Delta E < 0$ **then**
    Always accept the move from Y_i to Y_{i+1}
  else if downhill move **then**
    Accept the move with probability $P = e^{-\Delta E/T_i}$
  end if
  Reduce T
end while

also by the expected reuse scope, affiliation to customers or projects, complexity of interfaces and even organizational aspects.

Unlike the goal in the research field of software clustering, to find the best grouping of components to subsystems, i.e. the best cluster of an existing software system [Clarke et al.2003], the goal of our approach is to find the component group or cluster to maximize the reuse of components in a new system. In software clustering most of the techniques determine clusters (subsystems) using either source code component similarity [Muller et al.1993], sets of heuristic rules [Schwanke1991], concept analysis and clustering metrics [Anquetil2000, Hutchens and Basili1985, Lindig and Snelting1997, Van Deursen and Kuipers1999].

2.8. SAE AADL

For the modeling of software architectures a new discipline has recently emerged — model-driven engineering. Currently, the definition of model-driven engineering approaches is the subject of many efforts both from industry and academia. One important concern in these approaches is the choice of the most appropriate languages and tools to be used. Generally, this choice depends on the application domain and on the variety and maturity of the tools that support a particular language. Most of the model-driven engineering approaches under development rely either on UML (Unified Modeling Language), which is a general-purpose modeling language, or on ADLs (architecture description languages) that are usually domain-specific, or on a combination of both. For the framework which will be proposed in Chapter 7 an architecture description language is used.

The SAE Architecture Analysis and Design Language (AADL) [Aerospace2004] is a textual and graphical language used to model and analyze the software and hardware architecture of embedded systems. Feiler at. al. states in [Feiler et al.2006b] that the AADL describes the structure of such systems as an assembly of soft-

ware components mapped onto a hardware platform. Furthermore it is used to describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as temporal requirements) which is a crucial requirement for the analysis of real-time systems.

As stated in [Aerospace2004] the language standard does not specify how detailed the design of the architecture or the implementation of software and hardware components has to be. This allows different levels of abstraction within one model and therefore supports an iterative development approach. Furthermore AADL may be used in conjunction with existing standard languages in these areas (e.g. via an existing UML profile). The AADL describes interfaces and properties of hardware components including processor, memory, communication channels, and devices interfacing with the external environment. Detailed designs for such hardware components may be specified by associating source text written in a hardware description language such as VHDL. The AADL can describe interfaces and properties of application software components implemented in source text, such as threads, processes, and runtime configurations.

The language includes a standardized XML interchange format based on a Meta model specification of AADL to facilitate model interchange and integration of analytical models and supporting tools. The purpose of the SAE AADL is to provide a standard and sufficiently precise (machine- processable) way of modeling the architecture of an embedded real-time system, such as an automotive system or avionic system, to permit analysis of its properties and to support the predictable integration of its implementation. It provides a framework for system modeling and analysis, facilitates the automation of code generation and other development activities, and aims to significantly reduce design and implementation errors.

The AADL core language is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. Extensions can take the form of new properties and analysis specific notations or unique hardware attributes that can be associated with components.

3. Related Work

Component-based software engineering has achieved a lot of popularity in today's software development communities and several researcher have been working on component-based software architectures and component integration, retrieval strategies or component composition. After exploring the background to this research in the previous chapter, this chapter aims to put the thesis in context to related work and discuss similar approaches.

3.1. Component Selection Problem

By software composition usually the composition of components to satisfy functional requirements, where each component possesses a clearly defined interface and functional description, is meant [Bartholet et al.2005].

Component selection methods are traditionally done in an architecture-centric manner, i.e. they aim to answer the question: given a description of a component needed in a system, what is the best existing alternative available? Several Commercial of the Shelf (COTS) component selection approaches exist [Albert et al.2002, Comella-Dorda et al.2002, Mancebo and Andrews2005] which are largely based on the selection and implementation of COTS based on business and functional criteria. Reuse research as it relates to the perspective of component selection generally takes one of three forms:

- Designing reuse metrics and models.
- Representing components.
- Selecting the best set of components.

To predict the costs and benefits of a reuse strategy and enable the measurement of tradeoffs associated with specific components reuse metrics and models are used. Frakes and Terry provide in [Frakes and Terry1996] a extensive summarization of reuse metrics and models.

Component representations support the description of components and range from informal text-based descriptions which where summarized by Frakes and Pole in [Frakes and Pole1994] to formal descriptions introduced by Mili et. al. in

[Mili et al.1994]. More recently to the use of ontologies, domain models and frameworks has come to use.

Finally, there is a need for a method to retrieve the best set of components given metrics and descriptions. Component selection has been shown to be NP complete [Petty et al.2003] while more recently, a polynomial time approximation algorithm for component selection was discovered [Fox et al.2004]. For the selection and identification of components most approaches are based on system functionalities or system architecture [Haghpanah et al.2008, Vescan et al.2008]. Sugumaran and Storey propose a semantic-based approach for the retrieval of components in [Sugumaran and Storey2003]. Other methods include formal methods like model checking [Xie], artificial neural networks [Merkl et al.1994], information retrieval [Maarek et al.1991], decision and utility theory [Alves et al.2005] and genetic algorithms. For the latter Haghpanah et. al [Haghpanah et al.2008] adapted a greedy approach and a genetic algorithm to approximate the component selection problem. Hoover et. al proposed in [Hoover and Khosla1996, Hoover et al.1999] an analytical approach for the design and the change of the design of reusable real-time software. In this approach a combination of genetic and simulated annealing algorithms was used to measure the impact of change to the reusable system. While this method could also be used to partition software to reusable parts no information about the quality of components is taken into account and therefore does not qualify for the problem at hand.

3.2. Software Architecture Analysis

Another approach to evaluate component-based software architectures is based on software architecture analysis. Two basic classes of evaluation techniques, questioning and measuring, available at the architecture level are defined in two important research reports [Clements and Kazman2003] [Abowd et al.1996]. Questioning techniques generate qualitative questions to be asked of an architecture and they can be applied for any given quality. This class includes scenarios, questionnaires, and checklists. Measuring techniques suggest quantitative measurements to be made on an architecture. They are used to answer specific questions and they address specific soft- ware qualities and, therefore, they are not as broadly applicable as questioning techniques [Dobrica and Niemel2002].

A number of methods have been developed to evaluate quality related issues at the software architecture level. SAAM [Kazman et al.1996] and three extensions SAAM for Evolution and Reusability (SAAMER) [Lung et al.1997], Architecture Level Modifiability Analysis (ALMA) [Bengtsson et al.2004], Architecture Level Prediction of Software Maintenance [Bengtsson and Bosch1999], Scenario Based Architecture Reengineering [Bengtsson and Bosch1998], SAAM for Complex Scenarios [Lassing et al.1999], integrating SAAM in domain-Centric and Reuse-based

development [Molter1999], and the architecture trade-off analysis method (ATAM) [Kazman et al.1998]. These are scenario-based methods, a category of evaluation methods considered quite mature. A comprehensice overview about the methodologies was collected by Babar in [Babar et al.2004].

3.3. Evaluation using Metrics

Evaluating a software architecture using a metrics system is often based on the assumption that a object-oriented design is present. Metrics are used for different kinds of calculations of dependencies between and within classes, which can give guidelines on how good a structure the architecture in question has. Rosenberg and Hyatt [Rosenberg and Hyatt1997] define five different qualities that can be measured by metrics for object-oriented design: efficiency, complexity, understandability, reusability, and testability/maintainability.

A extensive collection of Metrics was presented by Harrison et al. [Harrison et al.1998] with the MOOD Framework while Chidamber and Kemerer proposed six object-oriented design metrics in [Chidamber et al.1994]. In their study of the use of object-oriented metrics to determine maintainability, Li and Henry [Li et al.1993] used five of the six object-oriented metrics proposed by Chidamber and Kemerer. The amount of software reuse (the reuse level) in a certain software system can be determined, according Banker in [Banker et al.1993], by the ratio of reused components (or their lines of code) to the total components of the system (or total amount of code lines). Suri et. al presented in [SURI and Garg2009] software reuse metrics to measure the independence of components and argue that more independent components are more reusable.

In [Dumke2003] software measurement is generally described as a process for quantifying the attributes of objects or components in software engineering. This is achieved by using measurement tools for specific measurement goals. This tools that support the measurement process are classified as Computer Assisted Software Measurement and Evaluation tools (CAME tools). The CAME tool area includes tools for model-based software components analysis, metrics application, presentation of measurement results, statistical analysis and evaluation [Dumke and Winkler1997]. Dumke et al. present in [Dumke and Winkler1997] examples of CAME tools with the different possibilities of model based presentation, metrics execution, component evaluation, and measurement education.

A large number of measurement approaches within the context of component-based development where identified by Schmietendorf et al. in [Dumke et al.2002]. Selected examples which picture general starting points for software measurement are shown in figure 3.1.

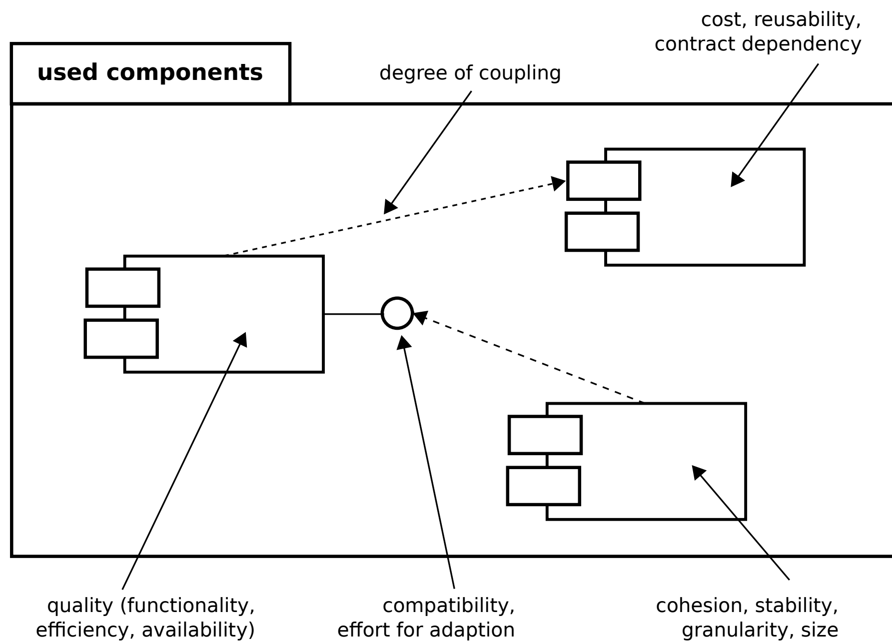


Figure 3.1.: Potential measurement approaches within the context of a component architecture after [Dumke et al.2002].

To be applicable in a large scale software development scope metrics are often composen in frameworks. This software measurement frameworks can be directed on the software process, product and resources components. Software measurement frameworks are proposed by Dumke et al. in [Dumke and Grigoleit1997] and by Fenton in [Fenton and Pfleeger1998]. In table 3.3 examples of metrics from [Dumke2003] for the assessment of software reuse are presented. A comprehensive overview about measurement techniques is provided by Ebert et al. in [Ebert and Dumke2007].

3.4. Evaluation using Human Expertise

The work suggested in the previous section uses formal methods for the evaluation of component-based systems. While this methods provide important support for system architects and engineers they can not completely replace human expertise. While metrics may give a very good value to individual quality requirements architectural decisions are based most of the time on multible objectives. Hence, although metrics can aid in architecture evaluation and are basically the only way of automated evaluation, they cannot replace the evaluation of experts.

The most widely used and known method for architecture evaluation is the Architecture Tradeoff Analysis Method (ATAM) by Kazman et al. [Kazman et al.2000]. Other known architecture evaluation methods are the Maintenance Prediction

Method, which concentrates in evaluating maintainability, and the Software Architecture Analysis Method developed in the Software Engineering Institute of Carnegie-Mellon University, which is mainly used for evaluating quality attributes that are related to modifiability [Bengtsson et al.2004].

3.5. Summary: Measurement Tasks

Subsuming the assessment of current approaches for component selection one can say that no general applicable approach exists. The approach has to be tailored to the task and needs to be based on a formal description of the architecture. Because of the computational complexity, heuristics have proven to provide applicable results. But to create a real benefit in automated component selection approaches they have to interface with CAME tools to measure the tradeoffs associated with specific components.

In [Kunz2010] general short-comings of existing CAME tools where discovered. This shortcomings include a rare native automation, high customization efforts, etc. and result foremost in the need for expert knowledge to analyze measurement results. Furthermore, to apply measurement tools in practical environments, it is necessary to assure a continuous application into a measurement process, which is integrated in the development process. One approach to provide the possibility of an automated tool based analysis of the reuse is the establishment of a customized architecture framework tailored to the development process of the environment with interfaces to selection and measurement approaches. Finally, none of the assessed metrics, discussed by Harrison et al. [Harrison et al.1998] and Dumke [Dumke2003], are applicable for the, required to be automated, measurement tasks at hand, which are:

- Calculate the overall reuse of a layered component-based automotive software system
- Assess individual components for reuse qualification
- Calculate the individual contribution of components to the overall reuse
- Calculate the contribution of component packages to the overall reuse
- Calculate individual components for the reuse contribution to individual packages
- Assess new groupings of components to reusable packages for contribution to the overall reuse

Metric	Type	Calculation
Amortization after Gaffney in [Poulin1997]	Calculation	$(RCR+RCWR/n-1)*R+1$, n = number of expected reuse, R = amount of reused code in the product
COCOMO Modification after Balda in [Poulin1997]	Estimation, Measure	$LM = aiNib$, $N1$ = KDSI for unique code use, $N2$ = KDSI for planned reuse, $N3$ = KDSI from reused code, $N4$ = KDSI of modified components (KDSI = Kilo delivered source instruction)
Cost Benefit after Bollinger in [Poulin1997]	Estimation	Benefit = withoutReuse - withReuse - Reuse Investment
Cost Benefit after Malan in [Poulin1997]	Estimation	Costs = (withoutReuse - withReuse)-ReuseOverhead
RCA [Poulin1997]	Estimation	Reuse Cost Avoidance = Development Cost Avoidance + Service Cost Avoidance
RCR (relative reuse costs) [Poulin1997]	Estimation	Effort for the use of a component without modification (black-box reuse)
RCWR (reuse costs for writing reuse component) [Poulin1997]	Estimation	Effort for the development of a component intended for reuse
Reusable Index [Sodhi and Sodhi1999]	Assessment	Component assessment: 4 - most reusable, ... , 1 - least reusable
Reuse Leverage [Poulin1997]	Estimation	ProductivityWithReuse / Productivity-WithoutReuse
Reuse Percent [Poulin1997]	Estimation	ReusedSoftware / TotalSoftware
RVA [Poulin1997]	Calculation	Reuse Value Added = (TotalSource Statements + SourceInstructionsReused ByOthers)/(TotalSourceStatements - ReusedSourceInstruction)

Table 3.1.: Example metrics for the assessment of software reuse ([Dumke2003]).

Part II.

Reusability Evaluation

4. System Reuse Modelling

In the following sections, an approach to collect data and represent the system architectures as component-graphs is presented. In Section 4.1 the prerequisite nomenclature used in this work for the generation of reuse packages is introduced. For this, the system architectures are transformed and presented as component-graphs and will be called **component reuse networks** in the following sections. This component-graph represents the architecture model, descriped in Section 2.3.2, and captures the design decisions of system architects in projects regarding reuse. In Chapter 5 network analysis measures are applied to this graph representation of the architecture model to evaluate it for key components and reusable groups of components.

4.1. Definition of the Solution Space

In practice, system architecture design decisions are, as described in Section 2.4, based on a variety of formal and informal constraints and presets. Only the knowledge and understanding of experienced system experts ensures the useful integration of a vast amount of needed components to a meaningful system. With years of development, new systems and system components the amount of different features and particular constraints becomes a challenging task, even for the most experienced system architects. To support the experts on design decisions for future systems the knowledge and experience integrated in past and ongoing systems has to be captured in a formal way.

To automatically capture knowledge about components and component groups, data is collected from a company's project repository following the approach presented in [Hobelsberger et al.2010]. For the approach to be applicable, it is assumed that information about the history of the company (e.g. finished projects or systems in the maintenance phase) and current systems (e.g. under active development) exists in form of project repositories/databases. Commonly, the information in a project repository includes an accurate and detailed picture of the organizational structure, the software architecture and its components.

A project can define a system or a subsystem and therefore can be part of bigger architectures. A project can also be structured in subprojects representing different

stages in the development process. This knowledge represents the experience and design decisions of system architects of past and current projects.

4.1.1. Data Collection

The sum of component-based system architectures from the project repository (e.g. table 4.1.1) is defined as a set $A = \{a_1, a_2, ..., a_n\}$ where $|A|$ denotes the number of elements and a corresponds to a candidate system architecture in the set A. The candidate system architecture $a = \{c_{a1}, c_{a2}, ..., c_{an}\}$ defines a distinct number of components from the data set D and represens a specific project of the company. The data set $D = \{c_1, c_2, ..., c_n\}$ contains all the candidate components c from which a system architecture can be built (see Figure 4.1). A component c is defined as a tuple $c = (N, C)$ where N is a unique identifier of the component (e.g. ID or name) and C is a class attribute (e.g. functional group or package ID).

Project	Components
Project 1	c_1, c_2, c_3,c_4, c_5,c_6
Project 2	c_4, c_6, c_8, c_9
Project 3	c_1, c_4, c_6, c_9, c_{16}
...	...

Table 4.1.: Project repository example. For the full list of components of the example project repository used in Section 5.3, see Appendix A.

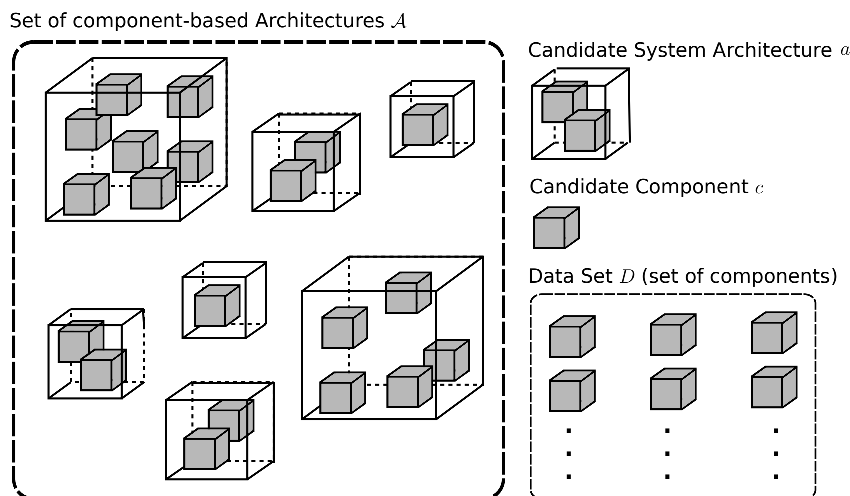


Figure 4.1.: Set A of component-based system architectures including candidate architectures a composed of candidate components c out of the data set D.

To be both language independent and evaluable through an algorithm the software architecture is transformed into a graph G. For finding meaningful clusters of com-

ponents or subsystems (e.g. reusable component groups) the graph G is partitioned into sets of non-overlapping clusters that cover all the vertices in the graph.

For this purpose, a component-graph, based on the common use of components in past projects, is constructed. Figure 4.2 shows the basic process to gain this experience about reused components and the construction of the component-graph.

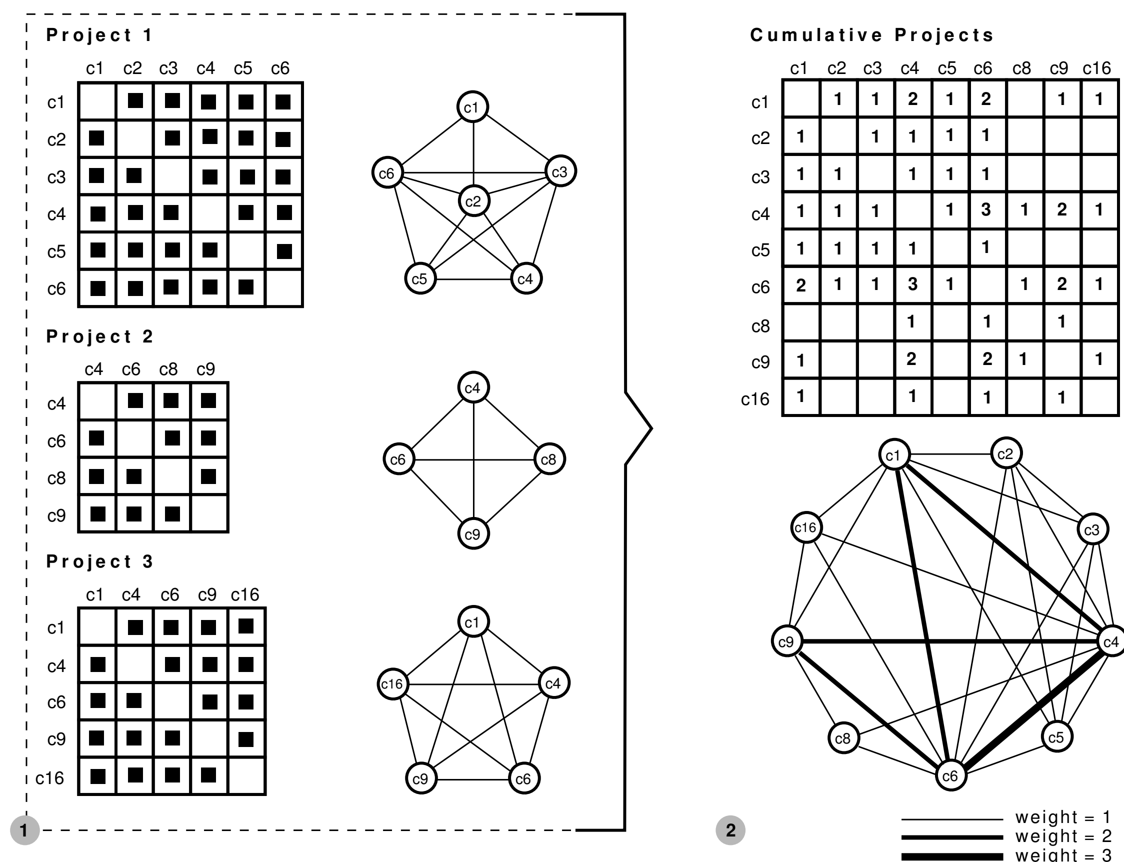


Figure 4.2.: Schematic view on the generation of the component-graph. The data for the network analysis is generated by the cumulated adjacency graphs (1) of the projects ($a = \{c_1, c_2, \ldots, c_n\}$, D is the collection of all possible modules). The final weighted component-graph (2) over all projects is analyzed with network analysis measures.

An adjacency matrix (see Figure 4.2 - 1) is generated for each project, based on the used components using the bills of material stored in the company repository for all projects. Every project differs in number and kind of components as not all of the modules are needed, alternative components exist and new components are implemented in case of specific needs. In this analysis no information about data dependencies (as it is not available in most repositories) is considered. Only the common use in a project, indicating compatibility from a functional point of view, is taken into account. Consequently, in this matrix, each component used in a dedicated project has a connection to every other component used in the

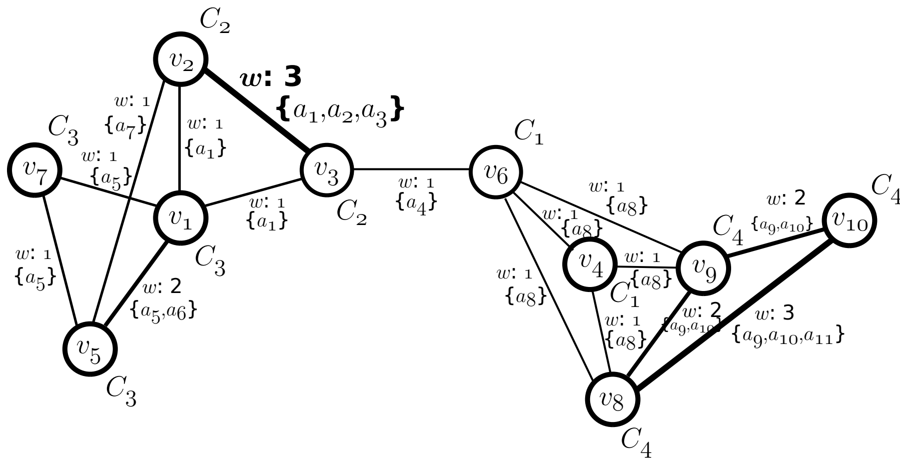


Figure 4.3.: Exemplary graphical representation of a system, generated with the methodology shown in Figure 4.2, as a component-graph G. The components v_2 and v_3 were both used in the candidate architectures a_1, a_2, a_3 represented by an connecting edge with weight $w = 3$ and a label with the corresponding candidate architectures. Each vertex has a node label class C_i representing the cluster a particular vertex is assigned to.

same project. Out of this, a cumulated adjacency graph (see Figure 4.2 - 2) is generated, which is transformed into a weighted component-graph (using a *degree of relationship* as weight between two components, defined as the total number of common use over all projects).

4.1.2. Generating the Component-Graph

A set of component-based system architectures A is represented as a *weighted*, *vertex-labeled* and *edge-labeled* component-graph $G = (V, E)$ (see Figure 4.3) where V is the finite set of vertices $\{v_1, ..., v_n\}$ and the number of vertices $n_v = |V|$ is the *order* of the component-graph.

Each individual component c in the data set D defined by a candidate system architecture a becomes a vertex v while the class attribute C (representing the cluster a particular vertex is assigned to) becomes a label of the corresponding vertex. The set of vertices V is a subset of the data set D ($V \subseteq D$) such as each vertex v in V is mapped to a distinct element of the data set D.

A vertex v corresponds to a candidate component in the vertex set V. The vertex set is partitioned into p classes $C_1, C_2, ..., C_p$, one for each set of candidate components where the class attribute of the candidate component corresponds to the class C_i. $|C_i|$ denotes the number of elements in the set C_i while $|C| = n$ denotes the number of unique classes. The class C for a set of components represents a

component group intended for reuse. The classes C_i and C_j are defined as *adjacent*, denoted by $C_i \sim C_j$, if components (vertices) in C_i and C_j are constrained in the same candidate architecture a and therefore are connected by edges.

The set E contains the *edges* $\{e_1, ..., e_m\}$ of the component-graph where $m = |E|$ denotes the number of elements in the set E (the *size* of the component-graph) such that $e_i = \{v_{i_1}, v_{i_2}\}$, with $v_{i_1} \neq v_{i_2}$ and $v_{i_1}, v_{i_2} \in V, i = 1, ..., m$. The pairwise use of components in a candidate system architecture a form the edges (E) of the component-graph.

In an *undirected* component-graph, each edge is an unordered pair (v, u) representing a connection between two components v and u. The vertices v and u are called *endpoints* of the edge.

The component-graph $g = (V_1, E_1)$ is defined as a *subgraph* of $G = (V, E)$ if $V_1 \subset V$, and $E_1 \subset E$. Every candidate architecture a represents a connected subgraph g in the cumulated component-graph G. In Chapter 5 analysis methods are discussed which are applied to the full component-graph as well as to a subgraph.

The edge labeling of an edge is the function $\nu : E \rightarrow L_E$ that assigns a label $L_E(e, G)$ from the finite label set L_E to each edge e of the component-graph G. An edge label $L_E = A', A' \subset A, \forall v, u \in A'$ of the component-graph G represents the candidate system architectures the endpoints of the edge have in common.

The node labeling of a vertex is the function $\mu : V \rightarrow L_V$ that assigns a label $L_V(v, G)$ from the finite label set L_V to each vertex v of the component-graph G.

In a *weighted* component-graph, a weight function $\omega : E \rightarrow R$ is defined that assigns a weight to each edge. The weight of the edge is defined as the number of candidate system architectures $|A'|$ the endpoints have in common.

If the set of vertices $\{v, u\}$ are element of E ($\{v, u\} \in E$), v is defined as a *neighbour* of u and u is said to be adjacent to v. Furthermore two edges are called adjacent if they have a common endpoint. The set of neighbours for a given vertex v is called the *neighbourhood* of v and is denoted by $\Gamma(v)$.

Two vertices v, u of a component-graph $G = (V, E)$ are *connected by a path* if there is a sequence of vertices $v_{r_1}, v_{r_2}, ..., v_{r_p}$ such that $v = v_{r_1}, u = v_{r_p}$, and $\{v_{r_i}, v_{r_{i+1}}\} \in E, i = 1, ..., p-1$. A component-graph $G = (V, E)$ is *connected* if any pair of distinct vertices $v, u \in V$ are connected by a path.

The *eccentricity* of a vertex v is the maximum distance from v to all other vertices in G. The *diameter* is the maximum eccentricity. The *periphery* of a component-graph is the set of vertices with an eccentricity equal to the diameter. The *radius* is the minimum eccentricity while the *center* is the set of vertices with eccentricity equal to radius.

For context specific analysis, different component-graphs can be generated, e.g. with focus only on closed projects in a specific customer cluster to analyse the

reuse on a per customer basis or limited to a special functionality to analyse the reuse level of specific component groups.

4.2. Summary: An Experienced-Based Component-Graph

Following the steps described in this chapter results in a component-graph those connections represent the relationship between components over different projects. This relationships capture and represent the design decisions of system architects and engineers and hence reflect the informal experience and knowledge of experts in an formal way. The fully automated transformation of the information to a formal component description, the component-graph, is mapped to standard graph theory definitions and therefore allows the application of network analysis measures presented in the next chapter.

5. Network Analysis Based Reuse Measurement

Once the network, based on definitions described previously (Section 4.1), is constructed, network analysis techniques can be applied. The network analysis focuses on the connections or relationships between components and not on the attributes of individual components. One of the most relevant features of graphs representing real systems is the application of clustering (or community structuring as it is referred to in social network analyis), i.e. the organization of vertices in clusters, with many edges joining vertices of the same cluster and comparatively few edges joining vertices of the other clusters. In Section 5.1 measures for the identification of central components in the network are discussed followed by a clustering approach in Section 5.2 which is applied on the example component repository in Section 5.3. Metrics to measure the network regarding reuse are introduced in Section 5.4.

5.1. Centrality Measures

Beside clustering, one of the primary uses of graph theory in network analysis is the identification of important, central or prestigious components in a network. Central components are more visible because of their links to other components. Prestigious components are more visible because they were more frequently chosen.

Centrality and prestige measures can be useful to understand the potential flows of information or resources as well as constraints on components in a network. Once interesting components or component groups are identified through network analysis, quantitative or qualitative methods can be applied to explore or make inferences about the attributes of the components. A component is important or central in the reuse context, if it has direct connections to a, relative to the number of components, high number of other components (and therefore has a high quality [Gaffney et al.1989, Lim1994]) or is a gatekeeper to key components (e.g. components with a high reuse or components, which are marked as crucial by an expert).

To measure this type of centrality the concepts of degree, eigenvector centrality and betweenness centrality are applied. They are defined as follows:

Degree of a component: The degree of a component is the number of edges connected to it (see Figure 5.1, Graph 1). In social network analysis, this parameter reflects the popularity of an actor. This means that most popular components are those maintaining the highest number of relationships. In the component reuse network, the degree of a component corresponds to the number of other components it was used with in the same project context. If versions and revisions of components are considered, a relatively high number of direct connections may assert that the component did undergo a lot of changes over the past projects while one conclusion of this could be to treat the component in future projects as customer or project specific implementation and remove it from the reuse library.

Weighted degree: In case of weighted component reuse networks (see Figure 5.1, Graph 2-4), the degree of a component may vary in its expressiveness. A component with a high degree may not necessarily be well connected to the network because all its edges may be weak. Alternatively, a component may be strongly attached to the network if the weight of all its connections is high. The weighted degree of a component is defined as the sum of the weights of all edges connected to it.

Eigenvector centrality: Eigenvector centrality is a measure of the importance of a node in a network. It assigns relative scores to all nodes in the network based on the principle that connections to high-scoring nodes (nodes that have a high level of reuse) contribute more to the score of the node in question than connections to low-scoring nodes [Wasserman and Faust1994]. In [Bonacich2007] Bonacich argued that the eigenvector makes a good network centrality measure. The eigenvector weights contacts according to their centralities unlike the degree, which weights every contact equally. Bonacich states that eigenvector centrality can also be seen as a weighted sum of not only direct connections but indirect connections of every length and thus takes the entire pattern in the network into account.

Betweenness centrality: With the criteria betweenness centrality the number of shortest paths traversing that particular vertex (corresponding to a component) is measured. The betweenness centrality affects the question how other components (so called gatekeepers) control or mediate the relations between dyads that are not directly connected (see Figure 5.1, Graph 4, vertex v_3 or vertex v_6). It measures the extent to which other components lie on the geodesic path (shortest distance) between pairs of actors in the network. Betweenness centrality is an important indicator of control over information exchange or resource flows within the network [Knoke and Yang2008]. Given a component v and a graph G it is defined as:

$$B_c(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{5.1}$$

with $\sigma_{st}(v)$ as the number of shortest paths from s to t via v, and σ_{st} is the total number of shortest paths between s and t.

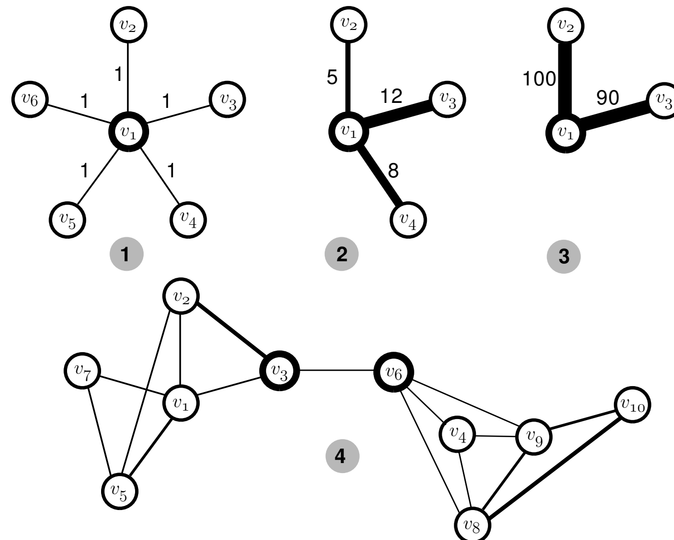


Figure 5.1.: Graphical representation of patterns in the reuse network. Graph {1} represents a vertex (v_1) with high degree (5) but low weighted degree. It corresponds to a component which was used often but in different contexts (systems) and therefore has a low reusability. Graph {2} and {3} show vertices with high reusability while Graph {3} represents a pattern which would lead to a high eigenvector centrality of vertex v_1. Graph {4} shows the concept of gatekeepers (v_3, v_6).

5.2. Cohesive Subgroups

To built clusters or groups of components, the concept of *cohesive subgroups* was used. Cohesive subgroups are subsets of components among which relatively strong, direct, intense, frequent or positive ties can be observed.

Subgroups can be determined by links between components, by structural similarity, or by clustering. For the approach presented in this work, clustering methods are used. Clustering methods are computational ways to iteratively select nodes that are more closely linked to each other than to others in the network. In the past few years, increased attention in the physics community has been directed to *community detection* in large network graphs. Physicists have used analogs to statistical mechanics, bridge circuits, and other physical systems to detect cohesive subgroups or clusters.

Girvan and Newman presented in [Girvan and Newman2002] a clustering method based on high edge betweenness scores. It is based on the assumption, that com-

ponents in communities have more *traffic*, as, e.g., the information flow between components in two communities (e.g. a subgraph within which vertex to vertex connections are dense, but between which connections are less dense). In Figure 5.2 such a network structure is shown.

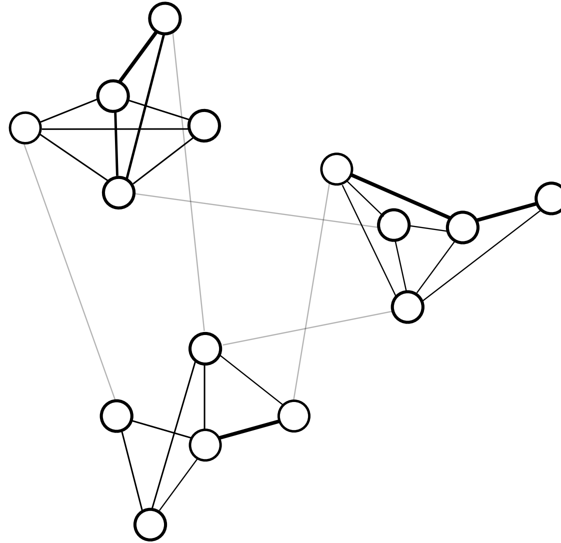
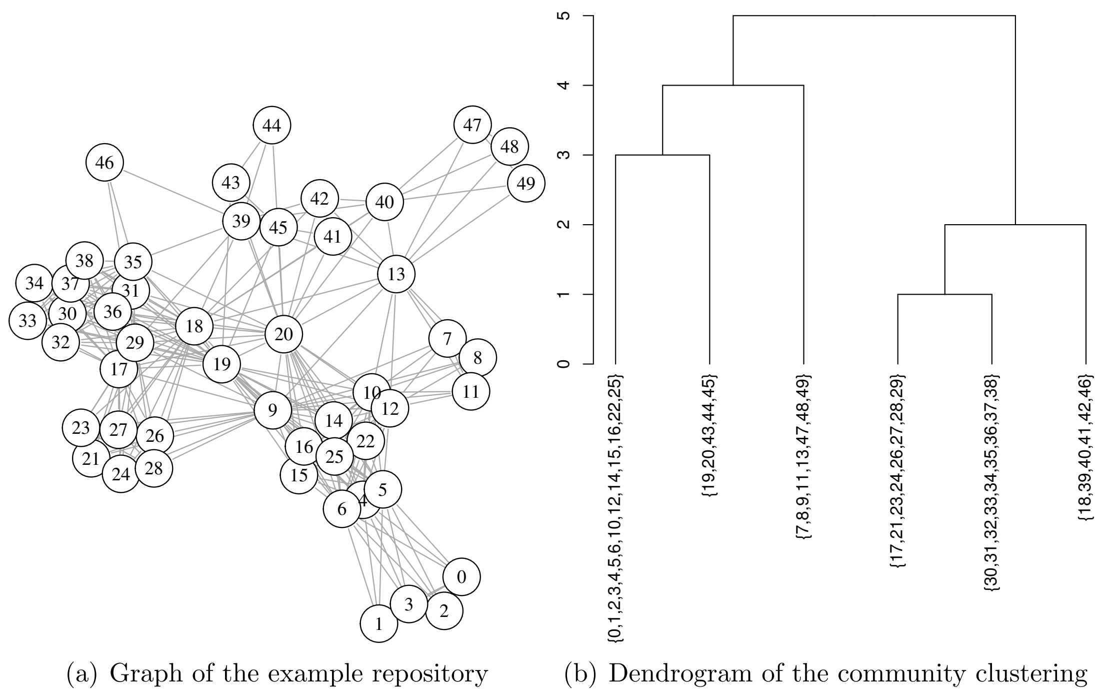


Figure 5.2.: The schematic representation of a network with community structure (as presented in [Girvan and Newman2002]). It shows three communities (with densely connected vertices) with much lower density connections (gray lines) between them.

This method has proven in several experiments to detect meaningfull clusters but is, due to high complexity to calculate the edge betweenness in large networks $O(m^2n)$ (where m is the number of edges and n the number of vertices), only applicable for small networks with up to a few thousand vertices [Newman2004]. The stopping point for clustering is determined by locating peaks in the modularity of the graph. The modularity measures when the division is a good one, in the sense that there are many edges within communities and only a few between them. The algorithm proposed by Girvan et. al. is simply stated as follows [Girvan and Newman2002]:

1. Calculate the betweenness for all edges in the network.
2. Remove the edge with the highest betweenness.
3. Recalculate betweennesses for all edges affected by the removal.
4. Repeat from step 2 until no edges remain.

The community clustering method qualifies in particular for the clustering of components with regard to interface dependencies to create packages with a strong interface encapsulation.



(a) Graph of the example repository (b) Dendrogram of the community clustering

Figure 5.3.: Figure (a) shows the graph of the example repository. The layouting of the graph shows found clusters (e.g. IDs 0,1,2,3). Figure (b) shows the corresponding dendrogram.

5.3. Application of Network Analysis

Once the component reuse network is generated the standard network analysis measures discussed in the previous sections can be applied to the weighted component-graph G. In the following, the measures are discussed by the use of the example project repository. By taking the repository shown in Appendix A as an example the result of Newmanns [Newman2006] community structure clustering algorithm is shown in Figure 5.3(a). Several clusters could be identified and the individual components are listed in the dendrogram in Figure 5.3(b), which could be used to support an architecture expert in the process of selecting relevant (in the context of functional dependencies and reuse) components from the repository.

In Figure 5.4, the result of the centrality analysis is shown. The eigenvector centrality is plotted versus the betweenness centrality. A component with very high betweenness but low eigenvector centrality may be a critical gatekeeper to a central component. The analysis shows that component 20 is an example for such a gatekeeper component. Likewise, a component with low betweenness but high eigenvector centrality may have unique access to central components as, for example, the component 37 in Figure 5.4.

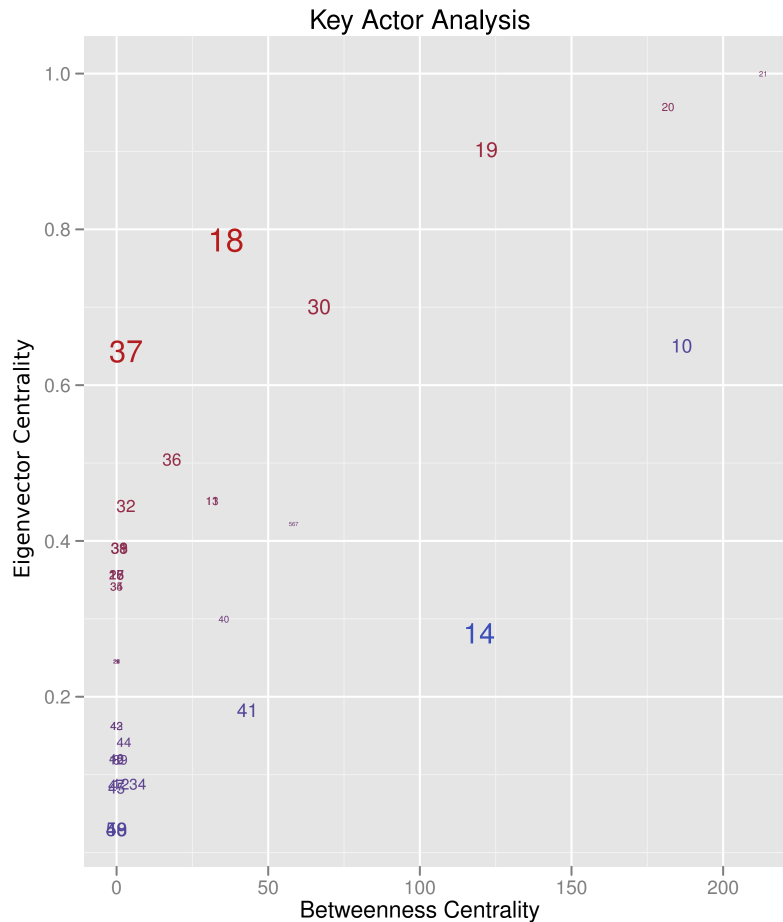


Figure 5.4.: The key component analysis. The IDs of the most important components in the network analyzed by eigenvector centrality and betweenness centrality.

5.4. Reuse Graph Metrics

In the following, component reuse network metrics, which are used to evaluate the network regarding reuse, are introduced and preliminary definitions are provided. This metrics will be incorporated in the heuristic optimization approach presented in the following chapter, to evaluate and optimize the component reuse network.

5.4.1. Preliminaries

To measure the immediate adjacency of a vertex, the *degree* of a graph is used. An edge connects two vertices which are said to be *incident* to that edge or equivalently, that edge is incident to those two vertices. The *degree* $d_G(v)$ of a vertex v in a

graph G is the number of edges incident to v and is denoted by $deg(v)$. The *maximum degree* of a graph G, denoted by $\Delta(G)$, and the *minimum degree* of a graph, denoted by $\delta(G)$, are the maximum and minium degrees of its vertices.

In a weighted undirected graph G, the *weighted degree* of a vertex v is defined as the sum of the weights of the edges incident to v in G divided by the weight of the vertex w_v. Let $W(G)$ be the sum of the weights of all vertices. For a vertex v we define the weighted degree $d_w(v, G)$ in G as follows:

$$d_w(v,G) = \frac{w(\Gamma(v))}{w_v} \tag{5.2}$$

The weight $w_{v,u}$ of the edge(v, u) is calculated based on the number of candidate system architectures a in which the vertices (or components respectively) were used together. The weight $w_{v,u}$ is defined as:

$$w_{v,u} = \sum_{i=1}^{n} a_i, \forall\{v,u\} \in a_i \tag{5.3}$$

5.4.2. Reuse Popularity

According to the definition of degree in social network analysis, where the number of connections represent the popularity of an actor in a network, the popularity of a vertex v, representing a component in the weighted network, is defined as **component reuse popularity** rp_v:

$$rp_v = \frac{d_w(v,G)}{d_G(v)} \tag{5.4}$$

The *reuse popularity* measures how often a component in the network was reused. It shows, following the construction method the network was generated with, which components were used together, how often and in which context. A value of $rp_v > 1$ (Figure 5.1, graph 3, vertex v_1) indicates a frequent reuse, while $rp_v = 1$ (Figure 5.1, graph 1, vertex v_1) equals no reuse of the component. As defined in Section 4.1, the weight of the edge of a vertex corresponds to the pairwise use of a component in the same candidate architecture a.

5.4.3. Component Rank

To measure the importance and impact of each component in the network, a ranking measure is introduced. The **component rank** cr, derived from the page rank equation [Page et al.1998], is defined as:

$$cr(v) = d \sum_{u \in \Gamma(v)} \frac{r(u)}{d_G(u)} \tag{5.5}$$

where v represents a vertex, $\Gamma(v)$ is its neighbourhood, d is a factor used for normalization and $r(v)$ and $r(u)$ are rank scores of the vertex v and u, respectively. $d_G(u)$ denotes the degree of a vertex u, which is the number of links to other vertices (see Section 5.1 and Figure 5.1 Image $\{1\}$ where node v_1 has a degree of 5). As only the probability of connected vertices [1] is important in the context of reuse clustering, the damping factor $(1 - d)$ for the so called random surfer problem [Page et al.1998], where it is possible to visit states in the graph which are not directly connected, is not applied.

5.4.4. Component Package Complexity

With the **component package complexity** cpc, a measure for the complexity (in terms of reuse) of a reusable package is introduced. A package is defined by the structure of a connected subgraph g of G. The *component package complexity* measures the graph distance, the *eccentricity*, of a vertex v to all other vertices in the subgraph g. If the eccentricity ecc of the package is $ecc > 1$, the package contains gatekeepers, which means that there exist components in the package that have not been used in common in a candidate architecture. This conflicts with the intention of reuse to have high quality components while the quality of a component is defined on the test case coverage (e.g. the components, which have no direct connection, are not tested together). The integration effort for such a package, which is measured by the probability that the integration fails, is increased with every component with no common use in a previous architecture.

5.5. Summary: Network Analysis Reuse-Metrics

Network analysis measures, which are applied in broad field in social sciences, provide powerful means to analyze relationships between components. The extracted information about the network and on individual components helps experts with

[1] The reuse is defined by the connections between components. The grouping of components to reusable packages, with no connection to each other is not allowed.

the design decisions and supports the extraction of relevant information from the component repositories. Unfortunately, the vast amount of information results in huge and very dense component reuse network, which results in poor analyzation results of the network analysis algorithms. Furthermore the control about clustering decisions of the algorithm has proved to be very limited and depends strongly on the topography of the generated network. One solution is the segmentation of the component reuse network in smaller graphs or the restriction of the information. Another solution is the combination of network analysis measures with a heuristic optimization approach, which is presented in the next chapter. In this optimization approach the metrics presented in the previous section are integrated in the optimization function.

6. Reuse-Oriented Component Package Generation

In Chapter 5, the *component reuse network* was analysed for central and important components with network analysis methods and component clusters were created with community clustering algorithms. While the extracted information about the network as well as individual components help system architects or system engineers in the selection of reusable components from a company's project repository, it does not provide enough control (the success of the clustering depends on the topography of the generated network and fails in the case of very dense graphs) over the generation of reusable packages.

To overcome these restrictions, a new approach for the selection of software components and their clustering to reusable packages is presented. The approach described in the following sections provides means to select and group components to reusable packages such as to maximize the reuse not of individual components but of component packages without the need to change the package structure (add or remove components). To achieve this, the components are combined in a manner that maximizes the common usage in candidate architectures. This results in 2^n possible combinations of connected components which explains why exhaustive search can not be feasible for this type of problem.

Therefore an optimization approach based on **simulated annealing** (SA) for the selection and grouping of components (to optimize reuse in component-based software architectures) was developed. When adapted efficiently to optimization problems, simulated annealing is often characterised by fast convergence and ease of implementation for real-world problems. These characteristics motivate the choice of SA for NP-hard combinatorial optimization problems such as the one described above.

Simulated annealing is a candidate search method. A SA algorithm repeats an iterative neighbour generation procedure and follows search directions that improve the objective function value describing the *fitness* of a solution. While exploring the solution space, the SA method offers the possibility to accept worse neighbour solutions in a controlled manner in order to escape from local minima.

The classical simulated annealing approach (as described in Algorithm 1) maintains only one solution at a time and whenever it accepts a new solution, it must discard the old one. There is no history of past solutions and as a result, good solutions

(or in the extreme case the global optimum) can be discarded and may never be regained. But then, this enables SA to handle NP-hard optimization problems. The probability to reach a global minima depends on the topography of the input problem and the parameters (e.g. the temperature profile).

SA is a variation of the *hill-climbing* algorithm [Russell and Norvig2009]. Both start off from a randomly selected point within the search space. Unlike the hill climbing algorithm, the new candidate solution is not automatically rejected if the fitness of a new candidate solution is less than the fitness of the current solution. Instead, it becomes the current solution with a certain transition probability $p(T)$. This transition probability depends on the difference in fitness ΔF and the temperature T. With SA applied to the type of problem here presented, the temperature is an abstract control parameter for the algorithm rather than a real physical measure (see Section 2.6).

In the following sections, the concept of simulated annealing is adapted to the component clustering problem at hand and extended to be population oriented. In Section 6.1, prerequisite preliminaries, the necessary parameters and the fitness function of the optimization algorithm are introduced. In Section 6.2, the algorithm for the combination of components to reusable packages is described and will be discussed by applying it to the example component repository A in Section 6.3.

6.1. Simulated Annealing for Component Clustering

In this section, preliminary definitions about simulated annealing and the respective parameters as well as the modification and adaption of those parameters to the actual optimization problem are introduced. The adaptation of SA to an optimization problem consists in general of the definition of its specific components [Ingber1993] which are:

- representation of a solution (and the initial solution) of the problem,
- the neighbour generation mechanism and the neighborhood for the solution space exploration,
- a method for an objective function value calculation,
- and a cooling scheme (and possible parameter adaptation methods) including stopping criteria.

6.1.1. Initializing the Solution and Parameters

In the following the specification and the respective parameters of the SA algorithm are presented.

6.1.1.1. Solution Representation

The solution representation is an important component of simulated annealing. It has to be designed to allow an easy generation of neighbors and a fast calculation of the objective function value degradation ΔF. Furthermore, it must guarantee accessibility of the entire solution space at any point in time of the algorithm run. With the representation of the component-based software architecture as a graph as defined in Chapter 4 Section 4.1, the solution space fulfills all of those specifications.

6.1.1.2. Feasible Solution

Given a graph $G = (V, E)$ a feasible solution will be a partition of *connected* subgraphs from a number of vertices $v > 1$, which are connected by a path. A feasible solution does not mandatorily need to have all subgraphs to have a positive value of the objective function, as the clustering of component packages should also allow the grouping of components to packages with no reusability and therefore allow to create clusters of components which will be considered as project specific implementations.

6.1.1.3. Initial Solution

The starting set of solutions is the generated reuse component graph G. The classes C of the candidate components specify each individual solution of the solution set. The classes, and therefore the solutions, can be generated randomly (e.g. each candidate component is its own solution, the solutions are uniformly distributed) or predefined solutions (e.g. previously defined reuse packages or components that are in the same solution because of expert decisions) are assumed. Unlike the standard simulated annealing described by Kirkpatrick and defined in Chapter 1, the adaption for the optimization problem at hand, optimizes not one solution but a set of solutions, which is called, analog to genetic algorithms, the **population** of the algorithm.

6.1.1.4. Solution Neighborhood

The neighborhood Γ of a selected candidate solution (as seen in Figure 6.1) is defined by the structure of the underlying graph G. A neighbor of a candidate solution c_i is any component c_j which is adjacent to c_i. The neighbourhood $\Gamma(v)$ of a candidate component is restricted to the components which are connected to c_i by a path through the network.

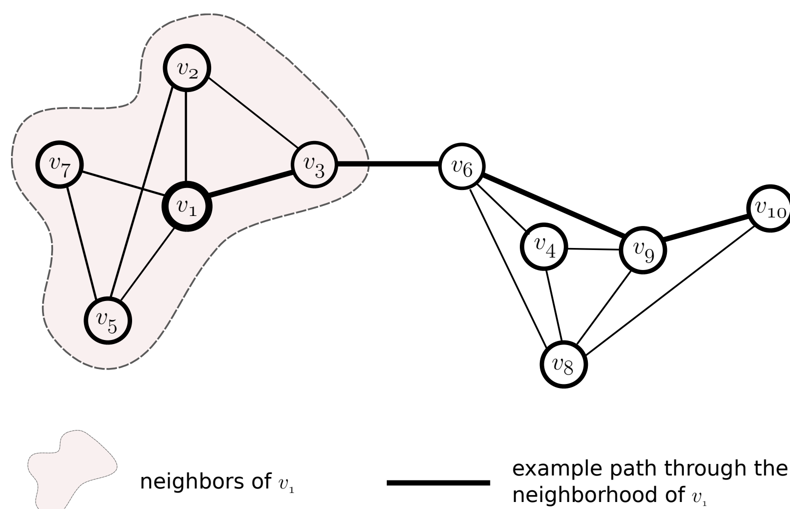


Figure 6.1.: Example of possible neighbors and the connected path of a candidate component. In this example, the neighborhood $\Gamma(v_1)$ of the component v_1 is formed by the adjacent components $\{v_2, v_3, v_5, v_7\}$ while the thick line marks an example of a connected path > 1 through the network starting with the vertex v_1 to the vertex v_{10}.

Because of the vast number of solutions in the search space, it is practically insufficient to choose in each run only one neighbour of the current solution as new candidate solutions. Therefore, it is recommended (at least in the exploration phase of the algorithm) to choose a number $x \geq 1$ of neighbour candidate solutions in a random direction of the current solution in order to progress in an acceptable time through the search space [Nolle et al.2001]. This number could either be a fixed step width sw of neighbours at each iteration (e.g. $sw = \frac{|\Gamma(v)|}{2}$), have an upper limit $sw_{\max}$, may be choosen randomly between those or can be adapted on-line while the algorithm performs (e.g. start with a high step width and degrade it towards the end of the run). This on-line adaption of the step width can be linked to the lowering of the temperature, coupled with the number of iterations or even respond to an equilibrium.

If the maximum step width is chosen to be to small and the starting point for a search run is too far away from the global optimum, the algorithm might not be able to get near this optimum before the algorithm *freezes*, i.e the temperature becomes so small that $p(T)$ is virtually zero and the algorithm starts to perform only hill climbing. In that case, it will get stuck in the nearest local optimum. If, on the other hand, the step width has been choosen to be to large, and the peak of the optimum is very narrow, the algorithm might well approach the global optimum but never reaches the top because the steps are too large and new candidate solutions might miss the peak. Hence, there is always a trade-off between accuracy and robustness in selecting the appropriate step width.

6.1.1.5. Fitness Function

The **fitness function** F is used to describe the quality of a solution in simulated annealing or genetic algorithms and is a particular type of objective function. To evaluate the *fitness* of a solution p, the current solution p is evaluated by an objective function $F(p)$ at each iteration. For each move, the objective difference $\Delta F = F(p') - F(p)$ is evaluated. For maximization problems p' replaces p whenever $\Delta F > 0$. Otherwise, p' could also be accepted with a probability P.

The objective function $F(p)$ for the optimization of component reuse is defined as a linear metric in Equation 6.1. For a given solution p, defined by the candidate components $\{v_1, ..., v_n\}$, the objective function is calculated by the product of the number of candidate components and the number of common candidate architectures multiplied with the sum of the candidate component ranks cr which serves as imbalance factor of the function and values the selection of a high ranking component (a component with a high reuse popularity) in case of a draw.

$$F_{\{v_1,...,v_n\}} = |\{v_1, ..., v_n\}||\{a_1, ..., a_n\}| \sum_{\forall v \in \{v_1,...,v_n\}} cr(v), \forall a \in \{v_1, ..., v_n\} \tag{6.1}$$

In the component based automotive system architecture described in Section 2.4 a desired property for a generated reusable package is, to consist of a high number of components but be still flexible enough for the reuse in many new architectures without modifying the package. The fitness function F favours, due the number of candidate components as the main weighting factor, the construction of few but big clusters. With the use in common candidate architectures as another factor in F it is guaranteed that the packages are, without modification, reusable in the same context. This context is defined by the candidate architectures the candidate components have in common.

The fitness of the complete system (the population) is the sum of the objective functions of all solutions $\sum_{i=1}^{n} F(p_i)$. For the application of simulated annealing in non-trivial industry problems, the fitness function ideally nearly correlates with the goal of the algorithm and yet has to be computed quickly. Therefore, the execution speed is very important as the algorithm must be iterated many times in order to produce usable results.

6.1.1.6. Temperature and Cooling Schedule

With the use of simulated annealing, several aspects of the algorithm must be considered carefully. These are in particular the starting temperature T_0, the rate or cooling schedule α which indicates how much the temperature is decreased, the

ending temperature T_{min}, the number of candidate iterations $I_{candidate}$, and the stopping criterion.

The algorithm starts with a high temperature T_0, which is subsequently reduced slowly, usually in steps, startig from the initial value T_0 and using an attenuation factor $\alpha (0 < \alpha < 1)$. The temperature can be controlled by a cooling scheme specifying, how the temperature should be progressively reduced to make the procedure more selective as the search progresses to neighborhoods of good solutions. An overview about different cooling schedules and a respective comparison is given in [Geman and Geman1993, Huang et al.1986]. The following equation shows the standard cooling function introduced by Kirkpatrick [Kirkpatrick and Gelatt1983]:

$$T_{n+1} = \alpha \; T_n \tag{6.2}$$

There exist theoretical schedules guaranteeing asymptotic convergence towards the optimal solution [Abbasi et al.2010]. This, however, requires infinite computing time. In practice, much simpler and finite computing time schedules are preferred even if they do not guarantee an optimal solution.

The starting temperature T_0 should be high enough that the probability of accepting a worse solution is, at least, of 80 % in the first iteration of the algorithm [Kirkpatrick and Gelatt1983]. This requires a temperature T_0 larger than the maximum difference in energy between any configurations. Such a high temperature allows the system to move to any configuration which may be needed, as the random initial configuration may be far from the optima. The decrease in temperature must be both gradual and slow enough to guarantee that the system can move to any part of the state space before being trapped in an unacceptable candidate minimum.

At the very least, annealing must allow $N/2$ transitions, because a global optimum may differ from an arbitrary configuration by at most this number of steps (in practice, annealing can require polling several orders of magnitude more times than this number [Duda et al.2001]). The final temperature T_{min} must be low enough to ensure, that the probability that a system moves out in case it is in a global minimum, is negligible. Early in the annealing process when the temperature is high, the system explores a wide range of system configurations. Later, as the temperature is lowered, only states close to the found minimum are tested.

On each step, the temperature must be held constant for an appropriate period of time (i.e. the number of iterations) in order to allow the algorithm to settle in a *thermal equilibirum*, i.e. in a balanced state. If this time is too short, the algorithm is likely to converge to a candidate minimum. The combination of temperature steps and cooling times is known as the *annealing schedule*, which is usually selected empirically. The whole process is commonly called the *cooling schedule*.

If computational resources are of no concern, a high initial temperature T_0, a high local iteration value and a large α are most desirable. Values in the range $0.75 < \alpha < 0.99$ have been found to work well in many real-world problems [Duda et al.2001].

The finite time implementation of SA implies the use of stopping criteria. This could be the total process time allowed for the search, the number of candidate and/or global iterations, a minimum temperature, a number of neighborhood structures used in the search or any condition on the objective value.

6.1.1.7. Acceptance Probability Function

The factors that influence acceptance probability are the degree of objective function value degradation ΔF (smaller degradations induce greater acceptance probabilities) and the temperature parameter T (higher values of T give higher acceptance probability). Simulated annealing is based upon the Metropolis procedure, which consists of a Markov Chain Monte Carlo method that produces a simulation of the system in equilibrium at a given temperature. If, for a minimization problem, the difference in ΔF, between the current state and the new one, is *negative* then the process is continued with the new state. If $\Delta F \geq 0$, then the probability of acceptance of the new state is given by $e^{-\Delta F/T}$. This acceptance rule for new states is referred to as the **Metropolis criterion** [Metropolis et al.1953]. Following this criterion, the system eventually evolves into thermal equilibrium, i.e. after a large number of iterations the probability distribution of the states approaches the Boltzmann distribution [Aarts and Korst1988]. This choice of the probability distribution is generally called Boltzmann annealing. The method of simulated annealing is defined by Kirkpatrick [Kirkpatrick and Gelatt1983] by the following three functional relationships:

- The probability density of the state-space of the solution space defined by the graph G.
- $P(\Delta F)$: Probability for acceptance of a new fitness given the previous value.
- $T(k)$: schedule of *annealing* the temperature T in annealing-time step k.

The acceptance probability is based on the chances of obtaining a new state with fitness F_{k+1} relative to a previous state with fitness F_k,

$$
\begin{aligned}
P(\Delta F) &= \frac{e^{-F_{k+1}/T}}{e^{-F_{k+1}/T} + e^{-F_k/T}} && (6.3)\\
&= \frac{1}{1 + e^{\Delta F/T}} && (6.4)\\
&= e^{-\Delta F/T} && (6.5)
\end{aligned}
$$

where ΔF represents the fitness difference between the present and previous values of the fitness (the values of the *objective function*) appropriate to the physical problem, i.e., $\Delta F = F_{k+1} - F_k$. This is commonly called the *Boltzmann trial*. A Boltzmann trial refers to a competition between two solutions i and j, where element i wins with probability $P = e^{-\Delta F/T}$.

6.2. A Recombinative Population-Oriented SA Algorithm

In this section a recombinative population-oriented SA algorithm (RPOSA) for the clustering of components to reusable packages is presented. As discussed in Section 2.6 simulated annealing as proposed by Kirkpatrick performs Boltzmann trials between a selected solution and a randomly choosen neighbor. The initial solution is repeatedly improved by choosing iteratively the solution with the highest fitness. For the clustering of packages this method is extended to support *populations* of solutions. To allow the generation of new populations the algorithm supports the **recombination** of individual solutions. Recombination is an operation to vary the configuration of a solution from one generation to the next. It is an analogy to the reproduction or biological crossover upon which genetic algorithms are based. The basic mechanism of the algorithm consists of the following steps:

1. Generate the initial population P (randomly, based on an initial clustering or respecting expert decisions).
2. Initialize the parameters $\alpha, x := X_0, y := Y_0, temp := T_0, T_{min}, l, L_{max}$.
3. Perform a *candidate recombination*: Choose a number $x \geq 1$ of randomly selected components from the current solution and hold a Bolzmann trial between the current solution p_i and a new solution p'_i where the selected components are removed.
4. Perform a *neighbor recombination*: Choose a neighbor solution from the population, remove selected components from the neighbor solution and add it to the current solution. Hold a Bolzmann trial between the newly generated solutions.
5. Repeat the steps for each solution in the population (candidate iterations) and decrease the temperature.

The main steps of the algorithm are shown in Diagram 6.2 and are discussed in more detail in the following section. The recombination steps four and five are described in more detail in the Sections 6.2.2 and 6.2.3.

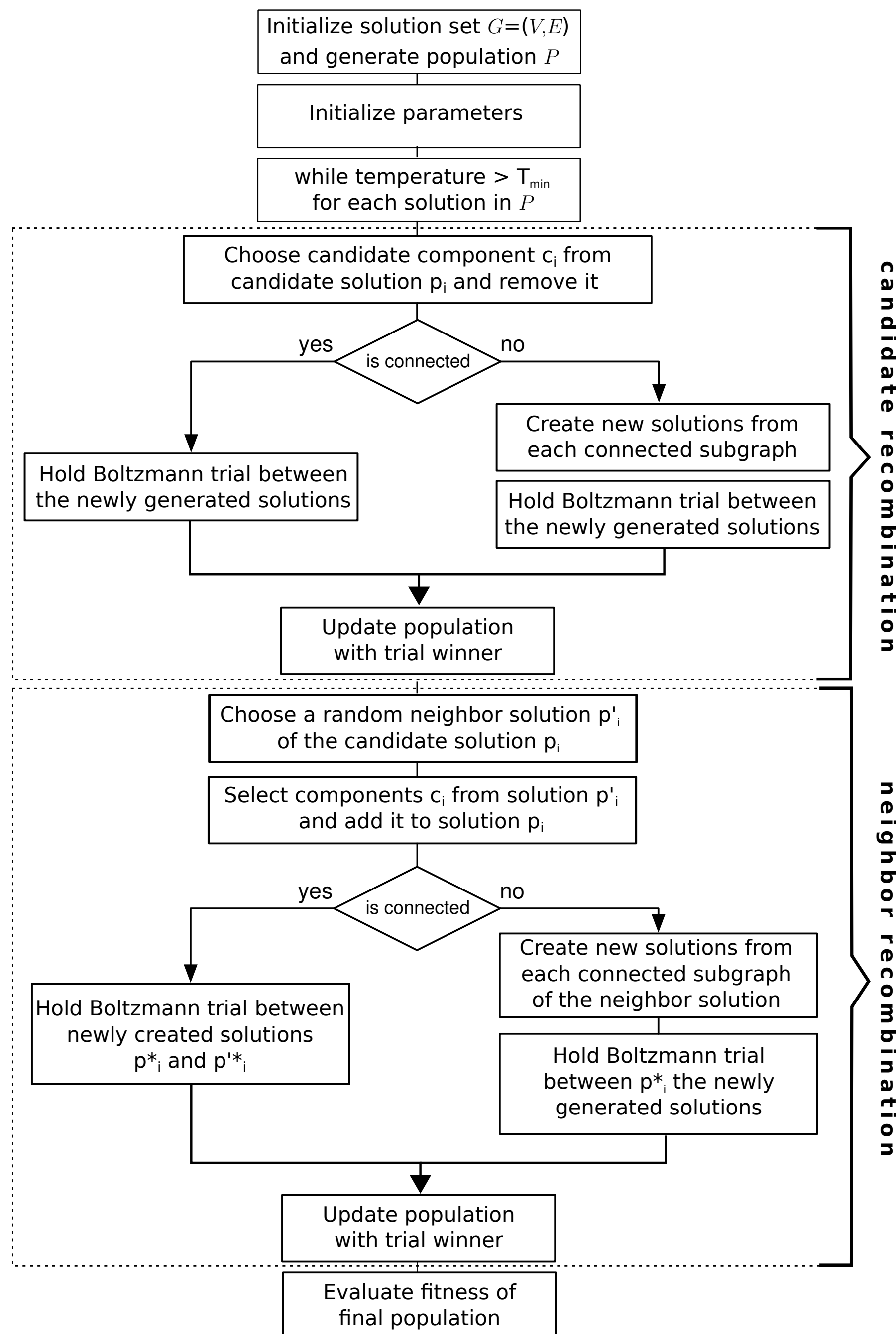


Figure 6.2.: The recombinative simulated annealing algorithm starts with initializing the solution set as a graph $G = (V, E)$, setting the initial parameters for the specific run (depending on the structure of the problem) and choosing randomly one candidate solution from the generated initial population P (which is evaluated according the objective function). On this candidate solution a *candidate recombination* and a *neighbor recombination* is performed a number l times and the next candiate solution is choosen from the current population. After the operations are performed on the current population P_i it is updated with the new candidate solutions and the temperature is decreased according the cooling schedule. The final population is evaluated according the objective function to measure the quality of the new solutions.

6.2.1. The Algorithm (RPOSA)

In Listing 2 the recombinative population-oriented simulated annealing (RPOSA) algorithm is listed in pseudocode. The RPOSA algorithm incoporatess the concept of basic simulated annealing and extends it with mechanisms for population-oriented annealing and on-line adaption mechanisms. In *step one* a graph is constructed (see section 4.1.2) from the set of architectures A and the initial population set P with size $|C| = n$, defining the solutions $p_{c_1}, p_{c_2}, \dots, p_{c_n}$, where $p_{c_i} \cap p_{c_j} = \emptyset$ if $i \neq j$, is generated. For this a vertex-label from the set of classes C is assigned to each vertex $v_1, ..., v_i$ of the weighted labeled graph $G = (V, E)$. The set of classes C represents possible packages the components can be clustered with. Following the basic simulated annealing mechanism the classes are assigned randomly while the number of classes $|C|$ represents the number of solutions (the start packages) of the initial population. Alternatively the initial definition and assignment of classes can be based on a previous clustering (e.g. with use of network analysis methods like community clustering). The initial solutions $p_{c_1}, p_{c_2}, \dots, p_{c_n}$ of the initial population $P_{initial}$ are evaluated according the objective function $F(p)$ of equation 6.1 to assess the quality of the initial solution and measure the difference in value to the final solution.

In *step two* the parameters are initialized. The right choice of the parameter values is crucial for the success of a simulated annealing algorithm [Duda et al.2001, Nolle et al.2001]. There are four parameters which determine the number of iterations: the initial temperature T_0, the final temperature T_{min}, the cooling schedule α and the number of candidate iterations L_{max}. As discussed in section 6.1.1.6, to allow free motion in the search space G, the probability that any uphill move is accepted should be close to 1 which implies a high starting temperature T_0 in the initial phase. The value of T_0 however depends heavily on the topography of G as the search space must not contain *highland basins* with no solution having objective function values close to the global minimum and therefore there is no need for a high initial temperature. If the basin depths are fairly moderate and the phenomenon of highland basins is rare then the cooling steps may be increased without seriously compromising the convergence properties of the algorithm.

The search of the RPOSA algorithm in the graph G (the solution space) is not only controlled by the temperature parameter. By allowing the adaption of the step width sw an aditional control parameter was introduced in Section 6.1.1.4. The step width allows a faster motion through the solution space and can be adapted on-line (and for every recombination operation individually) while the algorithms performs the search. However, as the effect of the control parameters depends highly on the topography of the solutions space G the values of the indiviual parameters must be determined empirically.

The population is evaluated in *step three* where the recombination operations (discussed in detail in section 6.2.2 and 6.2.3) are performed. In the first run the

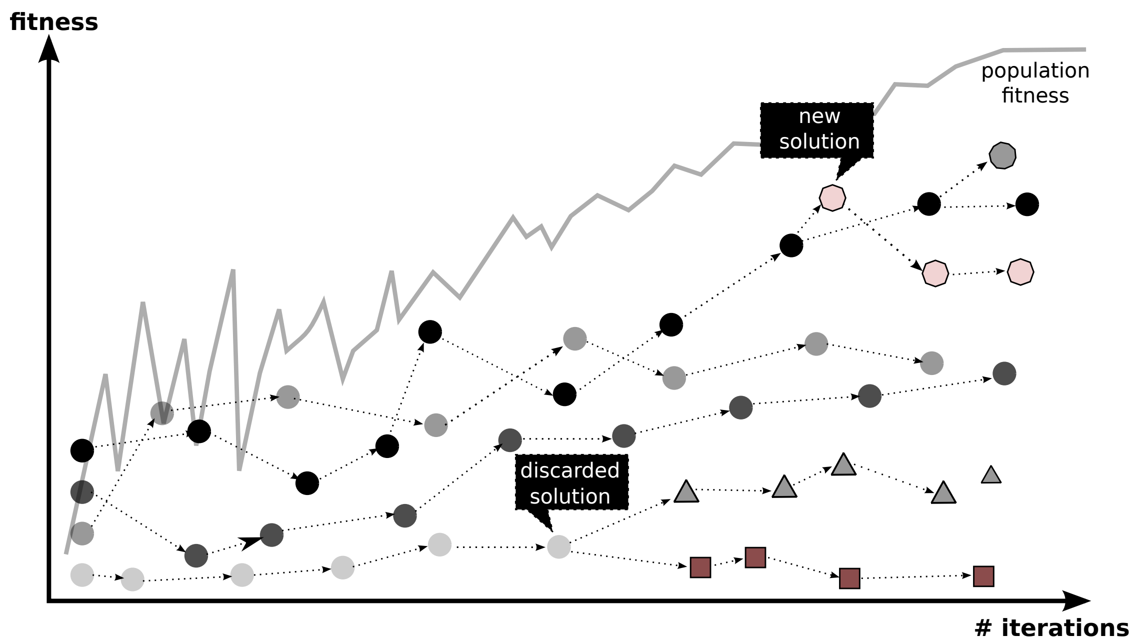


Figure 6.3.: Every candidate solution (each different color represents a different solution) in the population explores the solution space and contributes to the fitness of the packaging. Due to the two recombination methods the generation of new solutions as well as the discarding of solutions is possible. In this example the algorithm starts with a population containing only four candidate solutions. During the run new solutions are generated.

temperature parameter is set to the initial value. Each of the candidate solutions in the population is performing the recombination operations, moving in the search space and contributing to the fitness of the population. The candidate solutions are moving individually a number L_{max} times for each annealing schedule. After the maximum of candidate iterations is reached the population is updated with the changed, new or destroyed solutions and the temperature is reduced according the annealing schedule. This procedure is repeated until a stopping criterion (e.g. a minimum temperature or a maximum number of iterations) is reached. The final population is evaluated with the objective function and can be compared with the inital population.

As discussed, the information in a project repository reflects component-based system architectures. While it is desirable to be able to reuse all components of past projects it is not very likely to acomplish this. Project specific implementations of components provide special functionality or adaptions for a customers environment and are used only once or only for that specific customer. Therefore not all clusters will qualify as a reusable package. After the annealing is finished, packages with relatively low fitness are combined to a cluster of project specific implementations.

6.2.2. Candidate Recombination

To provide means for the generation of new solutions (enlarge or reduce the number of solutions in a population), a candidate recombination operation is performed (see diagram 6.2). After a candidate solution p_i is randomly choosen from the population P a number $x \geq 1$ of candidate components c_i of the current solution p_{orig} is selected. This selection can either be random or based on a fitness value (e.g. a component with the minimum reuse popularity rp of the candidate solution). A new solution is generated from the original solution without the previously selected components (which become a class of their own) as seen in Figure 6.5.

The newly created solution is evaluated if the components of the solution (represented as subgraph) are connected. If not, new solutions are created, one for each connected subgraph. After the solutions are generated a Boltzmann trial is hold between the original candidate solution p_{orig} and the newly created solution p_{new} (if the subgraph was not connected a trial is performed between the original solution and the cumulative objective values of the newly generated solutions). If the newly generated solutions have a higher fitness than the original solution they become the new solution otherwise the original solution is kept with the probability $P = e^{-\Delta F/T}$.

This operation provides means to enlarge or to reduce the number of solutions in the population size and sets individual components free from bounds inside a solution. Those components removed from the original solution have a class of their own and therefore have a fitness which equals zero. If, for example, a previous neighbor

Algorithm 2 RPOSA($G, F, \alpha, T_0, T_{min}, X_0, Y_0, L_{max}$)

Input: $G = (V, E)$: Solution set $v_1, ..., v_n$; $F()$: Objective function; $\alpha()$: Cooling schedule; T_0, T_{min}: Initial/Final temperature; X_0, Y_0: Step width; L_{max}: Number of candidate iterations.
Output: A set of reuse packages close to the optimum.

Step 1: Initialization of the population
Generate the Graph $G = (V, E)$ from the solution set A.
Generate the initial population and evaluate candidate solutions p_i with objective function $F()$

return Evaluated initial candidate solutions $p_1, ..., p_n$

Step 2: Initialize Parameters
Set $\alpha, x := X_0, y := Y_0, T_0, T_{min}, l, L_{max}$

Step 3: Evaluation of the Population
while $T_0 > T_{min}$ **do**
  for all $i \in \{1, ..., n\}$ **do** {For every solution in the population}
    repeat
      $l := 0$ {candidate iterations}

      Step 3.1: candidate recombination (see algorithm 3)
      → Perform a *candidate recombination* on the candidate solution p_i and accept or reject the solution regarding a performed Bolzmann trial.

      Step 3.2: Neighbor recombination (see algorithm 4)
      → Perform, a *neighbor recombination* on the candidate solution p_i and accept or reject the solution regarding a performed Bolzmann trial.

      $l := l + 1$
    until $l = L_{max}$

    Update population with changed candidate solutions.

  end for
  $T_0 := \alpha(T_0)$ {Reduce temperature}
end while
Evaluate final candidate solutions p_i with objective function $F()$
Cluster candidate solutions where fitness equals zero
return Evaluated final candidate solutions $p_1, ..., p_n$

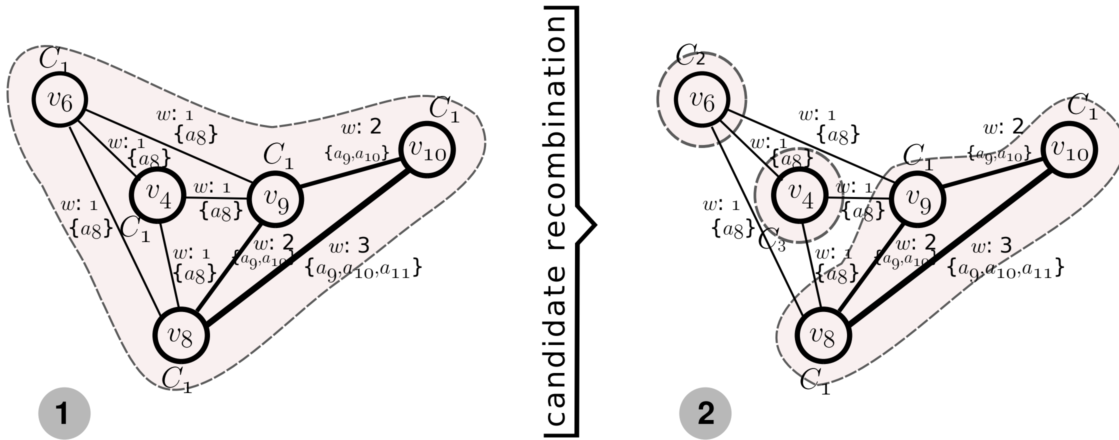


Figure 6.4.: Performing a candidate recombination operation. From the solution in image {1} the candidate components v_6 and v_4 are selected for recombination. Before the recombination the packages in image {1} have a fitness value of 0 as they share no common candidate architecture. In image {2} the new formed solutions are pictured. While the packages with one component each have a fitness of 0 the new formed package with the candidate components v_8, v_9, v_{10} have a fitness of 6 (ignoring the component rank in this example).

recombination was not successfull due to a high force of attraction (with the increase of the fitness value the bound of the components in a solution becomes stronger) of the component to the previous solution, now the component can be added to a new solution or become a new solution of its own. The adding of components (by removing them from a neighbor solution) to a solution is discussed in the following section 6.2.3.

6.2.3. Neighbor Recombination

To explore the solution space G each candidate solution performs a *neighbor recombination*, listed in Algorithm 4, operation after the candidate recombination as discussed in the previous section. The general procedure is pictured in diagram 6.2. For a randomly choosen candidate component c_i, represented by a vertice v_i in the graph G, a neighbor $p' \in N(p_i)$ is choosen. The selection of a neighbor is performed randomly in this approach, but can be easily adapted to select, for example, the neighbor solution with the highest reuse popularity rp or extend the neighbor selection to the whole neighborhood with an additional parameter for the path width. From the selected neighbor solution p'_i a number Y (the step width sw) of random (again, the selection can be adapted to be more selective respecting

Algorithm 3 Candidate Recombination

$\rightarrow$ Pick a number x of random adjacent elements v as a set V_i from p_i as set p'_i
if $|p_i| > 1$ and connected == True **then**
  Let $\Delta F = F(p'_i) - F(p_i)$
  if $\Delta F < 0$ **then** {uphill move}
    $\rightarrow$ Remove set V_i from p_i with probability $P = e^{-\Delta F/t}$
    if P **then** {uphill move}
      $\rightarrow$ Assign new class C_{n+1} to each element v of set V_i
      $\rightarrow$ Add new class to population P
    end if
  else if $\Delta F > 0$ **then** {downhill move}
    $\rightarrow$ Remove set V_i from p_i
    $\rightarrow$ Assign new class C_{n+1} to each element v of set V_i
    $\rightarrow$ Add new class to population P
  end if
else if $|p_i| > 1$ and connected == False **then**
  Assign a new class to each new connected candidate solution p'_{i_n}
  Let $\Delta F = F(\sum^n p'_{i_n}) - F(p_i)$
  if $\Delta F < 0$ **then** {uphill move}
    $\rightarrow$ Remove set V_i from p_i and generate net candidate solutions with probability $e^{-\Delta F/t}$
    if P **then** {uphill move}
      $\rightarrow$ Assign new class C_{n+1} to each element v of set V_i
      $\rightarrow$ Add new class to population P
    end if
  else if $\Delta F > 0$ **then** {downhill move}
    $\rightarrow$ Remove set V_i from p_i
    $\rightarrow$ Assign new class C_{n+1} to each element v of set V_i
    $\rightarrow$ Add new class to population P
  end if
end if

additional control parameters) adjacent components is choosen, removed from the neighbor solution and added to the candidate solution p_i.

Before the fitness difference $\Delta F = F(p_i'^* - p_i') + F(p_i^* - p_i)$ is calculated it is verified if the neighbor solution is still connected. If the neighbor solution is connected a Boltzmann trial is hold and with $\Delta F > 0$ the components selected from the neighbor solution p_i' are added to the candidate solution p_i. With $\Delta F < 0$ they are added with probability $P = e^{-\Delta F/t}$. If the neighbor solution is not connected it is split and each connected subgraph of the neighbor solution becomes a new solution in the population. The acceptance of this operation is also decided via a Boltzmann trial with $\Delta F = F(\sum^n p_{i_n}') - F(p_i)$.

Algorithm 4 Neighbor Recombination

→ Choose a neighbor $p' \in N(p_i)$
→ Remove a number y of random adjacent components c as a set V_i from a selected random neighbor p_i' as $p_i'^*$ and add it to p_i as p_i^*
if $|p_i'| > 1$ and connected == True **then**
  → Let $\Delta F = F(p_i'^* - p_i') + F(p_i^* - p_i)$
  if $\Delta F < 0$ **then** {uphill move}
    → $p_i = p_i^*$ with probability $P = e^{-\Delta F/t}$
    if P **then** {uphill move}
      → Add new class to population P
    end if
  else if $\Delta F > 0$ **then** {downhill move}
    → $p_i = p_i^*$
  end if
else if $|p_i'| > 1$ and connected == False **then**
  → Assign a new class to each new connected candidate solution p_{i_n}'
  → Let $\Delta F = F(\sum^n p_{i_n}') - F(p_i)$
  if $\Delta F < 0$ **then** {uphill move}
    → $p_i = p_i^*$ with probability $P = e^{-\Delta F/t}$
    if P **then** {uphill move}
      → Add new class to population P
    end if
  else if $\Delta F > 0$ **then** {downhill move}
    → $p_i = p_i^*$
    → Add new class to population P
  end if
end if

6.3. Application of RPOSA

In this section the previously discussed RPOSA algorithm is applied to the example repository (see annex A). In order to determine the optimal variation among the control parameters for the particular network topography of the example repository, a number of experiments where carried out. In table 6.3 the variations of the tested

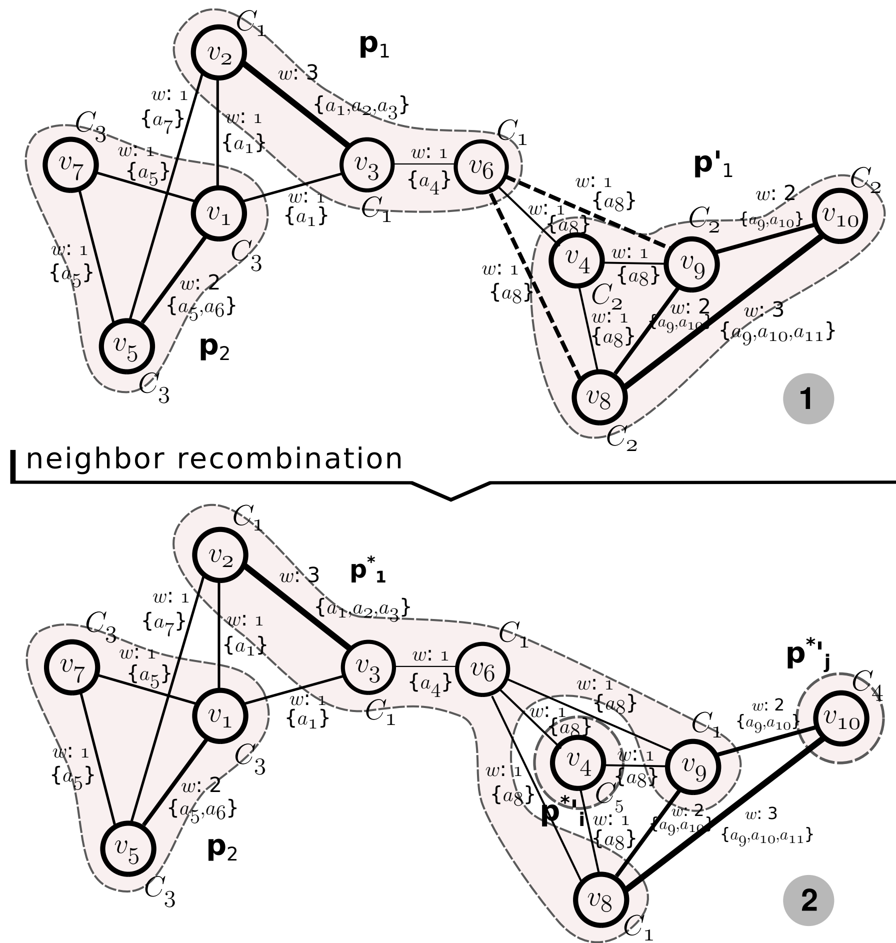


Figure 6.5.: Performing a neighbor recombination operation. From the solution p_1 in image {1} the candidate component v_6 selects the components v_8 and v_9 from the neighbor solution p_1' for a recombination operation. After the operation in image {2}, the new neighbor solution $p*'$ would not be connected and the new solutions $p*_i'$ and $p*_j'$ are created. The fitness of solution p_2 has an unchanged value of 3 (ignoring component ranks in this example) while the fitness values of the new solutions also remain 0 after the recombination

values for the different control parameters as well as the final values for the example repository are listed. The final configuration of the control parameters resulted in 1053 iterations and a total of 0.74 seconds running time (on a Intel Core 2 Duo Laptop with 2.4 Ghz and 8 GB RAM). The running time is composed of <1% for the generation of the graph (n=50), ∼10% for the calculation of the component rank and ∼90% for execution of the RPOSA algorithm.

Parameter	Tested Values	Used Value
Start temperature T_0	30,25,20,15,10,8,6,5,4,3,2,1,0.1,0.01	10
Final temperature T_{min}	0.1,0.01,0.001,0.0001,0.00001	0.001
Cooling schedule α	0.99,0.98,0.97,0.95,0.90,0.85,0.80	$0.85 \forall T > 1; 0.98$
Local Iterations L_{max}	20,15,10,5,3,2,1	3
Step width sw	$N(v)$, $\frac{\|N(v)\|}{2}$, 1	$\frac{\|N(v)\|}{2}$

Table 6.1.: Tested values for the control parameters and the final decisions for the example repository.

Figure 6.6 shows the result of the RPOSA algorithm. The dotted line represents the best population found in over 150 runs. While the fitness of the population in that particular run tends to be the maximum, a significant trade-off in running time (3238 iterations) was observed. Laarhoven et. al. [Laarhoven et al.1987] argue, that the performance of a SA algorithm has to concentrate on the following two quantities:

- the **quality** of the final solution obtained by the algorithm, i.e. the difference in cost value between the final solution and a globally minimal solution;
- the **running time** of the algorithm.

For the simulated annealing algorithm, these quantities depend on the problem instance as well as the cooling schedule. Two approaches for investigating the performance of simulated annealing are possible: a *theoretical analysis*, which tries to find a analytical expression for the quality of the solution and for the running time, and an *empirical analysis* which solves many instances with different parameters and draws conclusions from the result with respect to both, quality and running time. For the theoretical analysis the literature provides only worst-case analysis results [Lundy and Mees1986, Aarts and Van Laarhoven1985]. No theoretical average-case analysis results are known for the running time and quality of a solution [Laarhoven et al.1987].

As for the final configuration of the RPOSA algorithm, applied to the current problem, a trade-off in accuracy was accepted in favor of the running time.

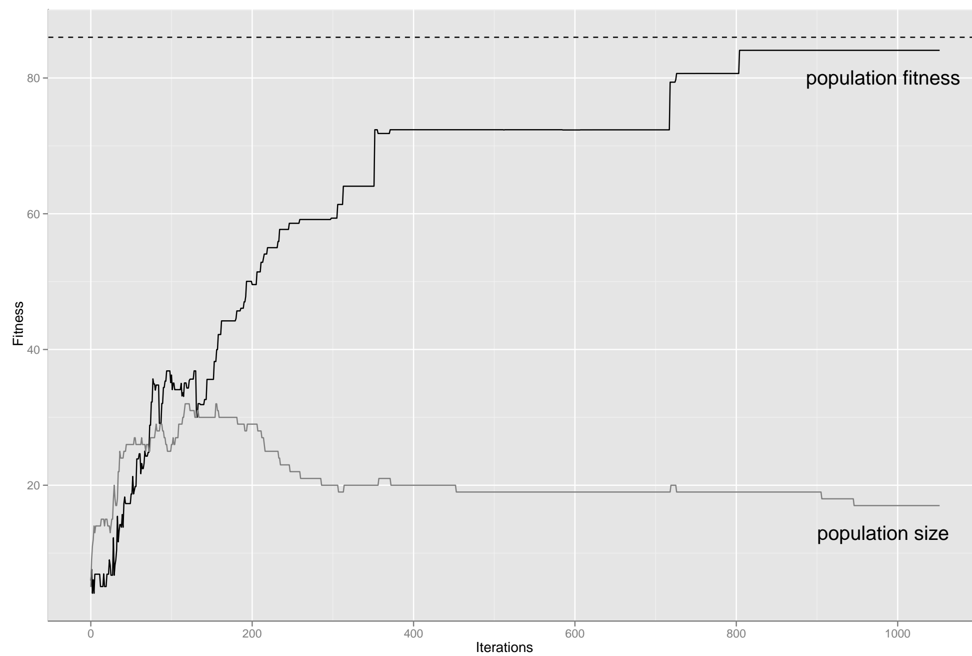


Figure 6.6.: RPOSA algorithm applied to the example repository. The dotted line represents the best solution found in over 150 runs while the black line pictures the fitness of the population over the iterations. The grey line shows the development of the number of clusters in the population.

It can be observed in Figure 6.6 that the algorithm initially explores the search space (large differences in fitness), while later on (after approximately 180 iterations in this case) it exploits the most promising region. Hence, the behavior of the algorithm at the beginning of the search is similar to a random walk, while towards the end it performs like ordinary hill climbing. The grey line in the figure shows the development of the population size. The randomly generated (uniform distribution) initial population had a total fitness of 0 and 5 solutions. In the exploration phase it raises to over 30 solutions and shrinks toward the end to a final number of 17 solutions in the population (shown in table 6.3) which has a final fitness of 84.076.

Components	Count	Fitness
c_{13}, c_{12}	2	1.219
c_7, c_6, c_9	3	1.33
c_8, c_{26}	2	1.471
c_{43}, c_{41}	2	1.509
c_{42}, c_{45}	2	1.509
c_{46}, c_{44}	2	1.509
c_{24}, c_{25}, c_{23}	3	1.661
c_{39}, c_{38}, c_{40}	3	1.781
c_{49}, c_{50}, c_{47}, c_{48}	4	2.436
c_{11}, c_{10}	2	5.091
c_{31}, c_{29}	2	5.654
c_{15}, c_{14}	2	6.094
c_{36}, c_{37}	2	6.823
c_{35}, c_{34}, c_{33},c_{32}, c_{30}, c_{28}, c_{27}	7	7.485
c_2, c_{20}, c_{21}, c_{22}, c_{17}, c_{16},c_{19},c_{18}	8	7.521
c_5, c_4	2	11.899
c_1, c_3	2	19.084
Sum of the 17 solutions in the population:	**50**	**84.076**

Table 6.2.: Final clustering to reusable packages of the components from the example repository (sorted in ascending order).

Not all packages of the final clustering would qualify as a reusable package. While the cluster with the candidate components c_1 and c_3 shows a very promising fitness

value, the clusters with 8 and 7 components have a relatively low fitness compared to the promising package. Those two packages, and respective their components, are combined to form a cluster of project specific implementations which are not considered for reuse.

Once meaningfull packages of components are clustered the previously (see section 5.1) discussed network analysis methods can be reapplied. With centrality measures important components (e.g. gatekeepers or central components) of individual reuse packages can be identified and will provide important knowledge for a system architect and for the integration engineers in a project.

6.4. Summary: Reuse-Oriented Component Packaging

In this chapter an optimization approach, which provides means to select and group components to reusable packages, was introduced.

To optimize the reuse of component packages, the clustering of components to individual reusable packages is improved, by measuring the impact of individual components to these clusters. This is achieved by incorporating the proposed reuse graph metrics (see Section 5.4) in a novel simulated annealing algorithm that uses the component-graph generated in Chapter 4 and was extended to support the optimization of a number of different solutions rathern than one particular.

Kunz argues in [Kunz2010] that in practice software measurement tools find small acceptance due to their high costs, inflexible structures, missing integration capabilities and the resulting unclear cost/benefit ratio. To overcome this obstacles and support the architects with a shedule how and when to apply the various measurement techniques, a reuse oriented process will be presented in the next chapter. In this proposed process the evaluation and clustering techniques presented in the previous sections are integrated in a company's development process which is supported by a model-based architecture framework to support the handling (e.g. storing and retrival) of individual components.

Part III.

Evaluation Process and Framework

7. Model-Based Architecture Evaluation Framework

In this chapter a framework is presented, that intends to support software and system architects as well as integration engineers at the process of developing new solutions from existing components. The process, wrapped around the framework, means to integrate modeling and evaluation of component-based automotive systems in a model-driven engineering (MDE) development process based on an architecture description language (ADL). First the importance for a model-based approach in the development of component-based embedded systems is motivated and the current process is opposed to the new process. This is followed by a general overview of the new process, a description of the enwrapped framework and a more detailed discussion of the individual parts of the framework. To be prepared for next generation automotive systems, SAE AADL has been extended to support the specification and modeling of multicore architectures.

7.1. Motivation

A basic idea in component-based software engineering is to build systems from existing components. This reusability is the ability of reusing the same component in different projects or even applications. But this desired property includes the assumption, that the architectural framework of the systems must be the same. Such an architectural framework is provided by many model-based development approaches, as they use the concept of formal description and specification of systems. Furthermore model-based approaches provide means to support communication (e.g. as interchange format between development groups or even customers), verification and provide models that can be analyzed for multiple qualities.

Torngren et. al. [Torngren et al.2005] argue that the two approaches, component-based system development and model-based methods, complement each-other and are both required for the development of complex systems. This motivates the process and framework, proposed in the following sections, which incorporates the concepts for reuseability described in the Chapters 5 and 6 and supports the various advantages [Selic2003, Puerta1997, Schaetz et al.2002] of model-based development like automated code generation or formal verification.

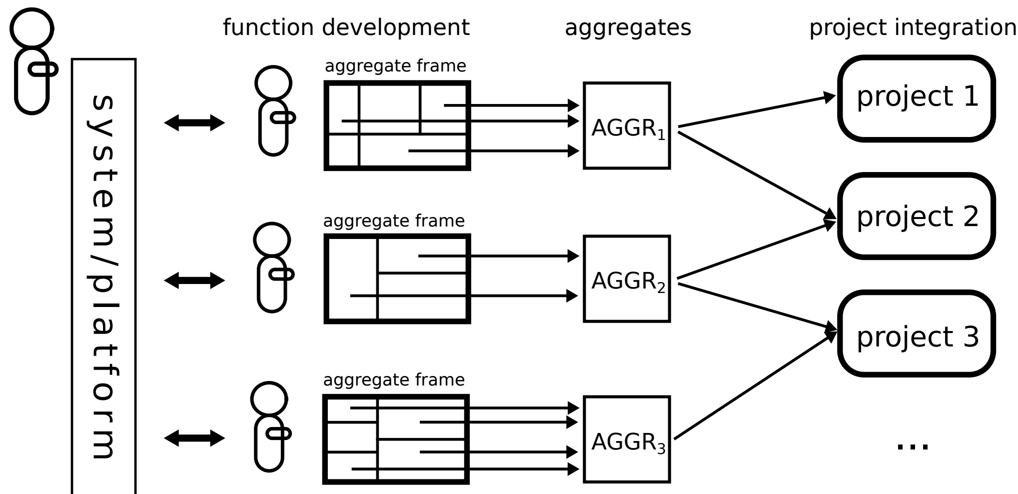


Figure 7.1.: The current reuse process based on functional partitioning. The system architect defines the functionality and the authority (the function representative) for each function implements and supports it. Parts of the function are composed together to an aggregate which is used in system projects.

7.2. Functional Partitioning to Reuse Partitioning

The current component partitioning and reuse concept, discussed in 2.4, is based solely on a functional partitioning of the product lines. In Figure 7.1 the concept is shown. The development process is triggered from a system/platform architect who specifies the new architecture and decides about new functionality for the system. The architect interfaces with the function representatives who manage the versions and revisions of each function. From different versions and revisions of the functions reuse packages are composed, which are used in different projects (e.g. diesel/gasoline or customer specific).

The implications of this approach, discussed in Section 2.4, motivate the need for a reuse-based process. The few but large aggregates, which serve as reusable packages, are not very flexible and make the reuse process very complicated and hard to overlook. Figure 7.2 shows a new development process based on reuse decisions, which would be mandatory to make use of the component package generation proposed in this thesis.

The functionality is, like in the current development process, defined by a system architect. The authority for the implementation and support of the functionality is also still with the function representative. New to the proposed concept is the replacement of the big (in terms of aggregated components) Aggregates, which would be replaced by more and smaller reuse packages, which then are used in

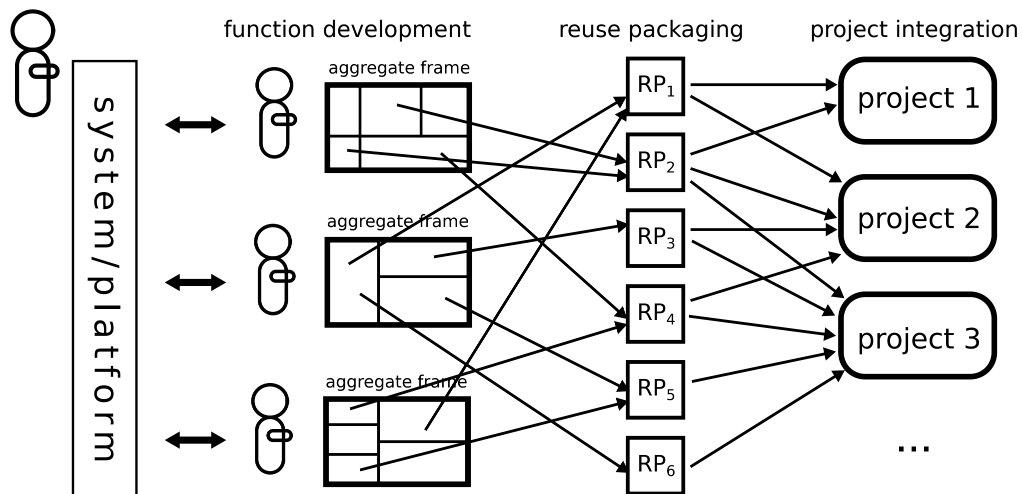


Figure 7.2.: The proposed reuse process based on reuse partitioning. Like in the current process the system architect defines the functionality and the authority (the function representative) for each function implements and supports it. Parts of each function are composed together in reusable packages which are used in system projects.

different system projects and product lines. Such reuse packages are automatically generated with the approach presented in this thesis.

7.3. Reuse-Oriented Development Process

In this section the various engineering steps of the reuse-oriented development process, shown in Figure 7.3, are discussed. The process consists of the following steps:

- Specification and modeling of the high level system architecture in an architecture description language.
- Enrichment of the system architecture with components and component packages from a component repository.
- Refinement of the architecture model with project or customer specific components.
- Instrumentation of the system with non-functional requirements.
- Generation of the architecture implementation model.
- Generation of software artefacts.

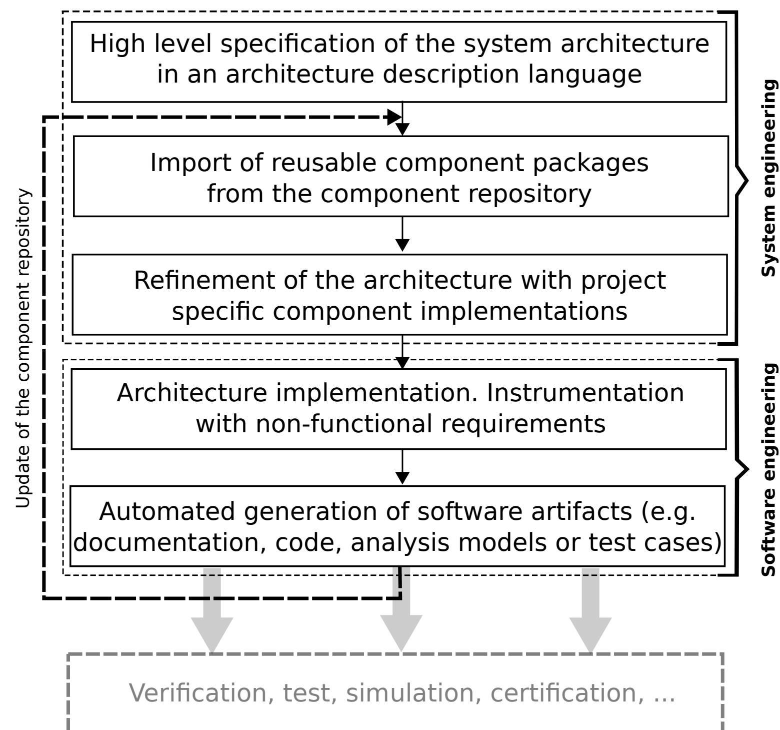


Figure 7.3.: The proposed process the framework is wrappend in. After step five (a component defines not only code but also documentation and specification) the component repository is updated with the new components.

Prior to those steps the component repository has to be constructed using the data collection methodology discussed in Section 4.1. After the component reuse network has been constructed, components are evaluated with network analysis methods to, for example, rank them and as such provide useful information for an system expert. Followed by the network analysis the components are clusterd with use of the RPOSA algorithm discussed in Section 6.1 to provide initial reuse packages which then can be used following the steps listed above. In sum, the steps needed prior to the development process are:

- Construction of the component reuse network(s)
- Evaluation and measurement of the components with network analysis methods
- Application of the RPOSA algorithm to create reuse packages

7.3.1. Model-Based Reuse Framework

A detailed description of the individual parts of the framework is shown in Figure 7.4. At first the high level system architecture is specified and modeled in an architecture description language. This architecture description language provides the common framework for all future systems in the product lifecycle. The architect is enabled to choose components and component packages from a component repository which suggest components, based on evaluations described in the Chapters 5 and 6. Choosen components are then added to the system architecture model. If this step is completed, the architecture model is refined with project or customer specific components. In the next step the system is instrumented with non-functional requirements and implemented. From this architecture implementation model several software artefacts are generated automatically including documentation, code, models for further analysis (e.g. a timing model for real-time analysis) as well as test cases.

7.3.2. High Level System Architecture Specification

As introduced in the previous section the development of a new solution starts with the specification of the architecture using models with well defined syntax and semantics for designing the functionality as for example SAE AADL [Aerospace2004] which was introduced in section 2.8. It offers the advantage of an international standard and allows the precise definition of the semantics of a predefined set of components. During this system engineering phase architectural decisions are made and high-level components are specified. The SAE AADL offers a solid foundation for the modeling approach as it is specialized for vehicular embedded systems

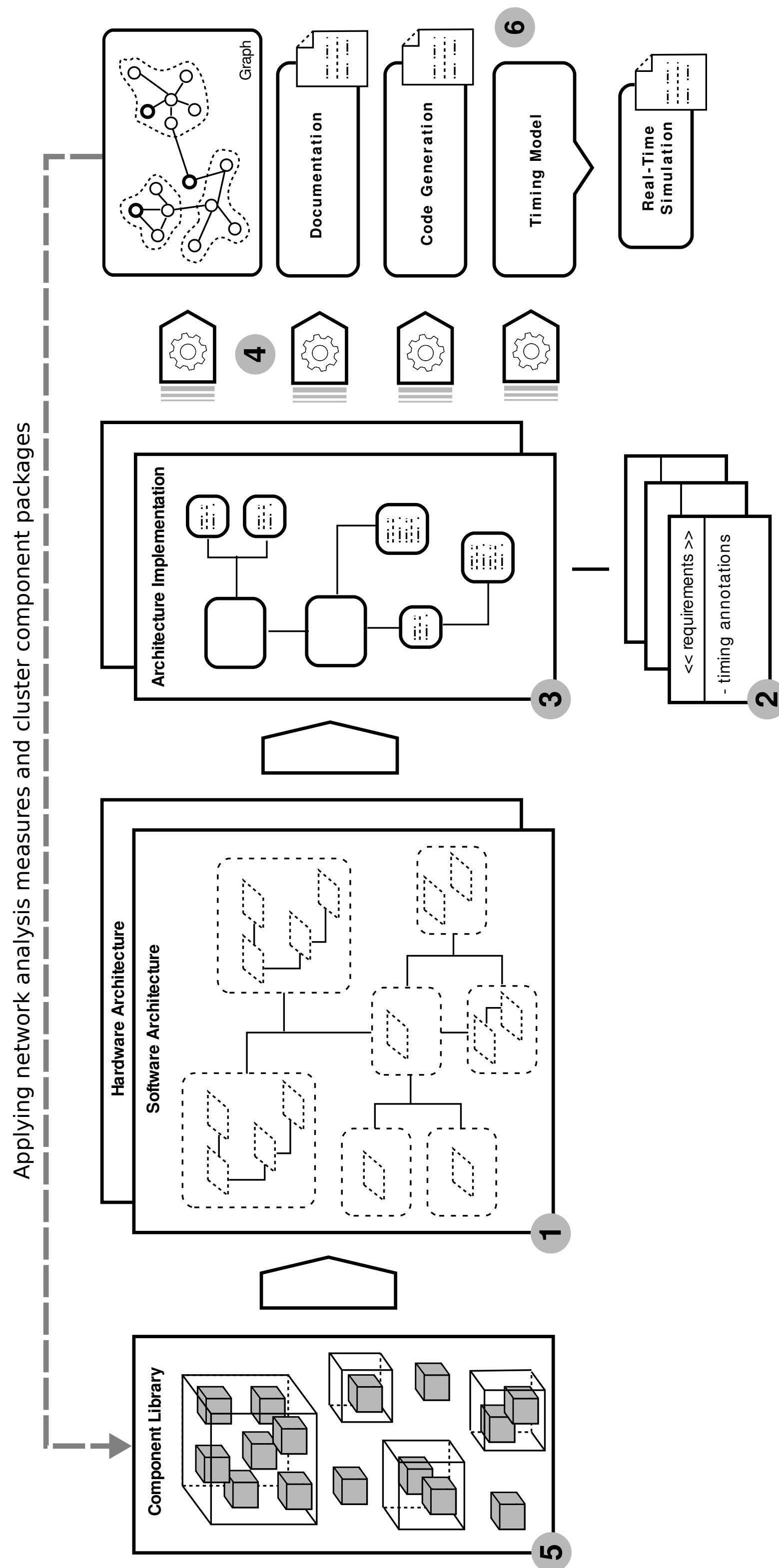


Figure 7.4.: Proposed architecture framework with an experience-based component repository (5), the software architecture (1) specified by an architecture description language and the architecture implementation (3) which is instrumented with various non-functional requirements (2) for further analysis (e.g. real-time simulation) and contains the functional parts for generation of code and documentation (4).

development. It also provides constructs and properties required for analysis of safety, reliability and timing, as well as explicit models of software components.

With SAE AADL the component development process can be separated from the system development process. This allows significantly shorter time-to market as products are integrated from already existing components. While formal abstractions are important for analysis and verification, visual representations of models are often important for comprehension of complex systems and the communication of designs. The SAE AADL supports both, a textual as well as a graphical notation and provides an open-source modeling environment [Feiler et al.2006a] which can be integrated in an existing development process.

7.3.3. Reusable Component Package Import

After the initial high level system architecture is specified and modeled the system architect uses a component library to search for reusable components and component packages. This component library is automatically generated and updated with the appraoch described in Chapter 6. Components and component packages are generated and stored with measurement information attached to each individual component or package. Implemented with sorting and search possibilities it serves as an expert system, supporting the system architect at the process of deciding which components or packages to reuse. For the use of SAE AADL as modeling language the tool STOOD supports libraries of SAE AADL models [Dissaux2005].

7.3.4. Architecture Refinement

After the high-level system was modeled, the initial components will then be refined by the addition of a hierarchy of subcomponents as well as lower level properties. They also incorporate more detail and peculiarities of the implementation, fulfilling the software requirements while preserving the global architecture of the particular system (see Figure 7.4 (2,3)).

7.3.5. Architecture Implementation and Instrumentation

Dissaux described in [Dissaux2005] that the normal continuation of this activity consists in performing the software detailed design and coding steps that will lead to the automatic generation of ready to compile target source code and its synchronized design documentation (see Figure 7.4 (4)). The capability to generate code from the architecture model has become a very important factor in the development of engine management systems at Continental Automotive Corporation.

Several approaches to integrate the capability into SAE AADL exist [Brun et al.2008, Dissaux2005] (e.g. the Ocarina tool suite [Lasnier et al.]) and are free to use. The functionality of the components is implemented in C code. The SAE AADL architecture model provides, with the concept of subprograms, a methodology to instrument the model with implemented functionality. As the SAE AADL has been chosen to describe these reusable components, a SAE AADL output transformation is used to generate source code in a similar way as the target language source code.

The implementation model of the architecture is now instrumented with non-functional requirements e.g. safety, dependability or performance properties. As an example and to support the modeling and simulative evaluation of emerging multicore systems in the embedded domain, the SAE AADL was extended with real-time properties for multicore systems in section 7.4.

7.3.6. Generation of Software Artefacts

An important side effect of the use of a modeling language for the design of the system and automatic generation of code is the possibility to automatically *document data dependencies and information flows.* This enables the data collection process, described in Section 4.1, to generate a more detailed and less dense component reuse network. Therefore means for more specific and more accurate evaluations as well as a robuster way to group components to reusable packages as described in section 6.1 are provided.

This architecture models can also be used for early design validation and verification as well as for testing and integration. Other usages of the models include information and process modeling. Developed in the right way, models become assets that can themselves be reused within and between projects and customers. The model-based methodology provides possibilities to develop and use a multitude of models for different system aspects (i.e. product properties and constraints) such as failure modes of components and their propagation, required timing behavior of a real-time implementation, power consumption, and the behavior of the expected environment.

7.4. Multicore Support

With the introduction of multicore systems to the embedded domain new requirements for the modeling and description of embedded systems emerge. This requirements include:

- Modeling of multicore hardware components including heterogeneous cores with variable processing speed

- Comprehensive execution time models, e.g. expressed by probabilistic distributions
- Support for the annotation of scheduling policies

To integrate the new framework for the specification and evaluation of component-based automotive systems architectures, the support for this new requirements has to be provided by the ADL.

While SAE AADL provides the means for modeling the hardware platform (e.g. processor or memory) it does not provide the possibility for the modeling or mapping of software components, tasks or ISRs, on a number of cores of a multicore processor. To support multicore systems, the properties of the standard execution platform component processor were extended to support the modeling of heterogeneous multicores [Deubzer et al.2010].

In Listing 7.1, an excerpt of the definition of the new properties is shown. In addition to the *Cores* property that defines a core of a processor, the possibility of adding a quartz to each core (*Quartz*), as well as defining the quartz frequency and the core instructions per quartz tick are defined. The mapping allows to assign quartzes to specific cores. Additionally, it is possible to configure scheduling policies.

Listing 7.1: Selected multicore extension of SAE AADL

```
property set multicoreEXT is

  Quartz: aadlinteger applies to (device);
  Cores: aadlinteger applies to (processor);
  Core_Instructions_per_Tick: aadlinteger  applies to (processor);
  Quartz_Frequence_Hz: aadlinteger applies to (device);

  -- Scheduling
  Scheduling_Protocol: type enumeration (osek, edf, global_edf, pf_pd2, pf_er_pd2,
       partly_pf_pd2, p_er_pd2, edzl, llref);

  -- Mappings
  -- Mapping Quartz to Core
  Quartz_Core_Binding: inherit reference(processor) applies to (device);
  .
  .
  .

end multicoreEXT;
```

With these additional properties and the standard scheduling annotations of the SAE AADL, the modeling of a multicore system is possible. In Listing 7.2 an example modeling of the hardware is shown. The processor *multicore.dualcore* has two cores which each have a mapping to the same quartz. The cores (*Cores.core0/1*) can be annotated with specific properties as defined in Listing 7.1.

Listing 7.2: Modeling of a dualcore processor in SAE AADL

```
-- multicore processor
 system implementation multicore.dualcore
   subcomponents
     core0: processor Cores.core0;
     core1: processor Cores.core1;
     quartz0: device Quartz.quartz0;
   properties
     -- Mapping core to quartz
     multicoreHW::Sim_Processor => 1;
     multicoreHW::Quartz_Core_Binding => reference core0 applies to quartz0;
     multicoreHW::Quartz_Core_Binding => reference core1 applies to quartz0;
 end multicore.dualcore;

-- processor-core
  processor implementation Cores.core0
    properties
      multicoreHW::Core => 1;
      multicoreHW::Core_Instructions_per_Tick => 10;
  end Cores.core0;

-- processor-core
  processor implementation Cores.core1
    properties
      multicoreHW::Core => 1;
      multicoreHW::Core_Instructions_per_Tick => 20;
  end Cores.core1;

-- quartz
  device implementation Quartz.quartz0
    properties
      multicoreHW::Quartz => 1;
      multicoreHW::Quartz_Frequence_Hz => 100000000;
  end Quartz.quartz0;
  .
  .
  .
```

After the system is modeled in SAE AADL and annotated with the multicore properties, it can be used as an input for the scheduling simulation. In addition to the scheduling analysis, this specified single SAE AADL model can be analyzed for multiple qualities e.g. availability and reliability, security or resource consumption.

7.5. Summary: Reuse-Oriented Development Process

The application of measurement tools in practical environments is often confronted with a number of different drawbacks, discussed by Lother in [Lother2007] and Kunz in [Kunz2010]. Most of such drawbacks result from a missing integration of the measurement tools in the development process. With the approach presented in this chapter, means to integrate the measurement into the process and support the development with a framework to ensure a continuous application, are provided. With the use of an architecture description language for the design of the architecture additional benefits emerge. In Section 3.1 implications resulting from the

component selection problem, especially the retrieve of components, is discussed. A tight integration of the modeling approach with a component repository, as proposed in this chapter, will support the system architect with the difficult task of finding and retrieving the right component for the current system under design.

Part IV.

Case Study: Engine Management System

8. Evaluation of the Engine Management System

The engine management system (EMS2), introduced in Section 2.4 is evaluated by using the discussed network analysis method of Chapter 5 and reusable packages are generated with the proposed RPOSA algorithm proposed in Section 6.2. In the following Section 8.1 the actual impact regarding reuse of the system is evaluated. The data for the generation of the architecture model is extracted in Section 8.2. In Section 8.3 the generated graph is evaluated with use of network analysis measures followed by Section 8.4, where reusable packages are generated with the use of the proposed simulated annealing approach.

8.1. EMS Reuse: Status Quo

In this section the actual impact and efficiency regarding the reuse of components of the component-based EMS2 architecture, discussed in Section 2.4, is evaluated. For this purpose, the baselines of successfully completed projects since the introduction of the new architectural concept were analyzed with database data mining techniques.

In Figure 8.1 it is shown how often modules, the "atoms" of the reuseable aggregates (see Figure 2.7(a)), were reused over different projects. The reuse level analysis reveals that around 54 percent of the modules were used in twenty or less projects, while 26 percent of the modules were used only once.

Consequently, 74 percent of the modules where reused at least in two projects while 12 percent of the modules were even used in one hundred or more projects. The overall use of modules over all projects is shown in Figure 8.2. While a big portion of around 87 percent where used less than 100 times, almost 700 modules where used more than that. While this seems to be a reasonable reuse level for the atomic parts, it does not give any information on the reuse efficiency of the aggregate grouping (see Figure 2.7(b)) as discussed in Section 2.4. To evaluate the current partitioning of modules to aggregates and apply network analysis methods on the components the component reuse network has to be constructed.

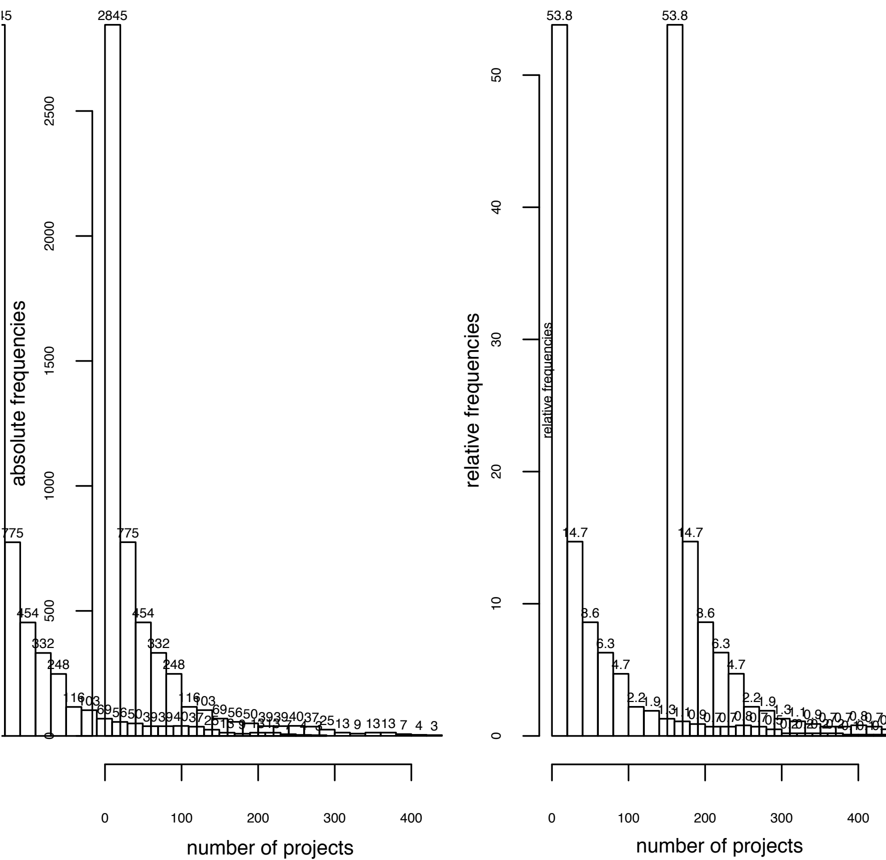


Figure 8.1.: This figure shows the reuse of modules over different projects.

8.2. Data Collection And Extraction

The information in the companies project repository includes an accurate and detailed picture of the organizational structure of the software architecture and its components.

By collecting data from the repository knowledge about components and component groups was automatically captured. This knowledge is gained from projects being already in the maintenance phase as well as projects being under active development at Continental Automotive Engine Systems. The data was collected by following the approach presented in Section 4.1.

For context specific analysis, different graphs where generated. For example graphs with the focus only on closed projects in a specific customer cluster to analyse reuse on a per customer basis were generated. To analyse the reuse level of specific aggregates graphs can even be limited to a special functionality. The resulting final weighted graph (over all projects, aggregates and customers) on the use of components in past projects, has around $50 \cdot 10^3$ nodes (components) and $26 \cdot 10^6$ edges (the number of connections between the components). To reduce the size and

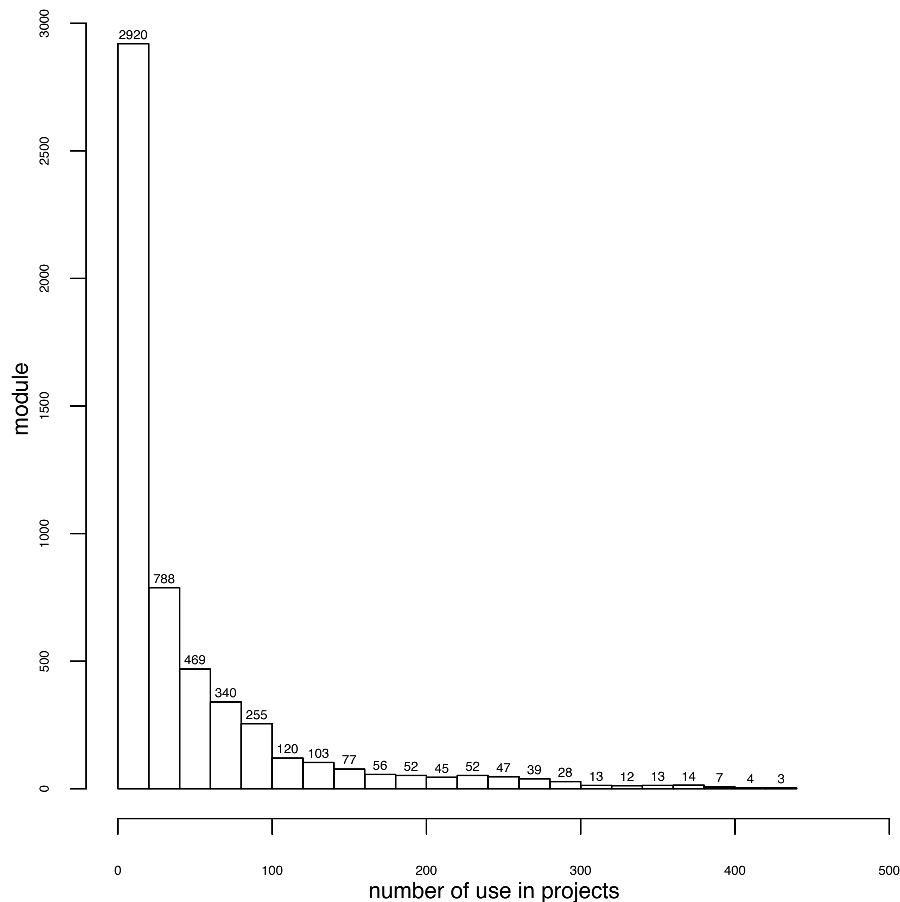


Figure 8.2.: This figure shows the reuse of modules over all projects. One bin represents 20 projects.

complexity of the network Prim's algorithm [Morris et al.2008] was used to find a maximum weighted spanning tree of the graph. This algorithm finds a subset of edges forming a tree which includes every component. The total weight of all the edges is maximized (see Figure 8.3). While some information on the degree of the graph (the connections each component has) is lost, the important information about the maximum reuse of components and component groups are preserved, as well as the strong relationships between components, which are a crucial metric for our clustering approach. For the application of the RPOSA algorithm from section 6.2 the full graph without the reduction to the spanning tree was used.

8.3. Application of Network Analysis Measures

Once the network based on the previous definitions is constructed, standard network analysis concepts can be applied.

In Figure 8.4 the result of the centrality analysis is shown. The eigenvector centrality, defined in Section 5.1, is plotted versus the betweenness centrality. The more a component is connected to other highly connected components the higher is the weight the eigenvector centrality gives to this component. A component connected to, for example, ten high-scoring components will have a higher eigenvector centrality than a component connected to ten low-scoring components. Thus, it can be interpreted as a measure for the importance of a component in a network and marks components with high reusability e.g. high quality due to a high test coverage. The betweenness centrality measures the number of shortest paths going through a specific component. Thus a high betweenness centrality marks the importance of a component as a central hub with a potentially high amount of information flowing through that component. A high betweenness centrality marks components which are used in different projects. The relatively low eigenvector centrality of those components suggest that components that are used in different projects are often changed and not much reused. Nevertheless this analysis helped to identify a number of key components which will play a central role in the repository.

The distribution of the degree in the network is shown in Figure 8.5. Few components of the network have a very high degree and are subject to further analysis. The degree distribution shows also that the network is a so called scale-free network, which is defined by sequential addition of new nodes and preferential attachment. One characeristic of a scale-free network is the relative commonness of components with a degree that greatly exceeds the average. The highest-degree nodes are often called hubs. For the component network this shows that high-degree nodes are placed in the middle of the network and progressively lower-degree components are added sequential (a project is started with legacy components and new components are added around them).

To find groups of components, the concept of Section 5.2 was used. As discussed in Section 2.4, the grouping of components to component groups (respectively

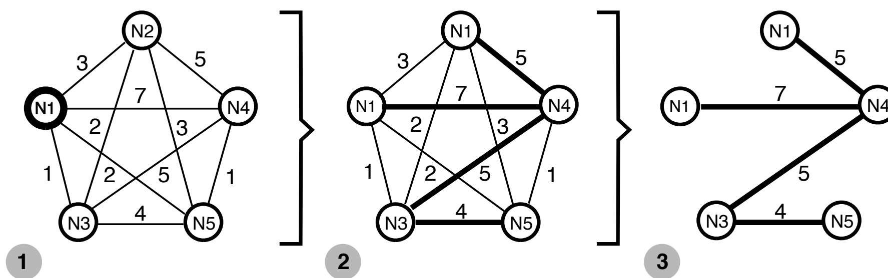


Figure 8.3.: Generation of the maximum weighted spanning tree. One component of the graph (1) is chosen as starting point. The components with the maximum weight are chosen (2). The final graph (3) contains all components with a reduced set of edges.

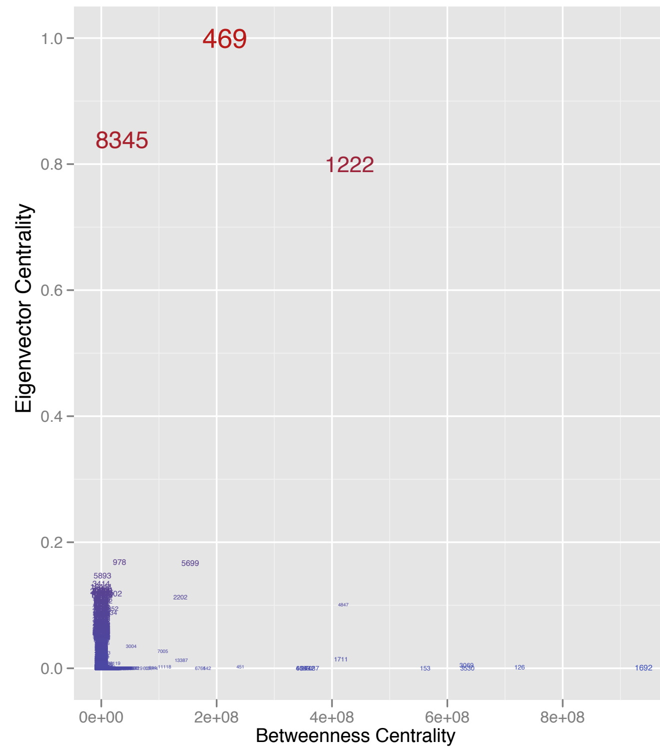


Figure 8.4.: This figure shows the key component analysis. The IDs of the most important components in the network analyzed by eigenvector centrality and betweenness centrality. Scaled by eigenvector centrality.

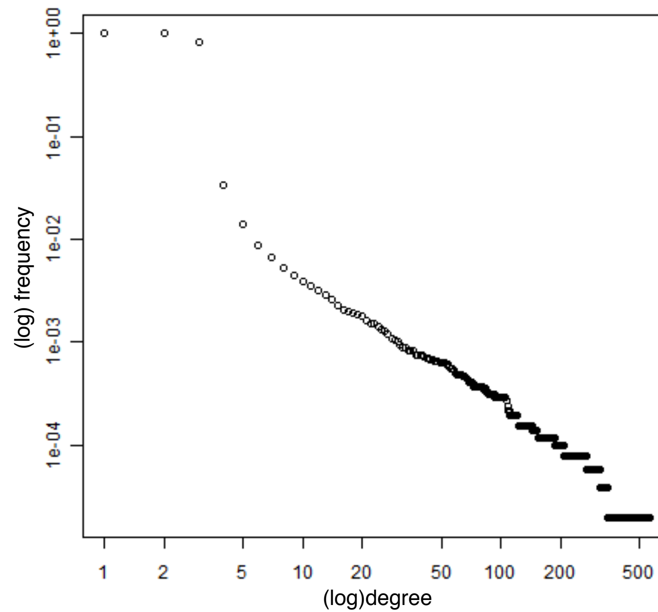


Figure 8.5.: This figure shows the degree distribution over the graph. Most of the components have a degree between one and four.

modules to aggregates) was initially driven by several structuring concepts e.g. the functional coherence or even organizational responsibility. The clustering approach intends to define new groups of components that focus on the reusability and quality of the components.

The result of the clustered graph is shown in figure 8.6 while groups of components are in the same color and arranged next to each other. Components are sized regarding their eigenvector centrality.

In Figure 8.7(a) the distribution of aggregates in a selected cluster is shown. This cluster is dominated by two aggregates (32 percent and 23 percent) to whom a very high degree of relationship could be detected. Another interesting fact that could be identified is the high number of components from different aggregates in the cluster. It shows a high degree of relationship between components of completely different aggregate groups. The graph of a smaller cluster with components from only four different aggregate groups is shown in Figure 8.7(b). Here, components which are so called *gatekeepers* between different aggregate groups have been identified.

Furthermore, two very big clusters where found. Analysis of these clusters showed that components of these clusters are mostly project specific modules with a weak degree of relationship. Consequently, the components grouped in this cluster are recommended to be moved into the project specific part of the aggregates (see figure 2.7(b)).

8.4. Application of the RPOSA Algorithm

In this section the RPOSA algorithm discussed in Section 6.2.1 is applied to the network generated in Section 8.2, which is refered to as *full network* in the following, as well as on subnetworks which define a specific context. This context can be a single aggregate (as defined in Section 2.4), a specific functional context (e.g. only gasoline or only diesel systems) or a cluster of customers. In case of an aggregate context the partitioning of modules to one aggregate is evaluated. In case of a functional context the partitioning of aggregates is analyzed for modules which are e.g. only used in gasoline or diesel projects and optimized regarding these. Similar applications to the analysis on customer clusters the RPOSA algorithm helps to decide if a partinioning for a specific context is meaningfull and favourable.

In order to determine the optimal variation among the control parameters for the particular network topography, a number of experiments where carried out. In Table 8.4 the variations of the tested values for the different control parameters as well as the final values for the example repository are listed. The experiments where carried out with two initial module partitionings (assignment of a class to a module). First the current aggregate partitioning was applied to the modules. Therefore the classes represent a specifc aggregate. This aggregate classification

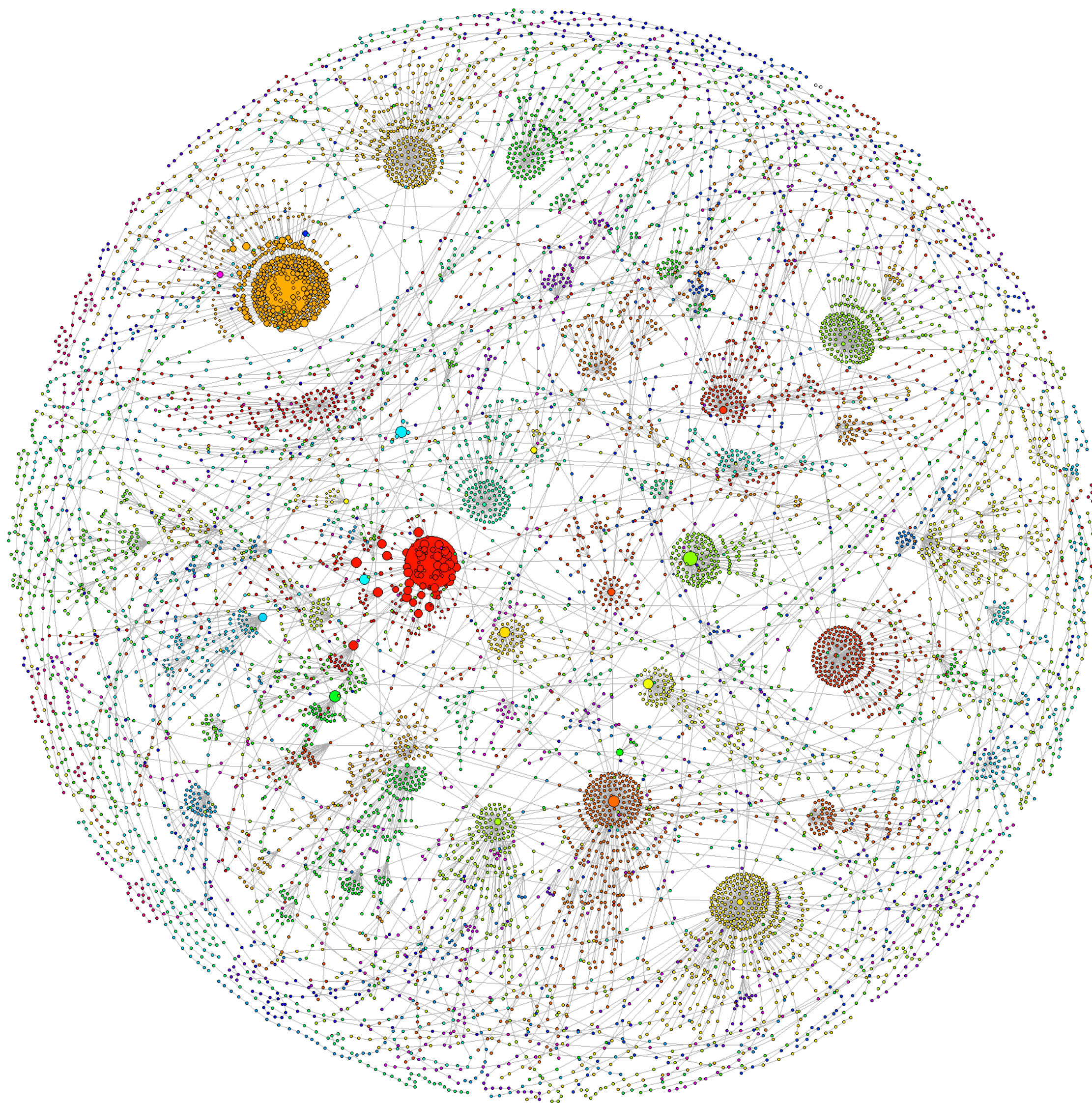
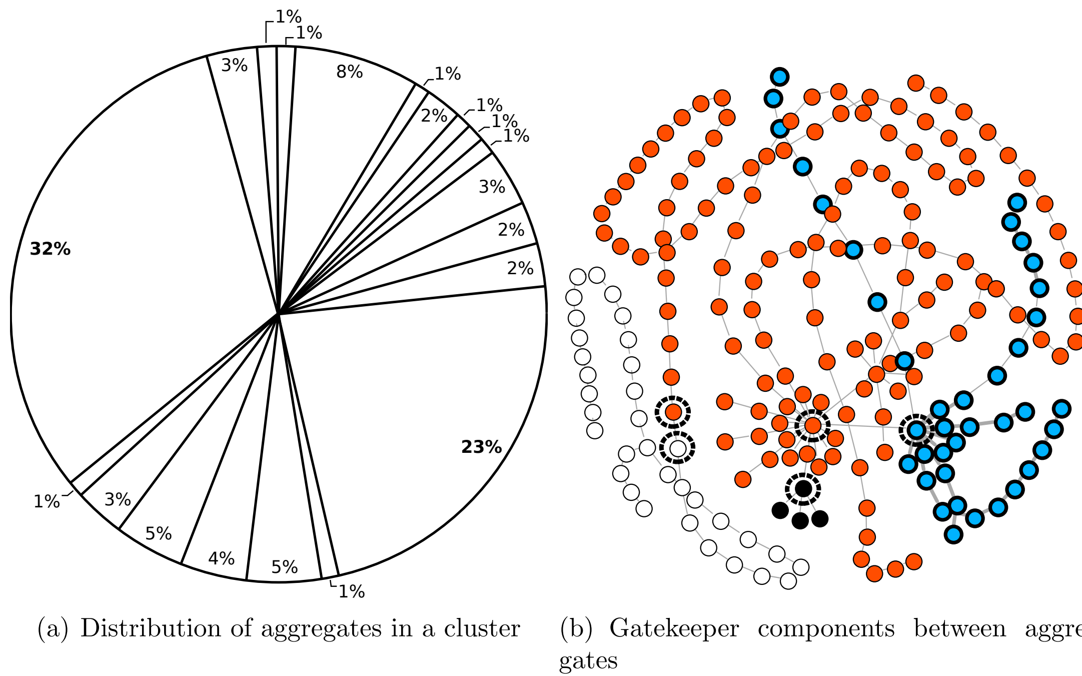


Figure 8.6.: The resulting graph after the edge betweenness community clustering



(a) Distribution of aggregates in a cluster

(b) Gatekeeper components between aggregates

Figure 8.7.: Figure (a) shows the distribution of different aggregates in a selected cluster. Two specific clusters have a very high share in this cluster. In total, 20 aggregates have been grouped to one cluster. Figure (b) shows a cluster with four different aggregates grouped together. Components, which act as gatekeepers between the different aggregates, could be identified.

was confronted with the assignment of 100 random classes to the modules. The results of the experiments showed that the initial clustering has a very low significance to the result of the optimization.

Parameter	Tested Values	Used Value
Start temperature T_0	300,280,250,220,200,150,120,100, 50,20,10,5,1	220
Final temperature T_{min}	0.1,0.01,0.001,0.0001,0.00001	0.0001
Cooling schedule α	0.99,0.98,0.97,0.95, 0.90, 0.85,0.80	$0.85 \forall T > 1; 0.98$
Local Iterations L_{max}	2000,1800,1500,1200,1000,800,600, 400,200	1000
Step width sw	$N(v)$, $\frac{\lvert N(v)\rvert}{2}$, 1,10,100,200	$\frac{\lvert N(v)\rvert}{2}$

Table 8.1.: Tested values for the control parameters and the final decisions for the full network.

The final configuration of the control parameters resulted in 456000 iterations and a total of 12 days running time (on an Intel Core 2 Duo Laptop with 2.4 Ghz and 8 GB RAM). The running time is composed of <1% for the generation of the graph (n=7881), ∼1% for the calculation of the component rank and ∼99% for execution of the RPOSA algorithm. The graph has 4.7 million edges and an average degree of 1197.6. Figure 8.8 shows the result of the RPOSA optimization while Figure 8.9 shows an extract after the first 100000 iterations. The dotted line represents the best population found in various test runs of the parameter evaluation. As discussed in Section 6.3 a significant trade-off in the runtime and the number of packages was observed in the run with the maximum fitness value.

The run shown in Figure 8.8 with the parameters of Table 8.4 did show the best results regarding fitness and number of packages. The initial fitness value (with 104 classes representing the aggregate configuration) of the full network is 754.291. After the optimization with the RPOSA algorithm the final fitness value is 18661.82. This resulted in 972 packages while 316 packages have a value which equals 0. 208 packages have a fitness value > 28.44, the arithmetic mean of the values, and are proposed as the new module packages. This represent a fitness value of 14808.

In the current architecture the modules are clustered to 100 aggregates. The clustering with the RPOSA algorithm produced about nine times the packages. This behaviour, to split the modules in smaller packages, is to be expected and while the effort to manage the packages might rise (the management of components is discussed in more detail in Chapter 7), the improvement in reusability justifies

the overhead in component management. Furthermore if only the most promising packages (e.g. a fitness value above the arithmetic value) are used as reusable packages the number of packages to manage drops to around twice the initial configuration. Beside the improvement of reusability a better flexibility is expected due smaller packages that can be used in more projects.

As discussed in Section 2.4, the decision how to cluster components is based on multiple objectives. The clustering presented in this secion is based solely on reuse information and design decisions of past projects. Therefore the RPOSA algorithm can only serve as a tool which recommends packages to an system architecture expert to support the decision process on which components should be packaged together.

Nevertheless, after the clustering the network analysis techniques discussed in Chapter 5 and applied in the previous section, can be reapplied to provide valuable information about the individual clusters. This is possible because the optimization algorithm preserves the network structure of the initial component reuse network. Because of the vast number of components in each individual cluster measures like the *reuse popularity*, *component package complexity* and the *component rank* can be applied to help the architect in the process of evaluating the clustering.

8.5. Summary: Case Study

In the previous sections the steps prior to the reuse-oriented development process where performed and the initial component repository was constructed. To follow the process described in Chapter 7, the system architecture of the engine management system has to be modeled in an architecture description language which provides interfaces to the component repository. At the time of this work several different architecture description languages and modeling approaches are beeing evaluated and no final choice has been made.

Component groups generated by the RPOSA algorithm have been subject to thorough reviews by architecture experts at Continental Automotive GmbH and have been proven to be meaningfull in terms of functional validity. On average an improvement of the overall reusability of about twenty five times the initial grouping of packages was observed while the number of packages was around twice as high (counting the most promising reuse packages) as the initial aggregate configuration.

Nevertheless the grouping of components to reusable packages is driven not only by reusability aspects but also by many constraints resulting from organizational structures or other aspects of the development process. Therefore the approaches applied to the engine management system can only serve as an expert system in the process of designing a future system architecture, supporting the architect by providing experience and guidance from past projects.

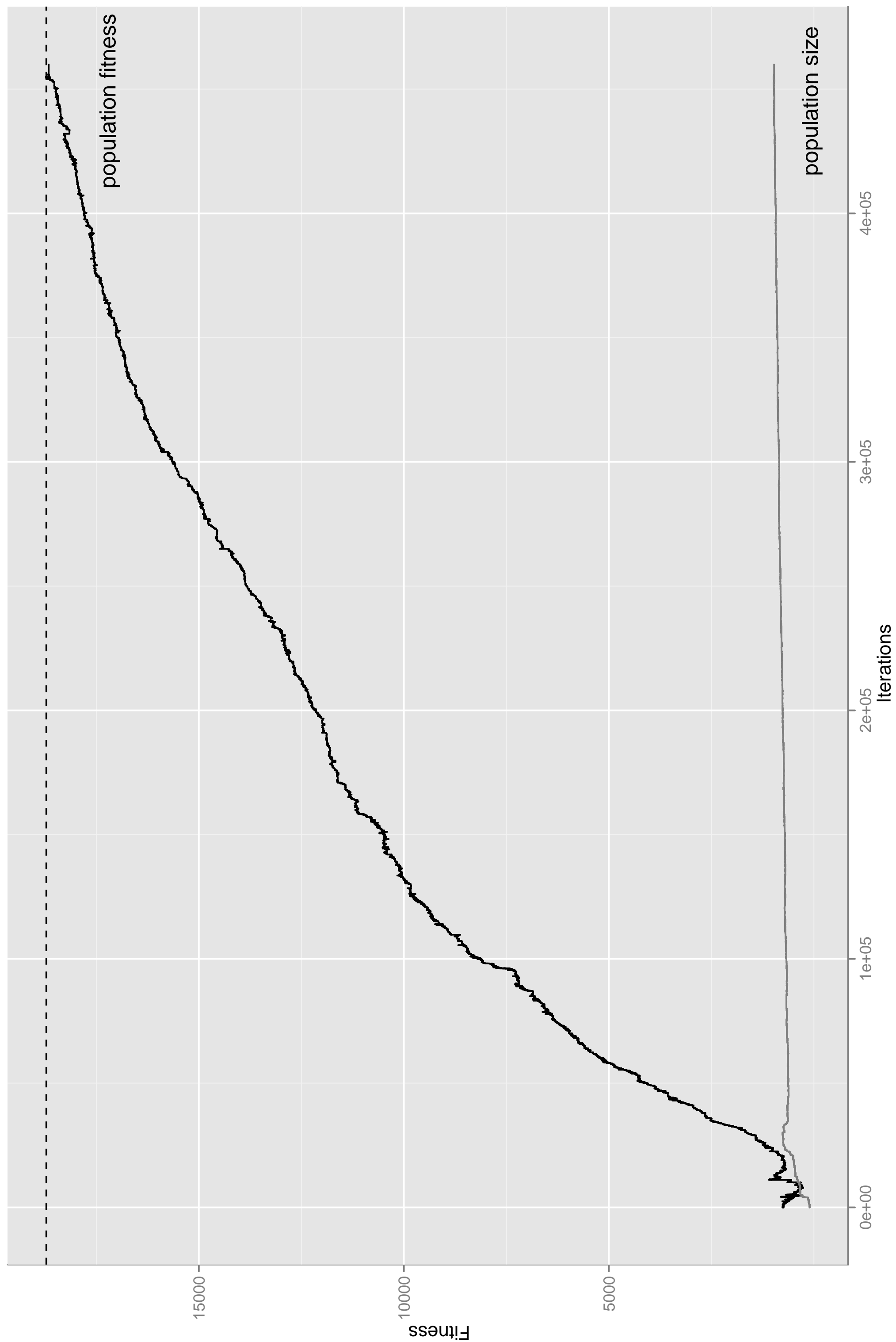


Figure 8.8.: RPOSA algorithm applied to the full network. The dotted line represents the best solution found (19208) in various test runs while the black line pictures the fitness of the population over the iterations (with a final value of 18661.82). The grey line shows the development of the number of solutions in the population (with an initial value of 104 and an final value of 972).

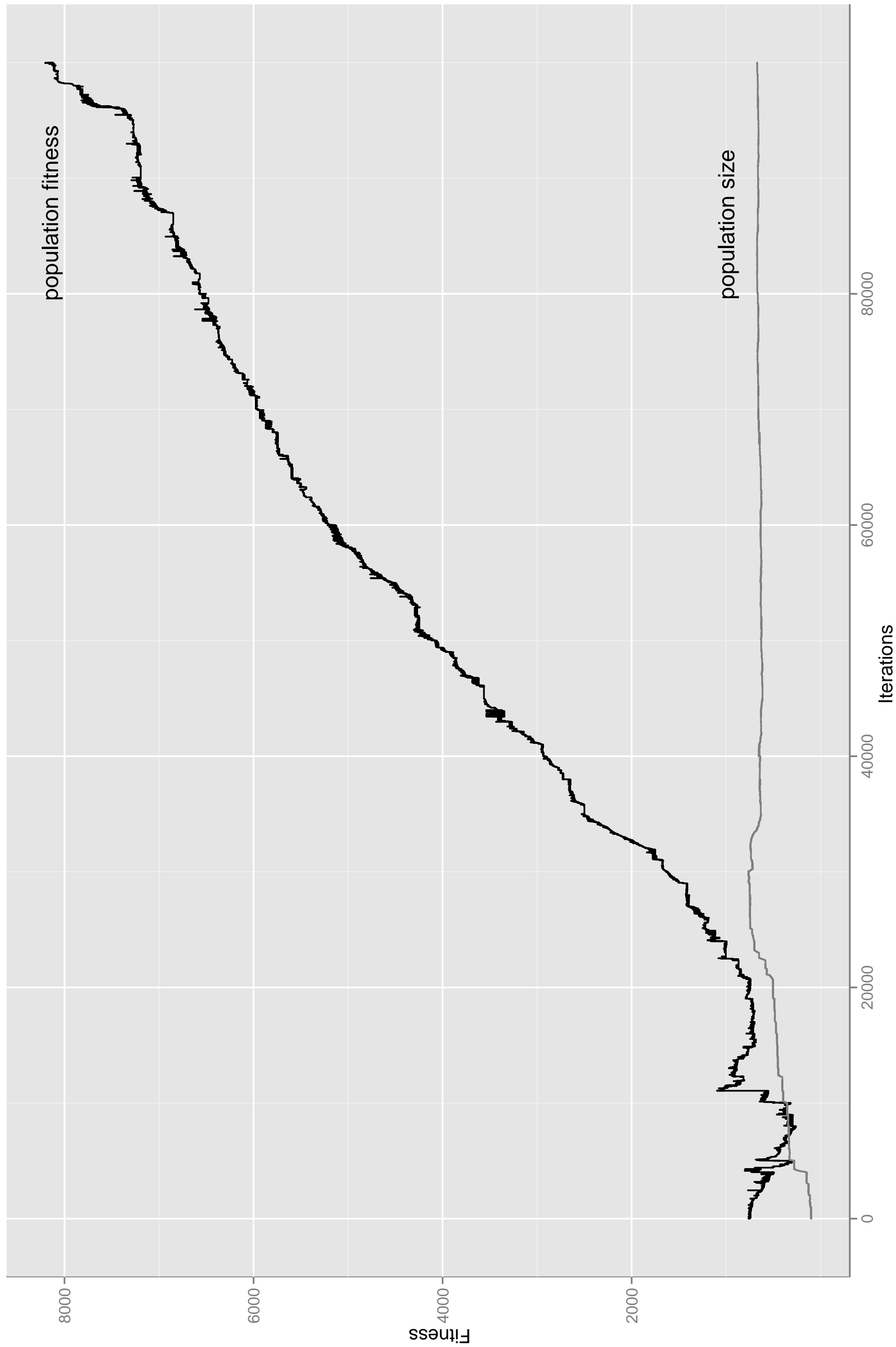


Figure 8.9.: This figure shows the first 100000 iterations of the RPOSA algorithm applied to the full network. This image pictures the exploration phase of the algorithm at the first 20000 iterations.

Part V.

Conclusions and Future Work

9. Conclusions and Future Work

This thesis has presented evaluation and clustering techniques in order to analyze component-based embedded software systems regarding reuse. Furthermore a model-based framework and a process to support system experts in the design of new solutions was proposed. This chapter summarizes theses techniques and points out interesting issues for future work on this topic.

9.1. Conclusions

The most important driver of innovation in modern cars are embedded systems. New-generation cars contain a huge amount of features which would not be possible without the support of electronic devices and their respective software. This demand for new features and functions led to an increasing complexity in the design and development of embedded systems. Due to this high complexity, the task of building such systems becomes increasingly challenging and raises major concerns in critical application domains like the automotive industry. In order to meet this challenge, component-based architectures where introduced to automotive embedded systems. Reusability is an important aspect in the development of these systems and is frequently seen as a powerful approach to develop high quality systems while reducing the complexity.

This thesis presents techniques to improve reusability in the systems described above and provides means to answer the following questions:

- How to evaluate component-based systems and in particular how to identify components which qualify for reuse?
- How to group components of existing solutions to reusable packages to reduce the integration effort for new solutions?
- How to support the handling of components in component-based architectures?

As for the evaluation of these component-based systems in order to improve reuse, two techniques are proposed in the thesis. Both techniques work with information on system components which is automatically collected from a company's project repository. This information represents the knowledge, experience and design de-

cisions of system architects of past and current projects. In order to be able to perform analysis on this data the information is transformed in a graph based structure, a so called component reuse network. The process to extract this information and generate a network is discussed in Chapter 4. In this component reuse network links between components are constructed based on reuse information and project scope.

The first technique, presented in Chapter 5, applies network analysis methods, derived from the research field of social network analysis and graph theory, on the component reuse network. The focus of this technique lies on the connections or relationships between components and not on the attributes of individual components. This technique provides means to identify important and central components which can be used to understand potential flows of information or resources as well as constraints on components in a network. Furthermore it allows to apply quantitative and qualitative methods to explore or make inferences about the role of individual components in the network. Clustering techniques support system architects to identify component groups and subgroups for the generation of reusable packages of components.

For large project repositories, the feasibility of the proposed network analysis concepts might be impeded due to very dense graphs. In such cases, the system architect has to resort to an optimization heuristic, the second component reuse clustering technique, proposed in Chapter 6 in this thesis. This optimization approach, a recombinative population-oriented simulated annealing algorithm (RPOSA), selects and groups components in order to optimize reuse in the component-based software architecture. The optimization algorithm operates on the generated component reuse network as well. On this graph, recombination operations are performend in order to optimize the initial clusters according to a defined objective function. The final clustering is evaluated according the objective function and analyzed with the network analysis measures proposed in the first technique.

To integrate the evaluation and clustering techniques in a company's development process and to support the handling (e.g. storing and retrival) of individual components a model-based architecture framework has been proposed in Chapter 7. In this chapter a process to improve reusability is proposed and the extension of an architecture description language to support the modeling of multicore systems is briefly discussed.

In order to demonstrate the feasibility of the approaches, network analysis and combinatorial optimization have been applayed to a realistic project repository in a case study in Chapter 8. In this case study both techniques where applied to the component-based engine management system architecture of the Continental Automotive Corporation resulting in valuable improvement proposals for the company's component architecture.

9.2. Future Work

This section presents topics and implications which should be investigated further in order to improve, refine and extend the techniques and methodologies presented in this thesis.

- *Performance optimizations of the RPOSA algorithm:* The running time and performance of the RPOSA algorithm could be improved with a parallelization of the algorithm following the method presented in [Ram et al.1996]. Experiments with a prototype implementation resulted in promising speedups between 2 and 4.5 times faster as the sequential implementation. Another performance optimization could result from the implementation of the algorithm in C or C++ using the Boost Network Library for the management of the graph. The current implementation uses Python 3.1 and the NetworkX graph library. While the implementation with python is sufficient for early experiments of the approach and easy to integrate in the development process, a moderate performance gain could be expected from an implementation in C/C++.
- *On-line adaption of parameters:* As argued the right choice of parameters is crucial for the success of the algorithm. Especially in real-world applications where the runtime of the algorithm is important. Nolle et. al. and Fogel [Nolle et al.2001, Fogel2002] discussed the possibility of the adaption of a parameter (e.g. the maximum step width) to other SA parameters like temperature or iterations or even to the fitness landscape at runtime. Nolle et. al. showed that this on-line adaption, at least for the adaption of the maximum step width for the neighbor selection, can significantly reduce the search time through the solution space and improve the overall fitness.
- *Multi-Objective Optimization:* Due to the multiobjective nature of real world problems the extension of the RPOSA algorithm to support multiobjective optimization could improve the results of the approach and the acceptance of an automatic clustering of components by the developers. Possible objectives could be other quality attributes (beside reuse) like safety or reliability. On the contrary multiobjective optimization has, unlike single objective, multiple goals and makes it difficult to measure the quality of the solution and remains an important research topic for scientists and researchers.
- *Different network construction techniques:* The expressiveness of the component reuse network could be improved by using input and output data of the interfaces of components for the construction of the network. The density of the network would be decreased and thus, network analysis techniques would provide better results.
- *Integration with ADLs and standard development environments:* To improve the search an retrival process for components and provide a seamless integra-

tion with the architecture description language an integration of the component repository a framework (e.g. the Eclipse Framework) is suggested. Such an integration would also contribute to the overall usability of the approaches presented.

- *Design-Space exploration:* A possible extension of the optimization approach would be the integration of design-space exploration means. With the possibility for experts to alter the component reuse network in a controlled manner, studies about the impact of architectural changes to the overall architecture could be conducted. Experts at Continental Corporation argued that this would be a valuable tool for the architecture development department.

Part VI.

Appendix

A. Example Project Repository

Project	Components
Project 1	c_1, c_2, c_3,c_4, c_5
Project 2	c_1, c_3, c_6, c_7, c_8, c_9
Project 3	c_3, c_4, c_8, c_{10}, c_{11}, c_{12}, c_{13}
Project 4	c_1, c_2, c_3, c_4, c_5, c_{14}, c_{15}, c_{16}, c_{17}, c_{18}, c_{19}, c_{20}
Project 5	c_{10}, c_{11}, c_{23}, c_{24}, c_{25}
Project 6	c_8, c_{14}, c_{15}, c_{26}
Project 7	c_1, c_3, c_{27}, c_{28}, c_{29}, c_{30}, c_{31}, c_{32}, c_{33}, c_{34}, c_{35}, c_{36}
Project 8	c_{10}, c_{33}, c_{36}, c_{37}, c_{38}, c_{39}, c_{40}
Project 9	c_4, c_5, c_{14}, c_{37}, c_{41}, c_{42}, c_{43}
Project 10	c_{29}, c_{30}, c_{31}, c_{47}, c_{48}, c_{49}, c_{50}

Table A.1.: Example of a companies possible project repository were individual components are associated to a particular project.

Bibliography

[Aarts and Korst1988] Aarts, E. and Korst, J. (1988). Simulated annealing and Boltzmann machines.

[Aarts and Van Laarhoven1985] Aarts, E. and Van Laarhoven, P. (1985). Statistical cooling: A general approach to combinatorial optimization problems. *Philips J. Res.*, 40(4):193–226.

[Abbasi et al.2010] Abbasi, B., Niaki, S., Khalife, M., and Faize, Y. (2010). A hybrid variable neighborhood search and simulated annealing algorithm to estimate the three parameters of the Weibull distribution. *Expert Systems with Applications*.

[Abowd et al.1996] Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., and Zaremski, A. (1996). Recommended best industrial practice for software architecture evaluation. *Software Engineering Institute Technical Report, CMU/SEI-96-TR-025*.

[Ackley et al.1985] Ackley, D., Hinton, G., and Sejnowski, T. (1985). A learning algorithm for boltzmann machines*. *Cognitive science*, 9(1):147–169.

[Aerospace2004] Aerospace, S. (2004). Architecture Analysis & Design Language (AADL). *AS-5506, SAE International*.

[Albert et al.2002] Albert, C., Brownsword, L., Bentley, C., Bono, T., Morris, E., et al. (2002). Evolutionary process for integrating COTS-based systems (EPIC): An overview. *SEI CMU, Pittsburgh, PA, Technical Report CMU/SEI-2002-TR-009*.

[Alves et al.2005] Alves, V., Matos Jr, P., Cole, L., Borba, P., and Ramalho, G. (2005). Extracting and evolving mobile games product lines. *Software Product Lines*, pages 70–81.

[Anquetil2000] Anquetil, N. (2000). A comparison of graphs of concept for reverse engineering. In *Proceedings of the 8th International Workshop on Program Comprehension*, page 231. IEEE Computer Society.

[Babar et al.2004] Babar, M., Zhu, L., and Jeffery, R. (2004). A framework for classifying and comparing software architecture evaluation methods. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 309–318.

[Bachmann et al.2000] Bachmann, F., Bass, L., Buhrman, C., Cornella-Dorda, S., Long, F., Robert, J., Seacord, R., and Wallnau, K. (2000). Volume II: Technical concepts of component-based software engineering. Technical report, Citeseer.

[Banker et al.1993] Banker, R., Kauffman, R., and Zweig, D. (1993). Repository evaluation of software reuse. *IEEE Transactions on Software Engineering*, 19(4):379–389.

[Barry1996] Barry, B. (1996). Component-Based Development of Smalltalk Applications. pages 25–26.

[Bartholet et al.2005] Bartholet, R., Brogan, D., and Reynolds Jr, P. (2005). The computational complexity of component selection in simulation reuse. In *Proceedings of the 37th conference on Winter simulation*, pages 2472–2481. Winter Simulation Conference.

[Basili et al.1992] Basili, V., Caldiera, G., and Cantone, G. (1992). A reference architecture for the component factory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):80.

[Basili and Rombach1988] Basili, V. and Rombach, H. (1988). Towards a comprehensive framework for reuse: A reuse-enabling software evolution environment. In *NASA, Goddard Space Flight Center, Proceedings of the Thirteenth Annual Software Engineering Workshop 45 p(SEE N 91-10606 01-61)*.

[Bengtsson and Bosch1999] Bengtsson, O. and Bosch, J. (1999). Architecture level prediction of software maintenance. In *csmr*, page 139. Published by the IEEE Computer Society.

[Bengtsson and Bosch1998] Bengtsson, P. and Bosch, J. (1998). Scenario-based software architecture reengineering. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 308–317.

[Bengtsson et al.2004] Bengtsson, P., Lassing, N., Bosch, J., and van Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147.

[Biggerstaff and Perlis1989] Biggerstaff, T. and Perlis, A. (1989). Software reusability: vol. 1, concepts and models.

[Bilbro et al.] Bilbro, G., Mann, R., Miller, T., Snyder, W., Van den Bout, D., and White, M. Optimization by mean field annealing. *Advances in neural information processing systems*, 1:91–98.

[Bonacich2007] Bonacich, P. (2007). Some unique properties of eigenvector centrality. *Social networks*, 29(4):555–564.

[Bouyssounouse and Sifakis2005] Bouyssounouse, B. and Sifakis, J. (2005). *Embedded systems design: the ARTIST roadmap for research and development*.

Springer Verlag.

[Braun] Braun, C. Reuse, in John J. Marciniak, editor. *Encyclopedia of Software Engineering*, 2:1055–1069.

[Braun1992] Braun, C. (1992). NATO standard for the development of reusable software components. *NATO Communications and Information Systems Agency.*

[Brun et al.2008] Brun, M., Delatour, J., and Trinquet, Y. (2008). Code generation from AADL to a real-time operating system: an experimentation feedback on the use of model transformation. In *Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, pages 257–262. IEEE Computer Society.

[Burton et al.1987] Burton, B., Aragon, R., Bailey, S., Koehler, K., and Mayes, L. (1987). The reusable software library. *IEEE software*, 4(4):25–33.

[Černỳ1985] Černỳ, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51.

[Chidamber et al.1994] Chidamber, S., Kemerer, C., and MIT, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493.

[Claraz et al.2004] Claraz, D., Eppinger, K., and Berentroth, L. (2004). Reuse strategy at siemens VDO automotive: the EMS 2 powertrain platform architecture. *Ingenieurs de l'Automobile*, 767.

[Clarke et al.2003] Clarke, J., Dolado, J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., et al. (2003). Reformulating software engineering as a search problem. *IEE Proceedings-Software*, 150(3):161–175.

[Clements and Kazman2003] Clements, P. and Kazman, R. (2003). *Software Architecture in Practices.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[Comella-Dorda et al.2002] Comella-Dorda, S., Dean, J., Morris, E., and Oberndorf, P. (2002). A process for COTS software product evaluation. *COTS-Based Software Systems*, pages 86–96.

[Cooper1994] Cooper, J. (1994). Reuse-the business implications. *Marciniak Mar94*, page 10711077.

[Crnkovic and Larsson2002] Crnkovic, I. and Larsson, M. (2002). *Building reliable component-based software systems.* Artech House Publishers.

[Davis1992] Davis, T. (1992). Toward a Reuse Maturity Model,". In *Proceedings of the WISR 5th Annual Workshop on Software Reuse,(Palo Alto, California).*

[Deubzer et al.2010] Deubzer, M., Hobelsberger, M., Mottok, J., Schiller, F., Dumke, R., Siegle, M., Margull, U., Niemetz, M., and Wirrer, G. (2010). Modeling and Simulation of Embedded Real-Time Multi-Core Systems. In *Proceedings of the 3rd Embedded Software Engineering Congress.*

[Devanbu et al.1991] Devanbu, P., Brachman, R., and Selfridge, P. (1991). LaSSIE: a knowledge-based software information system. *Communications of the ACM*, 34(5):34–49.

[Dissaux2005] Dissaux, P. (2005). AADL model transformations. In *Proc DASIA 2005 Conference in Edinburgh, UK*. Citeseer.

[Dobrica and Niemel2002] Dobrica, L. and Niemel (2002). A survey on software architecture analysis methods. *IEEE Transactions on software Engineering*, pages 638–653.

[Duda et al.2001] Duda, R., Hart, P., and Stork, D. (2001). *Pattern classification*, volume 2. Citeseer.

[Dumke2003] Dumke, R. (2003). *Software engineering: Eine Einfuehrung fuer Informatiker und Ingenieure; Systeme, Erfahrungen, Methoden, Tools;[mit Online-Service zum Buch].* Vieweg.

[Dumke et al.2002] Dumke, R., Abran, A., Bundschuh, M., and Symons, C. (2002). *Software Measurement and Estimation*. Vieweg.

[Dumke et al.1996] Dumke, R., Foltin, E., Koeppe, R., and Winkler, A. (1996). *Measurement-based Object-oriented Software Development of the Software Project" Software Measurement Laboratory"*. Citeseer.

[Dumke and Grigoleit1997] Dumke, R. and Grigoleit, H. (1997). Efficiency of CAMEtools in software quality assurance. *Software Quality Journal*, 6(2):157–169.

[Dumke and Schmietendorf2000] Dumke, R. and Schmietendorf, A. (2000). Metriken-basierte Bewertung von Software-Komponenten. *Softwaretechnik-Trends*, 20(4).

[Dumke and Winkler1997] Dumke, R. and Winkler, A. (1997). Managing the component-based software engineering with metrics. *sast*, page 0104.

[Ebert and Dumke2007] Ebert, C. and Dumke, R. (2007). Software Measurement. *Controlling for IT and Software.*

[Esteva1995] Esteva, J. (1995). Automatic identification of reusable components. In *Computer-Aided Software Engineering, 1995. Proceedings., Seventh International Workshop on*, pages 80–87.

[Ezran et al.2002] Ezran, M., Morisio, M., and Tully, C. (2002). *Practical software reuse.* Springer Verlag.

[Feiler et al.2006a] Feiler, P., Greenhouse, A., and List, C. (2006a). OSATE Plug-in Development Guide. *CMU. Pittsburgh.*

[Feiler et al.2006b] Feiler, P., Lewis, B., and Vestal, S. (2006b). The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems. In *2006 IEEE Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1206–1211.

[Fenton and Pfleeger1998] Fenton, N. and Pfleeger, S. (1998). *Software metrics: a rigorous and practical approach.* PWS Publishing Co. Boston, MA, USA.

[Fischer1998] Fischer, G. (1998). Seeding, evolutionary growth and reseeding: Constructing, capturing and evolving knowledge in domain-oriented design environments. *Automated Software Engineering*, 5(4):447–464.

[Fogel2002] Fogel, D. (2002). An introduction to simulated evolutionary optimization. *Neural Networks, IEEE Transactions on*, 5(1):3–14.

[Fox et al.2004] Fox, M., Brogan, D., and Reynolds Jr, P. (2004). Approximating component selection. In *Proceedings of the 36th conference on Winter simulation*, pages 429–434. Winter Simulation Conference.

[Frakes and Gandel1990] Frakes, W. and Gandel, P. (1990). Representing reusable software. *Information and Software Technology*, 32(10):653–664.

[Frakes and Nejmeh1986] Frakes, W. and Nejmeh, B. (1986). Software reuse through information retrieval. In *ACM SIGIR Forum*, volume 21, pages 30–36. ACM New York, NY, USA.

[Frakes and Pole1994] Frakes, W. and Pole, T. (1994). An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, pages 617–630.

[Frakes and Terry1996] Frakes, W. and Terry, C. (1996). Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435.

[Gaffney et al.1989] Gaffney, J. et al. (1989). Software reuse–key to enhanced productivity: some quantitative models. *Information and Software Technology*, 31(5):258–267.

[Geman and Geman1993] Geman, S. and Geman, D. (1993). Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images*. *Journal of Applied Statistics*, 20(5):25–62.

[Girvan and Newman2002] Girvan, M. and Newman, M. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821.

[Goldberg and Rubin1995] Goldberg, A. and Rubin, K. (1995). Succeeding with objects: decision frameworks for project management.

[Haghpanah et al.2008] Haghpanah, N., Moaven, S., Habibi, J., Kargar, M., and Yeganeh, S. (2008). Approximation algorithms for software component selection problem. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 159–166. IEEE.

[Harrison et al.1998] Harrison, R., Counsell, S., and Nithi, R. (1998). An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, pages 491–496.

[Heineman and Councill2001] Heineman, G. and Councill, W. (2001). *Component-based software engineering: putting the pieces together*. Addison-Wesley USA.

[Hilliard2000] Hilliard, R. (2000). IEEE-Std-1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE, http://standards. ieee. org*.

[Hobelsberger2007] Hobelsberger, M. (2007). Vergleich der Architekturbeschreibungssprachen SAE AADL und EAST ADL sowie ihrer Beziehung zu AUTOSAR. Master's thesis, University of Applied Sciences Regensburg.

[Hobelsberger et al.2010] Hobelsberger, M., Dumke, R., Mottok, J., Niemetz, M., and Wirrer, G. (2010). An Experiece-Based Repository of Reusable Components for an Component-Based Automotive Software System. In *Proceedings of the IWSM/MetriKon/Mensura 2010*, pages 218–240.

[Hobelsberger and Mottok2009] Hobelsberger, M. and Mottok, J. (2009). Software Qualitaet eine Glaubensfrage? Ein Ueberglick ueber die Modelle der Softwarezuverlaessigkeit. In *Proceedings of the 2nd Embedded Software Engineering Congress*, ISBN 978-3-8343-2402-3, pages 134–148.

[Hobelsberger et al.2008] Hobelsberger, M., Mottok, J., and Dumke, R. (2008). Modellbasierte Sicherheitsanalysen von Software-Architekturen. In *Proceedings of the 1th Embedded Software Engineering Congress*, ISBN 978-3-8343-2401-6, pages 436–443.

[Hobelsberger et al.2007] Hobelsberger, M., Mottok, J., and Kuntz, S. (2007). Architekturmodellierung: Vergleich von EAST ADL und SAE AADL. *Hanser Automotive 7-8.2007*, page 4.

[Hoover and Khosla1996] Hoover, C. and Khosla, P. (1996). An analytical approach to change for the design of reusable real-time software. In *Object-Oriented Real-Time Dependable Systems, 1996. Proceedings of WORDS'96., Second Workshop on*, pages 144–151. IEEE.

[Hoover et al.1999] Hoover, C., Khosla, P., and Siewiorek, D. (1999). Analytical design of reusable software components for evolvable, embedded applications. In

Application-Specific Systems and Software Engineering and Technology, 1999. ASSET'99. Proceedings. 1999 IEEE Symposium on, pages 170–177. IEEE.

[Huang et al.1986] Huang, M., Romeo, F., and Sangiovanni-Vincentelli, A. (1986). An efficient general cooling schedule for simulated annealing. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 381–384.

[Hutchens and Basili1985] Hutchens, D. and Basili, V. (1985). System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757.

[Ingber1993] Ingber, L. (1993). Simulated annealing: Practice versus theory. *Mathematical and computer modelling*, 18(11):29–57.

[Israel and Koutsougeras2002] Israel, P. and Koutsougeras, C. (2002). An annealing approach to associative recall in the CBM model. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 633–638. IEEE.

[Kazman et al.1996] Kazman, R., Abowd, G., Bass, L., and Clements, P. (1996). Scenario-based analysis of software architecture. *IEEE software*, 13(6):47–55.

[Kazman et al.1998] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, J. (1998). The architecture tradeoff analysis method. In *iceccs*, page 0068. Published by the IEEE Computer Society.

[Kazman et al.2000] Kazman, R., Klein, M., and Clements, P. (2000). ATAM: Method for architecture evaluation. *CMU/SEI*.

[Kirkpatrick and Gelatt1983] Kirkpatrick, MP Vecchi, S. and Gelatt, C. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

[Knoke and Yang2008] Knoke, D. and Yang, S. (2008). *Social network analysis*. Sage Publications.

[Krueger1992] Krueger, C. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183.

[Kunz2010] Kunz, M. (2010). Framework for a service-oriented measurement infrastructure. *Shaker Publ., Aachen*.

[Laarhoven et al.1987] Laarhoven, P., van Laarhoven, P., and Aarts, E. (1987). *Simulated annealing: theory and applications*. Springer.

[Lasnier et al.] Lasnier, G., Zalila, B., Pautet, L., and Hugues, J. OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. *Reliable Software Technologies–Ada-Europe 2009*, pages 237–250.

[Lassing et al.1999] Lassing, N., Rijsenbrij, D., and van Vliet, H. (1999). On Software Architecture Analysis of Flexibility, Complexity of Changes: Size Isnt Ev-

erything. In *Proc. Second Nordic Software Architecture Workshop NOSA*, volume 99, pages 1103–1581.

[Li et al.1993] Li, W., Henry, S., Drives, K., and Radford, V. (1993). Maintenance metrics for the object oriented paradigm. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 52–60.

[Lim1994] Lim, W. (1994). Effects of reuse on quality, productivity, and economics. *IEEE software*, 11(5):23–30.

[Lim1998] Lim, W. (1998). *Managing Software Reuse.* Prentice Hall.

[Lindig and Snelting1997] Lindig, C. and Snelting, G. (1997). Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th international conference on Software engineering*, pages 349–359. ACM.

[Lother2007] Lother, M. (2007). From Software Measurement to E-measurement: A Functional Size Measurement-Oriented Approach for Software Management.

[Lundy and Mees1986] Lundy, M. and Mees, A. (1986). Convergence of an annealing algorithm. *Mathematical programming*, 34(1):111–124.

[Lung et al.1997] Lung, C., Bot, S., Kalaichelvan, K., and Kazman, R. (1997). An approach to software architecture analysis for evolution and reusability. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 15. IBM Press.

[Maarek et al.] Maarek, Y., Berry, D., and Kaiser, G. An Information Retrieval Approach for Automatically Constructing Software Libraries (PDF). *IEEE Transactions on software Engineering*, 17(8).

[Maarek et al.1991] Maarek, Y., Berry, D., and Kaiser, G. (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on software Engineering*, pages 800–813.

[Mancebo and Andrews2005] Mancebo, E. and Andrews, A. (2005). A strategy for selecting multiple components. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1505–1510. ACM.

[McCarey et al.2008] McCarey, F., Ó Cinnéide, M., and Kushmerick, N. (2008). Knowledge reuse for software reuse. *Web Intelligence and Agent Systems*, 6(1):59–81.

[McClure and McClure2001] McClure, C. and McClure, D. (2001). *Software Reuse: a standards-based guide.* IEEE Computer Society, Los Alamitos, Calif.; Tokyo.

[McIlroy et al.1969] McIlroy, M., Buxton, J., Naur, P., and Randell, B. (1969). Mass produced software components. *Software Engineering Concepts and Techniques*, pages 88–98.

[Merkl et al.1994] Merkl, D., Tjoa, A., and Kappel, G. (1994). Learning the semantic similarity of reusable software components. In *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pages 33–41. IEEE.

[Metropolis et al.1953] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E., et al. (1953). Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087.

[Mili et al.1994] Mili, A., Mili, R., and Mittermeir, R. (1994). Storing and retrieving software components: A refinement based system. In *Proceedings of the 16th international conference on Software engineering*, pages 91–100. IEEE Computer Society Press.

[Mili et al.1995] Mili, H., Mili, F., and Mili, A. (1995). Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562.

[Mohagheghi and Conradi2007] Mohagheghi, P. and Conradi, R. (2007). Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12(5):471–516.

[Molter1999] Molter, G. (1999). Integrating SAAM in domain-centric and reuse-based development processes. In *Proceedings of the 2nd Nordic Workshop on Software Architecture, Ronneby*, pages 1–10. Citeseer.

[Morisio et al.2002] Morisio, M., Ezran, M., and Tully, C. (2002). Success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*, 28(4):340–357.

[Morris et al.2008] Morris, O., Lee, M., and Constantinides, A. (2008). Graph theory for image analysis: An approach based on the shortest spanning tree. *Communications, Radar and Signal Processing, IEE Proceedings F*, 133(2):146–152.

[Muller et al.1993] Muller, H., Orgun, M., Tilley, S., and Uhl, J. (1993). A reverse engineering approach to subsystem structure identification. *Practice*, 5(4):181–204.

[Newman2004] Newman, M. (2004). Detecting community structure in networks. *The European Physical Journal B-Condensed Matter and Complex Systems*, 38(2):321–330.

[Newman2006] Newman, M. (2006). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577.

[Nolle et al.1999] Nolle, L., Armstrong, A., Hopgood, A., and Ware, A. (1999). Optimum work roll profile selection in the hot rolling of wide steel strip using computational intelligence. *Computational Intelligence*, pages 435–452.

[Nolle et al.2001] Nolle, L., Goodyear, A., Hopgood, A., Picton, P., and Braithwaite, N. (2001). On step width adaptation in simulated annealing for continuous parameter optimisation. *Computational Intelligence. Theory and Applications*, pages 589–598.

[Nolle et al.2002] Nolle, L., Goodyear, A., Hopgood, A., Picton, P., and Braithwaite, N. (2002). Automated control of an actively compensated Langmuir probe system using simulated annealing. *Knowledge-Based Systems*, 15(5-6):349–354.

[Ostertag et al.1992] Ostertag, E., Hendler, J., Díaz, R., and Braun, C. (1992). Computing similarity in a reuse library system: an AI-based approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(3):228.

[Page et al.1998] Page, L., Brin, S., Motwani, R., and Winograd, T. (1998). The pagerank citation ranking: Bringing order to the web.

[Parnas1972] Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1058.

[Petty et al.2003] Petty, M., Weisel, E., and Mielke, R. (2003). Computational complexity of selecting components for composition. In *Proceedings of the Fall 2003 Simulation Interoperability Workshop*, pages 14–19.

[Poulin1997] Poulin, J. (1997). *Measuring software reuse*. Addison-wesley.

[Prieto-Diaz and Freeman1987] Prieto-Diaz, R. and Freeman, P. (1987). Classifying software for reusability. *IEEE software*, 4(1):6–16.

[Puerta1997] Puerta, A. (1997). A model-based interface development environment. *Software, IEEE*, 14(4):40–47.

[Ram et al.1996] Ram, D., Sreenivas, T., and Subramaniam, K. (1996). Parallel simulated annealing algorithms. *Journal of parallel and distributed computing*, 37(2):207–212.

[Rombach2003] Rombach, D. (2003). Software nach dem Baukastenprinzip. *Fraunhofer Magazin*, pages 30–31.

[Rosenberg and Hyatt1997] Rosenberg, L. and Hyatt, L. (1997). Software Quality Metrics for Object-Oriented Environments. *Crosstalk Journal, April*.

[Russell and Norvig2009] Russell, S. and Norvig, P. (2009). *Artificial intelligence: a modern approach*. Prentice hall.

[Sametinger1997] Sametinger, J. (1997). *Software engineering with reusable components*. Springer Verlag.

[Schaetz et al.2002] Schaetz, B., Pretschner, A., Huber, F., and Philipps, J. (2002). Model-based development of embedded systems. *Advances in Object-Oriented Information Systems*, pages 331–336.

[Schmietendorf et al.2002] Schmietendorf, A., Dumke, R., Dimitrov, E., and Nakonz, S. (2002). *Bewertungsaspekte der komponentenorientierten Softwareentwicklung am Beispiel von Java-Komponenten*. Otto-von-Guericke-Univ. Magdeburg, Dekan, Fak. fuer Informatik.

[Schwanke1991] Schwanke, R. (1991). An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th international conference on Software engineering*, pages 83–92. IEEE Computer Society Press Los Alamitos, CA, USA.

[Selic2003] Selic, B. (2003). The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25.

[Sodhi and Sodhi1999] Sodhi, J. and Sodhi, P. (1999). *Software Reuse - Domain Analysis and Design Process*. McGraw-Hill.

[Stevens et al.1974] Stevens, W., Myers, G., and Constantine, L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.

[Sugumaran and Storey2003] Sugumaran, V. and Storey, V. (2003). A semantic-based approach to component retrieval. *ACM SIGMIS Database*, 34(3):8–24.

[SURI and Garg2009] SURI, D. and Garg, N. (2009). Software reuse metrics: Measuring component independence and its applicability in software reuse. *IJCSNS*, 9(5):237.

[Szyperski1998] Szyperski, C. (1998). Component Software: Beyond Object-Oriented Software. *Reading, MA: ACM/Addison-Wesley*.

[Taylor et al.2009] Taylor, R., Medvidovic, N., and Dashofy, E. (2009). Software Architecture: Foundations, Theory, and Practice.

[Torngren et al.2005] Torngren, M., Chen, D., and Crnkovic, I. (2005). Component-based vs. model-based development: a comparison in the context of vehicular embedded systems. In *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 432–440. IEEE.

[Van Deursen and Kuipers1999] Van Deursen, A. and Kuipers, T. (1999). Identifying objects using cluster and concept analysis.

[Vescan et al.2008] Vescan, A., Grosan, C., and Pop, H. (2008). Evolutionary algorithms for the component selection problem. In *Database and Expert Systems Application, 2008. DEXA'08. 19th International Conference on*, pages 509–513. IEEE.

[Wasserman1996] Wasserman, A. (1996). Toward a discipline of software engineering. *IEEE Software*, 13(6):23–31.

[Wasserman and Faust1994] Wasserman, S. and Faust, K. (1994). *Social network analysis: Methods and applications*. Cambridge Univ Pr.

[Wegner] Wegner, P. Varieties of reusability. *Tutorial: Software Reusability*, pages 24–38.

[Weisbecker2002] Weisbecker, A. (2002). *Software-Management f "ur komponentenbasierte Software-Entwicklung*. Jost-Jetter.

[Wu and Sloane2002] Wu, Q. and Sloane, T. (2002). CMOS leaf-cell design using simulated annealing. In *Circuits and Systems, 1992., Proceedings of the 35th Midwest Symposium on*, pages 1516–1519. IEEE.

[Xie] Xie, G. Decompositional verification of Component-based Systems-A Hybrid approach. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 414–417. IEEE.